MATLAB® Compiler™

User's Guide

R2013a
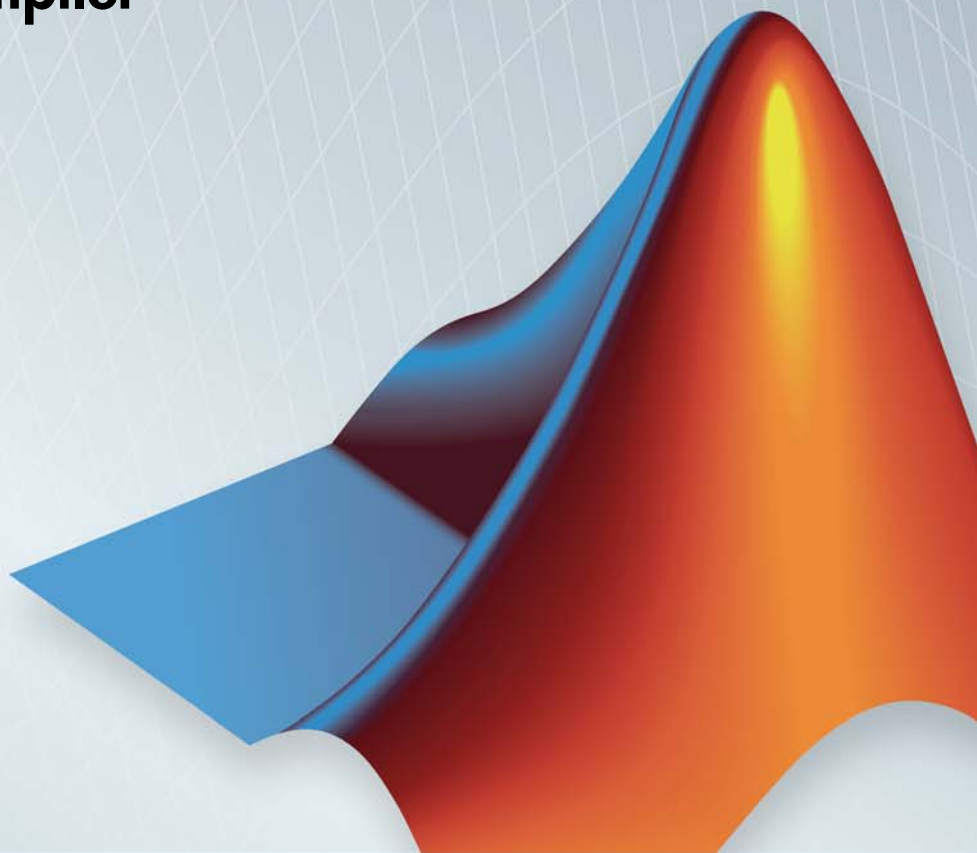
MATLAB®

MathWorks®

How to Contact MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB® Compiler™ User's Guide

© COPYRIGHT 1995–2013 by The MathWorks, Inc.

Trademarks

Patents

Contents

Getting Started

1

Installation and Configuration

2

MATLAB Code Deployment

3

C and C++ Standalone Executable and Shared Library Creation

4

Deployment Process

5

Compiler Commands

6

Standalone Applications

7

Libraries

8

Troubleshooting

9

Limitations and Restrictions

10

11

12

A

B

Using MATLAB Compiler on Mac or Linux

Index

Getting Started

Product Description

Build standalone executables and software components from MATLAB® code

MATLAB Compiler™ lets you share your MATLAB application as an executable or a shared library. Executables and libraries created with MATLAB Compiler use a runtime engine called the MATLAB Compiler Runtime (MCR). The MCR is provided with MATLAB Compiler for distribution with your application and can be deployed royalty-free.

Key Features

- Packages MATLAB applications as executables and shared libraries
- Lets you distribute standalone executables and software components royalty-free
- Lets you incorporate MATLAB based algorithms into applications developed using other languages and technologies
- Encrypts MATLAB code so that it cannot be viewed or modified

Product Overview

In this section...
"What is MATLAB® Compiler™?" on page 1-3
"How Do I Use This Product?" on page 1-4
"How Does This Product Work?" on page 1-5
"Limitations and Restrictions" on page 1-6

What is MATLAB Compiler?

MATLAB Compiler compiles a MATLAB application into a standalone application or shared library. The act of compiling MATLAB code into various targets (such as standalone executables or shared libraries) is also referred to as *building*.

Building with MATLAB Compiler enables you to run your MATLAB application outside the MATLAB environment. It reduces application development time by eliminating the need to translate code into different languages.

If you are building a standalone application, MATLAB Compiler produces an executable. If you integrate into C or C++, MATLAB Compiler provides an interface to a shared library. If you integrate into other development languages, MATLAB builder products (available separately) let you package MATLAB applications as software components. For example, you can build Java classes, .NET components, or Microsoft® Excel® add-ins.

Note If you are using a non-Windows operating system, "console applications" are referred to as "standalone applications".

When To Use MATLAB Compiler

Use MATLAB Compiler to:

• Deploy C or C++ code that interfaces with MATLAB

- Package MATLAB® applications as executables and shared libraries

- Distribute standalone applications and software components, royalty-free

- Incorporate MATLAB-based algorithms into applications developed using other languages and technologies

- Encrypt your MATLAB code, so that it cannot be viewed or modified

When Not To Use MATLAB Compiler

Do not use MATLAB Compiler and builder products for applications shown in the following table. Instead, use the MathWorks® product indicated.

To...	Use...
• Generate readable, efficient, and embeddable C code from MATLAB code • Generate MEX functions from MATLAB code for rapid prototyping and verification of generated C code within MATLAB • Integrate MATLAB code into Simulink® • Speed up fixed-point MATLAB code • Generate hardware description language (HDL) from MATLAB code	*MATLAB Coder™*
• Integrate custom C code into MATLAB with MEX files • Call MATLAB from C and Fortran programs	*MATLAB External Interfaces*

How Do I Use This Product?

Use MATLAB Compiler by running the Deployment Tool (`deploytool`) from MATLAB or by executing the `mcc` command.

How Does This Product Work?

MATLAB Compiler readies your application for deployment to enterprise computing environments using a combination of generated archives, libraries, and wrapper files.

Standalone Applications and Shared Libraries

An application or library generated by MATLAB Compiler consists of a platform-specific binary file and an archive file containing the encrypted MATLAB application and data.

A standalone binary file (also called a *standalone executable*) consists of one main function. A shared library binary consists of multiple functions for exporting.

Wrapper Files

MATLAB Compiler generates *wrapper* files. These files provide an interface to your MATLAB code when compiled. The wrapper files and MATLAB code are compiled into platform-specific binary files. Wrapper files differ depending on the execution environment.

MATLAB Compiler-generated Applications and the MATLAB Compiler Runtime (MCR)

The MATLAB Compiler Runtime (MCR) is a standalone set of shared libraries that enables the execution of compiled MATLAB applications or components on computers that do not have MATLAB installed.

When you package and distribute applications and libraries that MATLAB Compiler generates, you include the MCR so your MATLAB code can be run on systems without a licensed version of MATLAB. You set the system paths on the user system so your application finds the MCR and supporting files.

You have the option to include the MCR with every package generated by the Deployment Tool (`deploytool`). Include the MCR by clicking **Add MCR** on the **Package** tab or download it from the Web. Install it on target machines by running the self-extracting package executable. For more information on

the MCR and the MCR Installer, see "Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)" on page 1-23

See the MATLAB Compiler Runtime (MCR) product page for more information about downloading the MCR.

Limitations and Restrictions

Architecture Compatibility

MATLAB Compiler 32-Bit Applications are compatible with Windows 64. Applications created using a 64–bit version of MATLAB Compiler are generally not compatible with 32–bit applications.

For a specific list of requirements, see `http://www.mathworks.com/support/compilers/current_release/`.

Limitations on Deployability

MATLAB code can only be deployed if the toolbox with which it was created is compatible with MATLAB Compiler. MATLAB code generated by certain toolboxes cannot be deployed, usually due to legal, proprietary restrictions.

To determine if your toolbox is compatible, check your toolbox documentation as well as the MATLAB Compiler list of "MATLAB Functions That Cannot Be Compiled" on page 10-10

For a complete list of what makes MATLAB code deployable or non-deployable, see "MATLAB® Compiler™ Limitations" on page 10-2 and "Write Deployable MATLAB Code" on page 3-12 in this User's Guide.

MATLAB Compiler Prerequisites

In this section...
"Your Role in the Application Deployment Process" on page 1-7
"What You Need to Know" on page 1-8
"Products, Compilers, and IDE Installation" on page 1-9
"For More Information" on page 1-10

Your Role in the Application Deployment Process

Depending on the size of your organization, you play one role, or many, in the process of successfully deploying a standalone executable or shared library.

The table Application Deployment Roles, Tasks, and References on page 1-7 describes some of the different MATLAB Compiler roles or jobs. It also describes which tasks you would most likely perform when running in this chapter.

Application Deployment Roles, Tasks, and References

Role	Knowledge Base	Responsibilities	Task To Achieve Goal
MATLAB programmer	• MATLAB expert • No IT experience needed • No access to IT systems	• Develops models; implements in MATLAB • Creates a standalone executable that can be used directly by the end user, or a shared library that can be integrated into a C or	"Create a Standalone Application From MATLAB Code" on page 1-11 "Create a Shared Library from MATLAB Code" on page 1-16

Application Deployment Roles, Tasks, and References (Continued)

Role	Knowledge Base	Responsibilities	Task To Achieve Goal
		C++ application by the C/C++ developer	
C/C++ developer	• Little to no MATLAB experience • Moderate IT experience • C/C++ expert • Minimal access to IT systems	• Integrates shared libraries with C or C++ application • Call the integrated shared library from a C or C++ application	"Create a Shared Library from MATLAB Code" on page 1-16 "Integrate a Shared Library in an Application" on page 1-19
End user	• No MATLAB experience • Some knowledge of the data that is being displayed, but not how it was created	• Receives deployed application from MATLAB programmer or C/C++ developer • Downloads MATLAB Compiler Runtime (MCR) to the run application	"Share MATLAB Code Using the MATLAB Compiler Runtime (MCR)" on page 1-23

What You Need to Know

To use the MATLAB Compiler product, specific requirements exist for each user role.

Role	Requirements
![icon] **MATLAB programmer**	• A basic knowledge of MATLAB, and how to work with: ▪ MATLAB data types ▪ MATLAB structures
![icon] **C/C++ developer**	• Exposure to: ▪ Exposure to the C or C++ programming languages ▪ Procedural or object-oriented programming concepts

Products, Compilers, and IDE Installation

Install the following products to run the example described in this chapter:

- MATLAB
- MATLAB Compiler
- A supported C or C++ compiler

For more information about product installation and requirements, see "Installing MATLAB® Compiler™" on page 2-5.

Compiler Selection with mbuild -setup

The first time you use MATLAB Compiler, after starting MATLAB, run this command:

```
mbuild -setup
```

For more information about mbuild -setup, see "What Is mbuild?" on page 2-7.

For More Information

If you want to...	See...
Create a standalone application in C or C++	• "Create a Standalone Application From MATLAB Code" on page 1-11 • "When to Create a Standalone Application" on page 4-2
Create a shared library in C or C++	• "Create a Shared Library from MATLAB Code" on page 1-16 • "Integrate a Shared Library in an Application" on page 1-19 • "Integrate C Shared Libraries" on page 8-4 • "Integrate C++ Shared Libraries" on page 8-18
Learn more about standalone applications and shared libraries	"Supported Compilation Targets" on page 4-2
Verify your MATLAB code is deployable	"Write Deployable MATLAB Code" on page 3-12
Distribute your standalone or shared library to end user with the MATLAB Compiler Runtime (MCR)	"Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)" on page 1-23

Create a Standalone Application From MATLAB Code

The MATLAB programmer performs the following tasks.

MATLAB Programmer

Role	Knowledge Base	Responsibilities
MATLAB programmer	• MATLAB expert • No IT experience needed • No access to IT systems	• Develops mathematical functions; implements functions in MATLAB • Uses tools to create a component that is used by the C or C++ developer

Create a Standalone Executable or Console Application

This example shows how to create a standalone executable or console application from MATLAB code that generates a magic square.

Standalone applications can be implemented as either an executable (an EXE file) or as a console application, running in a DOS command window. For more information on each of these target types, see "When to Create a Standalone Application" on page 4-2 and "When to Create a Shared Library" on page 4-3.

Using MATLAB, locate and open magicsquare.m (see). magicsquare.m can be found in the *matlabroot*\extern\examples\compiler folder. Add this folder to the MATLAB path (using addpath) so MATLAB can find it.

```
function m = magicsquare(n)
%MAGICSQUARE generates a magic square matrix of the size
% specified by the input parameter n.

% Copyright 2013 The MathWorks, Inc.

if ischar(n)
    n=str2num(n);
end
m = magic(n);
```

At the MATLAB command prompt, enter `magicsquare(5)`, and view the results. The output appears as follows:

```
m =

17 24  1  8 15
23  5  7 14 16
 4  6 13 20 22
10 12 19 21  3
11 18 25  2  9

ans =

17 24  1  8 15
23  5  7 14 16
 4  6 13 20 22
10 12 19 21  3
11 18 25  2  9
```

Type `deploytool` at the command prompt, and press **Enter**. The Deployment Project dialog box opens.
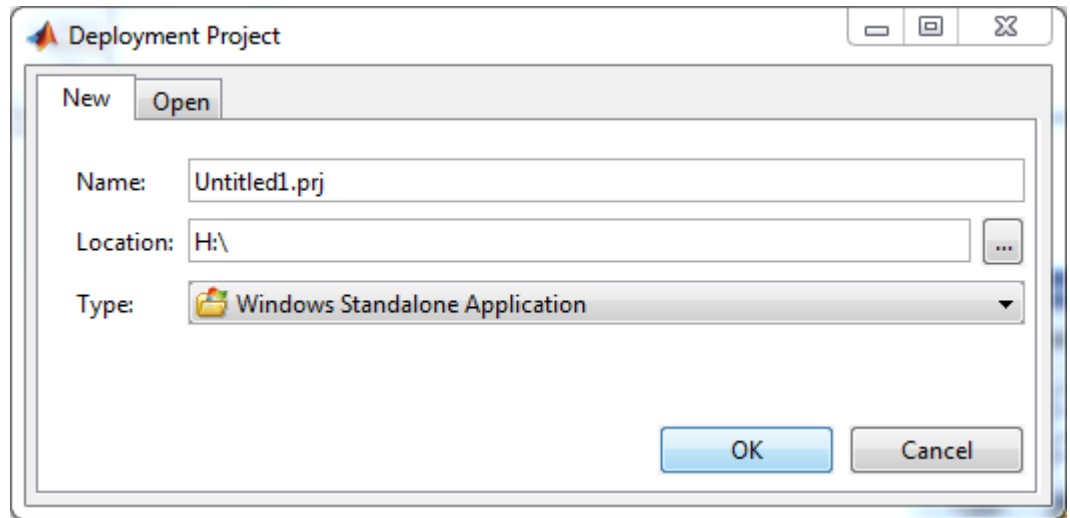
The Deployment Project Dialog Box

Create a deployment project using the Deployment Project dialog box:

- Enter `MagicExample.prj` in the **Name** field.

- Enter the location of your project in the **Location** field. Alternately, navigate to the location of the project and click **Select Folder**.

- Select **Windows Standalone Application** or **Console Application** from the **Type** drop-down menu.

Tip You can inspect the values in the Settings dialog before building your

project. To do so, click the Action icon (⚙▾) on the toolbar, and then click **Settings**. Verify where your `distrib` folder will be created. You will need to reference this folder later in this procedure.

Click **OK** when finished.

On the **Build** tab, click **Add main file** to open the Add Files dialog box.

Locate magicsquare.m in the *matlabroot*\extern\examples\compiler folder and click **Open**.

Note In this context, **main file** refers to the primary MATLAB file you want to deploy. It does not refer to the main module in a C or C++ program. You can add only one **main file**.

After you have added magicsquare.m to the **Build** tab, click the Build icon (⊞). Building may take several minutes. When the build finishes, click **Close** to dismiss the dialog box.

After you have built your standalone application with the Deployment Tool, you have the following in the distrib subfolder of your project folder:

- MagicExample.exe — the executable application.
- readme.txt — important information about how to use this component.

Run your standalone application by running the executable file.

- Locate the application (.exe file), in your distrib folder, where it was built by MATLAB Compiler.
- Run the application file. For example, to run the file from within MATLAB, change your working folder to distrib and enter the following at the MATLAB command line:

  ```
  !MagicExample 5
  ```

  A 5-by-5 magic square is displayed in MATLAB:

  ```
  17 24  1  8 15
  23  5  7 14 16
   4  6 13 20 22
  10 12 19 21  3
  11 18 25  2  9
  ```

Alternately, run your standalone application as a console application.

- Locate the application (.exe file), in your distrib folder, where it was built by MATLAB Compiler.

- Run the application file from a console.

  For example, open a Windows® or Linux® command window and change your working folder to distrib.

- Enter the following:

  ```
  MagicExample 5
  ```

  A 5-by-5 magic square is displayed in the console:

  ```
  17 24  1  8 15
  23  5  7 14 16
   4  6 13 20 22
  10 12 19 21  3
  11 18 25  2  9
  ```

What Is a Magic Square?

A *magic square* is simply a square array of integers arranged so that their sum is the same when added vertically, horizontally, or diagonally.

How Do I Access the Examples?

The examples for MATLAB Compiler are in *matlabroot*\extern\examples\compiler. For *matlabroot*, substitute the MATLAB root folder on your system. Type matlabroot to see this folder name.

Note If you are using a non-Windows operating system, "console applications" are referred to as "standalone applications".

Create a Shared Library from MATLAB Code

In this section...
"Create a Shared Library from MATLAB Code" on page 1-16
"Integrate a Shared Library in an Application" on page 1-19

The MATLAB programmer performs the following tasks.

MATLAB Programmer

Role	Knowledge Base	Responsibilities
MATLAB programmer	• MATLAB expert • No IT experience needed • No access to IT systems	• Develops models; implements in MATLAB • Uses tools to create a component that is used by the C or C++ developer

Create a Shared Library from MATLAB Code

This example shows how to create a shared library from MATLAB code. You create shared libraries for integration into either C or C++ applications.

In this example, three MATLAB functions that perform arithmetic calculations are compiled into one shared library with MATLAB Compiler. The functions are then integrated into a simple driver application written in C++.

Using MATLAB, locate and open addmatrix.m, multiplymatrix.m, and eigmatrix.m (see). Add these files to the MATLAB path so MATLAB can find them.

Running the functions in MATLAB, with 1 assigned to variable a1 and 2 assigned to variable a2, produces the output shown below each function source code listing.

```
function a = addmatrix(a1, a2)
%ADDMATRIX Add two matrices
```

```
%    This function adds the two matrices passed as input.

% Copyright 2013 MathWorks, Inc.

a = a1 + a2;

ans =

     3

function m = multiplymatrix(a1, a2)
%MULTIPLYMATRIX Multiplies two matrices
%    This function mulitplies the two matrices passed as input.

% Copyright 2013 MathWorks, Inc.

m =  a1*a2;

ans =

     2

function e = eigmatrix(a1)
%EIGMATRIX Returns the eigen value of the given matrix

% Copyright 2013 MathWorks, Inc.

 try
  %Tries to calculate the eigen value and return it.
  e = eig(a1);
 catch
  %Returns a -1 on error.
  e = -1;
end

ans =

     1
```

Type `deploytool` at the command prompt, and press **Enter**. The Deployment
Project dialog box opens.

Create a deployment project using the Deployment Project dialog box:

- Enter libmatrixp.prj in the **Name** field.
- Enter the location of your project in the **Location** field. Alternately, navigate to the location of the project and click **Select Folder**.
- Select **C++ Shared Library** from the **Type** drop-down menu.

Tip You can inspect the values in the Settings dialog before building your

project. To do so, click the Action icon (⚙▾) on the toolbar, and then click **Settings**. Verify where your distrib folder will be created. You will need to reference this folders later in this procedure.

Click **OK** when finished.

On the **Build** tab, click **Add files** to open the Add Files dialog box..

Locate addmatrix.m, multiplymatrix.m, and eigmatrix.m in the *matlabroot*\extern\examples\compiler folder and click **Open** to add each file to the project.

Tip Alternately, open multiple files at one time by holding down the **CTRL** key and selecting all three files. Click **Open** to add them to the project.

After you have added the MATLAB functions to the **Build** tab, click the Build icon (📅). Building may take several minutes. When the build finishes, click **Close** to dismiss the dialog box.

After you have built your C shared library with the Deployment Tool, you have the following in the distrib subfolder of your project folder:

- libmatrixp.dll — the dynamic load library.
- libmatrixp.lib — the object code library.
- libmatrixp.h — the C++ header file.

- readme.txt — important information about how to use this component.

Integrate a Shared Library in an Application

This example shows how to call a C++ shared library built with MATLAB Compiler from a C++ application. Integrating a C library requires similar steps. For examples of integrating C shared libraries, see "Integrate C Shared Libraries" on page 8-4.

You integrate the library by compiling and linking using mbuild. mbuild generates an executable file (EXE) from your C or C++ code that calls the shared library built with the Deployment Tool.

Copy the example file matrixdriver.cpp to the distrib subfolder of your project folder. This folder should also contain the built object library (libmatrixp.lib) and the DLL (libmatrixp.dll). See for the location of matrixdriver.cpp.

Open matrixdriver.cpp in a text editor. In the C++ code, note the following:

- The main module controls program execution. main calls mclInitialize and returns the status of the MATLAB functions being called in the program, using mclRunMain.

  ```
  int main()
  {
      mclmcrInitialize();
      return mclRunMain((mclMainFcnType)run_main,O,NULL);
  }
  ```

- You must initialize both the application and the shared library with respective calls to the mclInitializeApplication and libmatrixpInitialize methods in the run_main module.

  ```
  if (!mclInitializeApplication(NULL,O))
      {
          std::cerr << "could not initialize the application properly"
                      << std::endl;
       return -1;
      }
      if( !libmatrixpInitialize() )
  ```

```
        {
             std::cerr << "could not initialize the library properly"
                         << std::endl;
      return -1;
```

- You create input data, the output array, call your MATLAB functions, and display return values in the body of the program (in the try portion of the try/catch block).

```
 // Create input data
              double data[] = {1,2,3,4,5,6,7,8,9};
              mwArray in1(3, 3, mxDOUBLE_CLASS, mxREAL);
              mwArray in2(3, 3, mxDOUBLE_CLASS, mxREAL);
              in1.SetData(data, 9);
              in2.SetData(data, 9);

              // Create output array
              mwArray out;

              // Call the library function
              addmatrix(1, out, in1, in2);

              // Display the return value of the library function
              std::cout << "The value of added matrix is:" << std::endl;
              std::cout << out << std::endl;

              multiplymatrix(1, out, in1, in2);
              std::cout << "The value of the multiplied matrix is:"
                        << std::endl;
              std::cout << out << std::endl;

              eigmatrix(1, out, in1);
              std::cout << "The eigenvalues of the first matrix are:"
                        << std::endl;
              std::cout << out << std::endl;
          }
```

- You must throw exceptions, if needed, using instances of mwException (in the catch portion of the try/catch block).

```
catch (const mwException& e)
```

```
                    {
                      std::cerr << e.what() << std::endl;
                      return -2;
                    }
                    catch (...)
                    {
                      std::cerr << "Unexpected error thrown" << std::endl;
                      return -3;
                    }
```

- You must terminate both the application and the shared library
  with respective calls to the mclTerminateApplication and
  libmatrixpTerminate methods.

```
   // Call the application and library termination routine
          libmatrixpTerminate();
      }
  /* Note that you should call mclTerminate application at the end of
   * your application.
   */
      mclTerminateApplication();
      return 0;
  }
```

In MATLAB, use the command line to navigate to the distrib folder
(libmatrixp) where you built the C++ shared library. Add this folder to the
MATLAB path so MATLAB can find it. Use mbuild to compile the C++
program (using the default compiler you specified with mbuild in "Compiler
Selection with mbuild -setup" on page 1-9) and link it to the object library
built using the Deployment Tool.

```
mbuild matrixdriver.cpp libmatrixp.lib
```

If you are running on a Linux or Mac platform, set environment variables
using then shell script (libmatrixp.sh) found in your distrib folder. For
more information, see "Set MCR Paths on Mac or Linux with Scripts" on
page B-12.

Run the driver application from the distrib folder, at a DOS prompt, by
entering the following.

```
matrixdriver.exe
```

The following output is generated:

```
The value of added matrix is:
     2      8     14
     4     10     16
     6     12     18
The value of the multiplied matrix is:
    30     66    102
    36     81    126
    42     96    150
The eigenvalues of the first matrix are:
   16.1168
   -1.1168
   -0.0000
```

Share MATLAB Code Using the MATLAB Compiler Runtime (MCR)

The end user performs these additional tasks to prepare integrating the shared library in an enterprise environment.

End User

Role	Knowledge Base	Responsibilities
End user	• No MATLAB experience • Some knowledge of the data that is being displayed, but not how it was created	• Receives deployed application from MATLAB programmer or C/C++ developer • Downloads MATLAB Compiler Runtime (MCR) to run the application on a system without an installed copy opf MATLAB

Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)

On target computers without MATLAB, install the MCR, if it is not already present on the deployment machine.

Install MATLAB Compiler Runtime (MCR)

The *MATLAB Compiler Runtime (MCR)* is an execution engine made up of the same shared libraries MATLAB uses to enable execution of MATLAB files on systems without an installed version of MATLAB.

The MATLAB Compiler Runtime (MCR) is now available for downloading from the Web to simplify the distribution of your applications or components created with the MATLAB Compiler. Download the MCR from the MATLAB Compiler product page.

The MCR installer does the following:

1 Installs the MCR (if not already installed on the target machine)

   2 Installs the component assembly in the folder from which the installer is
   run

   3 Copies the MWArray assembly to the Global Assembly Cache (GAC), as
   part of installing the MCR

MCR Prerequisites

   1 Since installing the MCR requires write access to the system registry,
   ensure you have administrator privileges to run the MCR Installer.

   2 The version of the MCR that runs your application on the target computer
   must be compatible with the version of MATLAB Compiler that built the
   component.

   3 Avoid installing the MCR in MATLAB installation directories.

Add the MCR Installer To Your Deployment Package (Optional)

The example shows how to optionally include the MCR in your deployment
package, using the Deployment Tool. A *package* is a zipped (compressed) file
containing all files needed to run the standalone application or shared library
built with MATLAB Compiler.

On the **Package** tab of the deploytool interface, click **Add MCR**.

Select one of the following options:

- **Embed the MCR in the package** — This option adds the MCR to
  the Deployment package, which contains all the files needed to run the
  standalone application or shared library built with MATLAB Compiler.
  Choosing this option adds MCRInstaller.exe to your package, which can
  increase the size of the package substantially.

- **Add a batch file to invoke the MCR over the network** — The
  option adds a batch file that, when executed, invokes the MCR from a
  single location on your network. You must customize this file to point
  to the location of the MCR on your network. Choosing this option adds
  MCRInstaller.cmd to your package.

Click the Package button ( ).

In the Package dialog box, navigate to the location where you want to save the package file. In the **File name** field, enter a name for the package file (for example, `libmatrixp.exe`).

Click **Save**. The Package dialog appears and your package is created in the location specified. When packaging is complete, click **Close** to dismiss the dialog box.

Distribute the package file as needed to additional end users.

Installing the MCR

This example shows how to install the MATLAB Compiler Runtime (MCR) on a system.

Locate the MCR Installer that was either packaged with the Deployment Tool or downloaded from the Web.

Run the installer as you would any other executable on your system, following the Wizard instructions.

Note If you are running on a platform other than Windows, set the system paths on the target machine. Setting the paths enables your application to find the MCR.

Windows paths are set automatically. On Linux and Mac, you can use the run script to set paths. See Appendix B, "Using MATLAB® Compiler™ on Mac or Linux" for detailed information on performing all deployment tasks specifically with UNIX variants such as Linux and Mac.

For More Information

About This	Look Here
Detailed information on standalone applications	"Deploying Standalone Applications" on page 7-3
Creating libraries	"Integrate C Shared Libraries" on page 8-4 "Integrate C++ Shared Libraries" on page 8-18
Using the mcc command	"mcc Command Line Arguments Grouped by Task" on page A-8
Troubleshooting	"Common Issues" on page 9-4 "Failure Points and Possible Solutions" on page 9-5 "Troubleshooting mbuild" on page 9-15 "MATLAB® Compiler™" on page 9-17 "Deployed Applications" on page 9-21

2

Installation and Configuration

- "Before You Install MATLAB® Compiler™" on page 2-2
- "Installing MATLAB® Compiler™" on page 2-5
- "Configuring the MCR Installer For Invocation From a Network Location" on page 2-6
- "Configuring Your Options File with mbuild" on page 2-7
- "Solving Installation Problems" on page 2-13

Before You Install MATLAB Compiler

In this section...
"Install MATLAB" on page 2-2
"Install an ANSI C or C++ Compiler" on page 2-2

Systems Administrator

Role	Knowledge Base	Responsibilities
Systems administrator	• No MATLAB experience • Access to IT Systems • IT expert	• Gets updates to a deployed component or the larger application out to end users • Manages versions of the application for end users • Manages versions of the MCR

Install MATLAB

To install MATLAB, refer to the *MATLAB Installation Guide*.

See MATLAB Compiler Platform & Requirements for details. The memory and disk requirements to run MATLAB Compiler software are the same as the requirements for MATLAB.

Install an ANSI C or C++ Compiler

Install supported ANSI® C or C++ compiler on your system. Certain output targets require particular compilers.

To install your ANSI C or C++ compiler, follow vendor instructions that accompany your C or C++ compiler.

Note If you encounter problems relating to the installation or use of your ANSI C or C++ compiler, consult your C or C++ compiler vendor.

Supported ANSI C and C++ Windows Compilers

Use one of the following 32-bit C/C++ compilers that create 32-bit Windows dynamically linked libraries (DLLs) or Windows applications:

- Microsoft Visual C++® (MSVC).
  - The only compiler that supports the building of COM objects and Excel plug-ins is Microsoft Visual C++.
  - The only compiler that supports the building of .NET objects is Microsoft Visual C# Compiler for the Microsoft .NET Framework.

  See the *MATLAB Builder™ NE Release Notes* for a list of supported .NET Framework versions.

Note For an up-to-date list of all the compilers supported by MATLAB and MATLAB Compiler, see the MathWorks Technical Support notes at:

`http://www.mathworks.com/support/compilers/current_release/`

Supported ANSI C and C++ UNIX Compilers

MATLAB Compiler software supports the native system compilers on:

- Linux
- Linux x86-64
- Mac OS X

MATLAB Compiler software supports `gcc` and `g++`.

Common Installation Issues and Parameters

When you install your C or C++ compiler, you sometimes encounter requests for additional parameters. The following tables provide information about common issues occurring on Windows and UNIX® systems where you sometimes need additional input or consideration.

Windows Operating System

Issue	Comment
Installation options	(Recommended) Full installation.
Installing debugger files	For the purposes of MATLAB Compiler, it is not necessary to install debugger (DBG) files.
Microsoft Foundation Classes (MFC)	Not needed.
16-bit DLLs	Not needed.
ActiveX®	Not needed.
Running from the command line	Make sure that you select all relevant options for running your compiler from the command line.
Updating the registry	If your installer gives you the option of updating the registry, perform this update.
Installing Microsoft Visual C++ Version 6.0	To change the install location of the compiler, change the location of the Common folder. Do not change the location of the VC98 folder from its default setting.

UNIX Operating System

Issue	Comment
Determine which C or C++ compiler is available on your system.	See your system administrator.
Determine the path to your C or C++ compiler.	See your system administrator.
Installing on Maci64	Install X Code from installation DVD.

Installing MATLAB Compiler

In this section...

Installing Your Product

To install MATLAB Compiler software on Windows, follow the instructions in the *MATLAB Installation Guide*. If you have a license to install MATLAB Compiler, select the product as you proceed through the installation process.

Compiler Options

MATLAB Compiler software requires a supported ANSI C or C++ compiler on your system. Refer to the "Install an ANSI C or C++ Compiler" on page 2-2 for more information.

Configuring the MCR Installer For Invocation From a Network Location

Download the MCR from the Web at
`http://www.mathworks.com/products/compiler/mcr`

When you deploy an application, you have an option to either embed the MCR in the package or invoke the MCR installer from a network location.

As Systems Administrator, you should:

- Decide which option makes the most sense to implement for your installation. See for more information and criteria for making your selection.

- Communicate what option you select to your end users to prevent them from creating multiple copies of the MCR Installer on various network shares, providing you choose the network invocation option.

Configuring Your Options File with mbuild

In this section...
"What Is mbuild?" on page 2-7
"When Not to Use mbuild -setup" on page 2-7
"Running mbuild" on page 2-8
"Locating and Customizing the Options File" on page 2-10

What Is mbuild?

Running the mbuild configuration script creates an option file that:

- Sets the default compiler and linker settings for each supported compiler.
- Allows you to changes compilers or compiler settings.
- Builds (compiles) your application.

Note The following mbuild examples apply only to the 32-bit version of MATLAB.

About mbuild and Linking

Static linking is not an option for applications generated by MATLAB Compiler. Compiled applications all must link against MCLMCRRT. This shared library explicitly dynamically loads other shared libraries. You cannot change this behavior on any platform.

When Not to Use mbuild -setup

Run mbuild -setup before using any deployment product unless you are doing one of the following:

- Using MATLAB Builder JA
- Using MATLAB Builder NE

• Creating a standalone or Windows standalone target with MATLAB Compiler

Running mbuild

1 Run `mbuild -setup` from MATLAB.

2 Select a compiler. Additional platform-specific information follows.

> **Note** The compiler-specific options file specifies that your compiler contains flags and settings to control the operation of the installed C and C++ compiler. For information on modifying the options file to customize your compiler settings, see "Locating and Customizing the Options File" on page 2-10.

Windows

Executing `mbuild -setup` on Windows displays a message of this type:

```
Welcome to mbuild -setup.  This utility will help you set up
a default compiler.  For a list of supported compilers, see
http://www.mathworks.com/support/compilers/R2012b/win64.html

Please choose your compiler for building shared libraries
or COM components:

Would you like mbuild to locate installed compilers [y]/n? y

Select a compiler:
[1] Microsoft Visual C++ 2010 in C:\Program Files (x86)\
                          Microsoft Visual Studio 10.0
[2] Microsoft Visual C++ 2008 SP1 in c:\Program Files (x86)\
                          Microsoft Visual Studio 9.0

[0] None
```

The preconfigured options files included with MATLAB for Windows appear in the following table.

Note These options apply only to the 32-bit version of MATLAB.

Options File	Compiler
`msvc80compp.bat`	Microsoft Visual C/C++, Version 8.0 Microsoft Visual C/C++, Version 8.0 Express Edition
`msvc90compp.bat`	Microsoft Visual C/C++, Version 9.0 Microsoft Visual C/C++, Version 9.0 Express Edition
`msvc100compp.bat`	Microsoft Visual C/C++, Version 10.0 Microsoft Visual C/C++, Version 10.0 Express Edition

UNIX

Executing the command on UNIX displays a message of this type:

```
mbuild -setup

Using the 'mbuild -setup' command selects an options file that
is placed in ~/.matlab/current_release and
 used by default for 'mbuild'. An options file in the current
working directory or specified on
the command line overrides the default options file
in ~/.matlab/current_release.

Options files control which compiler to use, the compiler
and link command options, and the run time libraries to link
against.

To override the default options file, use the 'mbuild -f'
command (see 'mbuild -help' for more information).

The options files available for mbuild are:
```

```
1: matlabroot/bin/mbuildopts.sh :
Build and link with MATLAB C-API or MATLAB Compiler-generated
library via the system ANSI C/C++ compiler

matlabroot/bin/mbuildopts.sh is being copied to
/home/user/.matlab/current_release/mbuildopts.sh
```

The preconfigured options file for UNIX is mbuildopts.sh, which uses gcc for Linux and Macintosh.

See the reference page for more information about mbuild. For examples of mbuild usage, see "Compiling the Driver Application" on page 8-19.

Locating and Customizing the Options File

- "Locating the Options File" on page 2-10
- "Changing the Options File" on page 2-11

Locating the Options File

Windows Operating System. To locate your options file on Windows, the mbuild script searches the following locations:

- Current folder
- The user profile folder

mbuild uses the first occurrence of the options file it finds. If it finds no options file, mbuild searches your machine for a supported C compiler and uses the factory default options file for that compiler. If mbuild finds multiple compilers, it prompts you to select one.

The Windows user profile folder contains user-specific information such as desktop appearance, recently used files, and **Start** menu items. The mbuild utility stores its options files, compopts.bat, in a subfolder of your user profile folder, named Application Data\MathWorks\MATLAB\current_release.

Under Windows with user profiles enabled, your `user profile` folder is `%windir%\Profiles\username`. However, with user profiles disabled, your `user profile` folder is `%windir%`. You can determine if user profiles are enabled by using the **Passwords** control panel.

UNIX Operating System. To locate your options file on UNIX, the `mbuild` script searches the following locations:

- Current folder
- `$HOME/.matlab/`*current_release*
- *matlabroot*`/bin`

`mbuild` uses the first occurrence of the options file it finds. If `mbuild` finds no options file, an errors message appears.

Changing the Options File

Although it is common to use one options file for all of your MATLAB Compiler related work, you can change your options file at anytime. The `setup` option resets your default compiler to use the new compiler every time. To reset your C or C++ compiler for future sessions, enter:

```
mbuild -setup
```

Modifying the Options File on Windows. You can use the `-setup` option to change your options file settings on Windows. The `-setup` option copies the appropriate options file to your `user profile` folder.

To modify your options file on Windows:

1 Enter `mbuild -setup` to make a copy of the appropriate options file in your local area.

2 Edit your copy of the options file in your `user profile` folder to correspond to your specific needs, and save the modified file.

After completing this process, the `mbuild` script uses the new options file every time with your modified settings.

Modifying the Options File on UNIX. You can use the `setup` option to change your options file settings on UNIX. For example, to change the current linker settings, use the `setup` option.

The `setup` option creates a user-specific `matlab` folder in your home folder and copies the appropriate options file to the folder.

Do not confuse these user-specific `matlab` folders with the system `matlab` folder.

To modify your options file on the UNIX:

1 Use `mbuild -setup` to make a copy of the appropriate options file in your local area.

2 Edit your copy of the options file to correspond to your specific needs, and save the modified file.

Solving Installation Problems

You can contact MathWorks:

- Via the Web at `www.mathworks.com`. On the MathWorks home page, click **My Account** to access your MathWorks Account, and follow the instructions.

- Via email at `service@mathworks.com`.

MATLAB Code Deployment

MATLAB Application Deployment Products

MATLAB Programmer

Role	Knowledge Base	Responsibilities
MATLAB programmer	• MATLAB expert • No IT experience • No access to IT systems	• Develops models; implements in MATLAB • Serves as tool builder • Uses tools to create a component that is used by the C or C++ developer

The following table and figure summarizes the target applications supported by each product.

MATLAB Suite of Application Deployment Products

Product	Target	Stand-alones?	Function Libraries?	Graphical Apps?	Web Apps?	WebFigures?
MATLAB Compiler	C and C++ standalones	Yes	Yes	Yes	No	No
MATLAB Builder NE	C# .NET components Visual Basic COM components	No	Yes	Yes	Yes	Yes
MATLAB Builder JA	Java® components	No	Yes	Yes	Yes	Yes
MATLAB Builder EX	Microsoft Excel add-ins	No	Yes	Yes	No	No
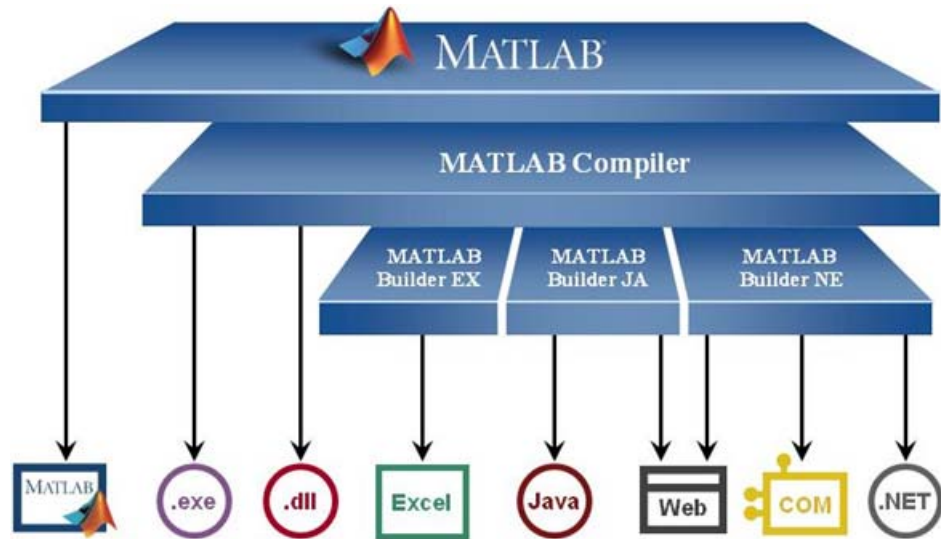
MATLAB® Application Deployment Products

As this figure illustrates, each of the builder products uses the MATLAB Compiler core code to create deployable components.

Application Deployment Products and the Deployment Tool

What Is the Difference Between the Deployment Tool and the mcc Command Line?

When you use the Deployment Tool (`deploytool`) GUI, you perform any function you would invoke using the MATLAB Compiler `mcc` command-line interface. The Deployment Tool interactive menus and dialogs build `mcc` commands that are customized to your specification. As such, your MATLAB code is processed the same way as if you were compiling it using `mcc`.

Deployment Tool advantages include:

- You perform related deployment tasks with a single intuitive GUI.

- You maintain related information in a convenient project file.

- Your project state persists between sessions.

- Your previous project loads automatically when the Deployment Tool starts.

- You load previously stored compiler projects from a prepopulated menu.

- Package applications for distribution.

How Does MATLAB Compiler Software Build My Application?

To build an application, MATLAB Compiler software performs these tasks:

1 Parses command-line arguments and classifies by type the files you provide.

2 Analyzes files for dependencies using the Dependency Analysis Function (`depfun`). Dependencies affect deployability and originate from functions called by the file. Deployability is affected by:

- File type — MATLAB, Java, MEX, and so on.
- File location — MATLAB, MATLAB toolbox, user code, and so on.
- File deployability — Whether the file is deployable outside of MATLAB

For more information about `depfun`, see "Dependency Analysis Function (depfun)" on page 3-7.

- ■ Circles: Input Files
- ■ Boxes: MathWorks-supplied components
- ■ Ellipses: Output Files
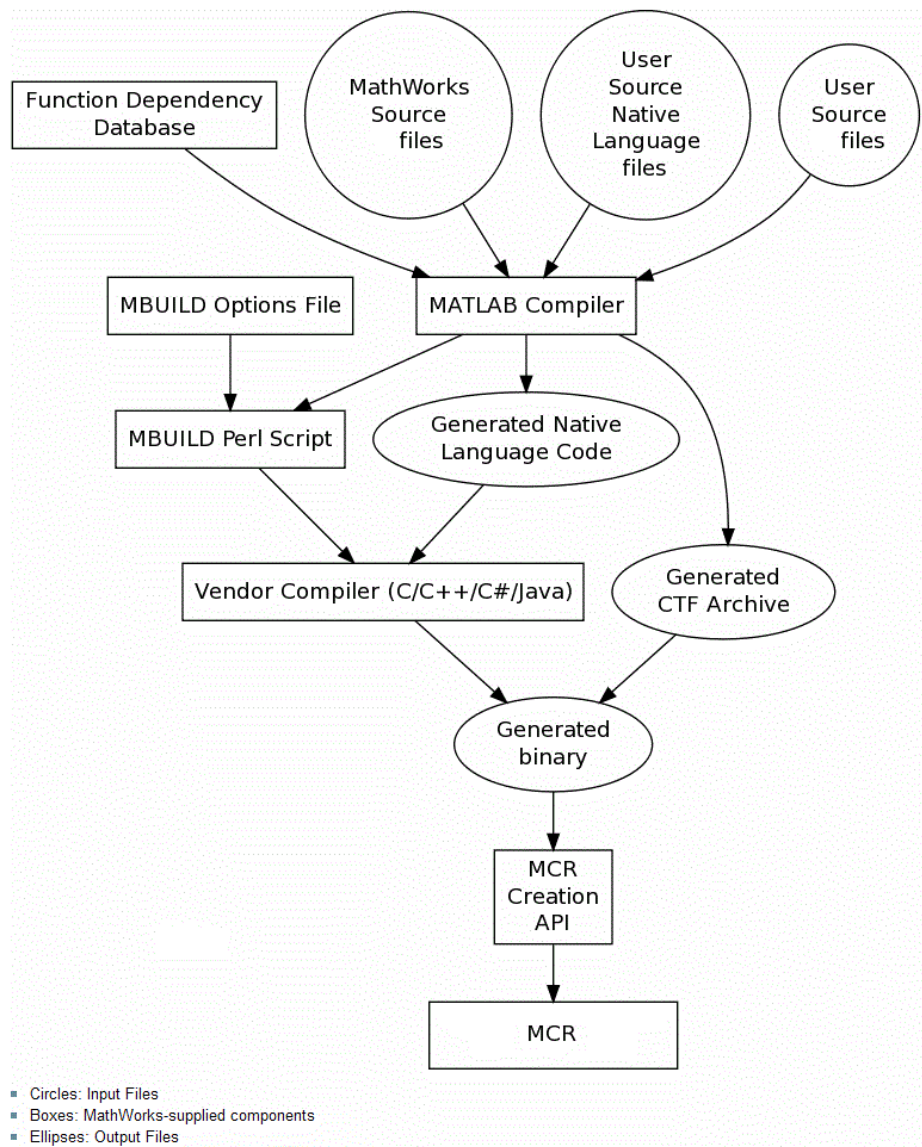
MATLAB® Compiler™ Build Process

3 Validates MEX-files. In particular, `mexFunction` entry points are verified. For more details about MEX-file processing, see "MEX-Files, DLLs, or Shared Libraries" on page 3-8.

4 Creates a CTF archive from the input files and their dependencies. For more details about CTF archives see "Component Technology File (CTF Archive)" on page 3-8.

5 Generates target-specific wrapper code. For example, a C main function requires a very different wrapper than the wrapper for a Java interface class.

6 Invokes a third-party target-specific compiler to create the appropriate binary software component (a standalone executable, a Java JAR file, and so on).

Dependency Analysis Function (depfun)

MATLAB Compiler uses a dependency analysis function (`depfun`) to determine the list of necessary files to include in the CTF package. Sometimes, this process generates a large list of files, particularly when MATLAB object classes exist in the compilation and `depfun` cannot resolve overloaded methods at compile time. Dependency analysis also processes `include/exclude` files on each pass (see the `mcc` flag "-a Add to Archive" on page 12-21).

Tip To improve compile time performance and lessen application size, prune the path with "-N Clear Path" on page 12-37, "-p Add Directory to Path" on page 12-39. You can also specify **Toolboxes on Path** in the `deploytool` **Settings**

For more information about `depfun`, `addpath`, and `rmpath`, see "Dependency Analysis Function (depfun) and User Interaction with the Compilation Path" on page 4-14.

`depfun` searches for executable content such as:

- MATLAB files

- P-files

- Java classes and `.jar` files
- `.fig` files
- MEX-files

`depfun` does not search for data files of any kind. You must manually `include` data files in the search

MEX-Files, DLLs, or Shared Libraries

When you compile MATLAB functions containing MEX-files, ensure that `depfun` can find them. Doing so allows you to avoid many common compilation problems. In particular, note that:

- Because `depfun` cannot examine MEX-files, DLLs, or shared libraries to determine their dependencies, explicitly include all executable files these files require. To do so, use either the `mcc -a` option or the options on the **Advanced** tab in the Deployment Tool under **Settings**.

- If you have any doubts that `depfun` can find a MATLAB function called by a MEX-file, DLL, or shared library, then manually include that function. To do so, use either the `mcc -a` option or by using the options on the **Advanced** tab in the Deployment Tool under **Settings**.

- Not all functions are compatible with MATLAB Compiler. Check the file `mccExcludedFiles.log` after your build completes. This file lists all functions called from your application that you cannot deploy.

Component Technology File (CTF Archive)

Each application or shared library you produce using MATLAB Compiler has an associated Component Technology File (CTF) archive. The archive contains all the MATLAB based content (MATLAB files, MEX-files, and so on) associated with the component.

MATLAB Compiler also embeds a CTF archive in each generated binary. The CTF houses all deployable files. All MATLAB files encrypt in the CTF archive using the Advanced Encryption Standard (AES) cryptosystem.

If you choose to extract the CTF archive as a separate file, the files remain encrypted. For more information on how to extract the CTF archive refer to the references in the following table.

Information on CTF Archive Embedding/Extraction and Component Cache

Product	Refer to
MATLAB Compiler	"MCR Component Cache and CTF Archive Embedding" on page 6-14
MATLAB Builder NE	"MCR Component Cache and CTF Archive Embedding"
MATLAB Builder JA	"CTF Archive Embedding and Extraction"
MATLAB Builder EX	Using MCR Component Cache and CTF Archive Embedding

Generated Component (EXE, DLL, SO, etc)

Target-
Specific
Binary
Code

CTF Archive

Encrypted
MATLAB Data

Data File

MEX-File
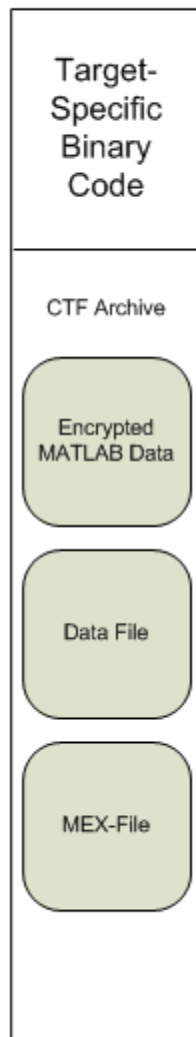
Additional Details

Multiple CTF archives, such as those generated with COM, .NET, or Excel components, can coexist in the same user application. You cannot, however, mix and match the MATLAB files they contain. You cannot combine encrypted and compressed MATLAB files from multiple CTF archives into another CTF archive and distribute them.

All the MATLAB files from a given CTF archive associate with a unique cryptographic key. MATLAB files with different keys, placed in the same CTF archive, do not execute. If you want to generate another application with a different mix of MATLAB files, recompile these MATLAB files into a new CTF archive.

MATLAB Compiler deletes the CTF archive and generated binary following a failed compilation, but only if these files did not exist before compilation initiates. Run `help mcc -K` for more information.

Note CTF archives are extracted by default to *user_name*\AppData\Local\Temp*userid*\mcrCache*n.nn*.

Caution **Release Engineers and Software Configuration Managers**: Do not use build procedures or processes that strip shared libraries on CTF archives. If you do, you can possibly strip the CTF archive from the binary, resulting in run-time errors for the driver application.

Write Deployable MATLAB Code

In this section...
"Compiled Applications Do Not Process MATLAB Files at Runtime" on page 3-12
"Do Not Rely on Changing Directory or Path to Control the Execution of MATLAB Files" on page 3-13
"Use ismcc and isdeployed Functions To Execute Deployment-Specific Code Paths" on page 3-14
"Gradually Refactor Applications That Depend on Noncompilable Functions" on page 3-14
"Do Not Create or Use Nonconstant Static State Variables" on page 3-15
"Get Proper Licenses for Toolbox Functionality You Want to Deploy" on page 3-15

Compiled Applications Do Not Process MATLAB Files at Runtime

MATLAB Compiler secures your code against unauthorized changes. Deployable MATLAB files are suspended or frozen at the time MATLAB Compiler encrypts them—they do not change from that point onward. This does not mean that you cannot deploy a flexible application—it means that *you must design your application with flexibility in mind*. If you want the end user to be able to choose between two different methods, for example, both methods must be available in the built component.

The MCR only works on MATLAB code that was encrypted when the component was built. Any function or process that dynamically generates new MATLAB code will not work against the MCR.

Some MATLAB toolboxes, such as the Neural Network Toolbox™ product, generate MATLAB code dynamically. Because the MCR only executes encrypted MATLAB files, and the Neural Network Toolbox generates unencrypted MATLAB files, some functions in the Neural Network Toolbox cannot be deployed.

Similarly, functions that need to examine the contents of a MATLAB function file cannot be deployed. HELP, for example, is dynamic and not available in deployed mode. You can use LOADLIBRARY in deployed mode if you provide it with a MATLAB function prototype.

Instead of compiling the function that generates the MATLAB code and attempting to deploy it, perform the following tasks:

1 Run the code once in MATLAB to obtain your generated function.

2 Compile the MATLAB code with MATLAB Compiler, including the generated function.

Tip Another alternative to using EVAL or FEVAL is using anonymous function handles.

If you require the ability to create MATLAB code for dynamic run time processing, your end users must have an installed copy of MATLAB.

Do Not Rely on Changing Directory or Path to Control the Execution of MATLAB Files

In general, good programming practices advise against redirecting a program search path dynamically within the code. Many developers are prone to this behavior since it mimics the actions they usually perform on the command line. However, this can lead to problems when deploying code.

For example, in a deployed application, the MATLAB and Java paths are fixed and cannot change. Therefore, any attempts to change these paths (using the cd command or the addpath command) fails

If you find you cannot avoid placing addpath calls in your MATLAB code, use ismcc and isdeployed. See the next section for details.

Use ismcc and isdeployed Functions To Execute Deployment-Specific Code Paths

The `isdeployed` function allows you to specify which portion of your MATLAB code is deployable, and which is not. Such specification minimizes your compilation errors and helps create more efficient, maintainable code.

For example, you find it unavoidable to use `addpath` when writing your `startup.m`. Using `ismcc` and `isdeployed`, you specify when and what is compiled and executed.

For an example of using `isdeployed`, see "Passing Arguments to and from a Standalone Application" on page 6-26.

Gradually Refactor Applications That Depend on Noncompilable Functions

Over time, refactor, streamline, and modularize MATLAB code containing non-compilable or non-deployable functions that use `ismcc` and `isdeployed`. Your eventual goal is "graceful degradation" of non-deployable code. In other words, the code must present the end user with as few obstacles to deployment as possible until it is practically eliminated.

Partition your code into design-time and run time code sections:

- *Design-time code* is code that is currently evolving. Almost all code goes through a phase of perpetual rewriting, debugging, and optimization. In some toolboxes, such as the Neural Network Toolbox product, the code goes through a period of self-training as it reacts to various data permutations and patterns. Such code is almost never designed to be deployed.

- *Run-time code*, on the other hand, has solidified or become stable—it is in a finished state and is ready to be deployed by the end user.

Consider creating a separate directory for code that is not meant to be deployed or for code that calls undeployable code.

Do Not Create or Use Nonconstant Static State Variables

Avoid using the following:

- Global variables in MATLAB code

- Static variables in MEX-files

- Static variables in Java code

The state of these variables is persistent and shared with everything in the process.

When deploying applications, using persistent variables can cause problems because the MCR process runs in a single thread. You cannot load more than one of these non-constant, static variables into the same process. In addition, these static variables do not work well in multithreaded applications.

When programming with the builder components, you should be aware that an instance of the MCR is created for each instance of a new class. If the same class is instantiated again using a different variable name, it is attached to the MCR created by the previous instance of the same class. In short, if an assembly contains n unique classes, there will be maximum of n instances of MCRs created, each corresponding to one or more instances of one of the classes.

If you must use static variables, bind them to instances. For example, defining instance variables in a Java class is preferable to defining the variable as `static`.

Note This guideline does not apply to MATLAB Builder EX. When programming with Microsoft Excel, you can assign global variables to large matrices that persist between calls.

Get Proper Licenses for Toolbox Functionality You Want to Deploy

You must have a valid MathWorks license for toolboxes you use to create deployable components.

If you do not have a valid license for your toolbox, you cannot create a deployable component with it.

How the Deployment Products Process MATLAB Function Signatures

In this section...
"MATLAB Function Signature" on page 3-17
"MATLAB Programming Basics" on page 3-17

MATLAB Function Signature

MATLAB supports multiple signatures for function calls.

The generic MATLAB function has the following structure:

```
function [Out1,Out2,...,varargout]=foo(In1,In2,...,varargin)
```

To the *left* of the equal sign, the function specifies a set of explicit and optional return arguments.

To the *right* of the equal sign, the function lists explicit *input* arguments followed by one or more optional arguments.

All arguments represent a specific MATLAB type.

When the compiler or builder product processes your MATLAB code, it creates several overloaded methods that implement the MATLAB functions. Each of these overloaded methods corresponds to a call to the generic MATLAB function with a specific number of input arguments.

In addition to these methods, the builder creates another method that defines the return values of the MATLAB function as an input argument. This method simulates the `feval` external API interface in MATLAB.

MATLAB Programming Basics

Creating a Deployable MATLAB Function

Virtually any calculation that you can create in MATLAB can be deployed, if it resides in a function. For example:

```
>> 1 + 1
```

cannot be deployed.

However, the following calculation:

```
function result = addSomeNumbers()
 result = 1+1;
end
```

can be deployed because the calculation now resides in a function.

Taking Inputs into a Function

You typically pass inputs to a function. You can use primitive data type as an input into a function.

To pass inputs, put them in parentheses. For example:

```
function result = addSomeNumbers(number1, number2)
 result = number1 + number2;
end
```

Load MATLAB Libraries using loadlibrary

> **Note** It is important to understand the difference between the following:
>
> - MATLAB `loadlibrary` function — Loads shared library into MATLAB.
>
> - Operating system `loadlibrary` function — Loads specified Windows or UNIX operating system module into the address space of the calling process.

With MATLAB Compiler version 4.0 (R14) and later, you can use MATLAB file prototypes as described below to load your library in a compiled application. Loading libraries using H-file headers is not supported in compiled applications. This behavior occurs when `loadlibrary` is compiled with the header argument as in the statement:

```
loadlibrary(library, header)
```

In order to work around this issue, execute the following at the MATLAB command prompt:

```
loadlibrary(library, header, 'mfilename', 'mylibrarymfile');
```

where *mylibrarymfile* is the name of a MATLAB file you would like to use when loading this library. This step only needs to be performed once to generate a MATLAB file for the library.

In the code that is to be compiled, you can now call `loadlibrary` with the following syntax:

```
loadlibrary(library, @mylibrarymfile, 'alias', alias)
```

It is only required to add the prototype `.m` file and `.dll` file to the CTF archive of the deployed application. There is no need for `.h` files and C/C++ compilers to be installed on the deployment machine if the prototype file is used.

Once the prototype file is generated, add the file to the CTF archive of the application being compiled. You can do this with the `-a` option (if using the

mcc command) or by dragging it under **Other/Additional Files** (as a helper file) if using the Deployment Tool.

With MATLAB Compiler versions 4.0.1 (R14+) and later, generated MATLAB files will automatically be included in the CTF file as part of the compilation process. For MATLAB Compiler versions 4.0 (R14) and later, include your library MATLAB file in the compilation with the `-a` option with mcc.

Restrictions on Using MATLAB Function loadlibrary with MATLAB Compiler

Note the following limitations in regards to using loadlibrary with MATLAB Compiler. For complete documentation and up to date restrictions on loadlibrary, please reference the MATLAB documentation.

- You can not use loadlibrary inside of MATLAB to load a shared library built with MATLAB Compiler.

- With MATLAB Compiler Version 3.0 (R13SP1) and earlier, you cannot compile calls to loadlibrary because of general restrictions and limitations of the product.

Use MATLAB Data Files (MAT Files) in Compiled Applications

Explicitly Including MAT files Using the %#function Pragma

MATLAB Compiler excludes MAT files from "Dependency Analysis Function (depfun)" on page 3-7 by default.

If you want MATLAB Compiler to explicitly inspect data within a MAT file, you need to specify the %#function pragma when writing your MATLAB code.

For example, if you are creating a solution with Neural Network Toolbox, you need to use the %#function pragma within your GUI code to include a dependency on the gmdistribution class, for instance.

Load and Save Functions

If your deployed application uses MATLAB data files (MAT-files), it is helpful to code LOAD and SAVE functions to manipulate the data and store it for later processing.

- Use isdeployed to determine if your code is running in or out of the MATLAB workspace.

- Specify the data file by either using WHICH (to locate its full path name) define it relative to the location of ctfroot.

- All MAT-files are unchanged after mcc runs. These files are not encrypted when written to the CTF archive.

For more information about CTF archives, see "Component Technology File (CTF Archive)" on page 3-8.

See the ctfroot reference page for more information about ctfroot.

Use the following example as a template for manipulating your MATLAB data inside, and outside, of MATLAB.

Using Load/Save Functions to Process MATLAB Data for Deployed Applications

The following example specifies three MATLAB data files:

- user_data.mat

- userdata\extra_data.mat

- ..\externdata\extern_data.mat

1 Navigate to *matlab_root*\extern\examples\compiler\Data_Handling.

2 Compile ex_loadsave.m with the following mcc command:

```
mcc -mv ex_loadsave.m -a 'user_data.mat' -a
     '.\userdata\extra_data.mat' -a
     '..\externdata\extern_data.mat'
```

ex_loadsave.m.

```
function ex_loadsave
% This example shows how to work with the
% "load/save" functions on data files in
% deployed mode. There are three source data files
% in this example.
%    user_data.mat
%    userdata\extra_data.mat
%    ..\externdata\extern_data.mat
%
% Compile this example with the mcc command:
%     mcc -m ex_loadsave.m -a 'user_data.mat' -a
%     '.\userdata\extra_data.mat'
%         -a '..\externdata\extern_data.mat'
% All the folders under the current main MATLAB file directory will
%    be included as
% relative path to ctfroot; All other folders will have the
%    folder
% structure included in the ctf archive file from root of the
```

```
%     disk drive.
%
% If a data file is outside of the main MATLAB file path,
%     the absolute path will be
% included in ctf and extracted under ctfroot. For example:
%   Data file
%      "c:\$matlabroot\examples\externdata\extern_data.mat"
%      will be added into ctf and extracted to
%   "$ctfroot\$matlabroot\examples\externdata\extern_data.mat".
%
% All mat/data files are unchanged after mcc runs. There is
% no excryption on these user included data files. They are
% included in the ctf archive.
%
% The target data file is:
%   .\output\saved_data.mat
%   When writing the file to local disk, do not save any files
%    under ctfroot since it may be refreshed and deleted
%   when the application isnext started.

%==== load data file =============================
if isdeployed
    % In deployed mode, all file under CTFRoot in the path are loaded
    % by full path name or relative to $ctfroot.
    % LOADFILENAME1=which(fullfile(ctfroot,mfilename,'user_data.mat'));
    % LOADFILENAME2=which(fullfile(ctfroot,'userdata','extra_data.mat'));
    LOADFILENAME1=which(fullfile('user_data.mat'));
    LOADFILENAME2=which(fullfile('extra_data.mat'));
    % For external data file, full path will be added into ctf;
    % you don't need specify the full path to find the file.
    LOADFILENAME3=which(fullfile('extern_data.mat'));
else
    %running the code in MATLAB
    LOADFILENAME1=fullfile(matlabroot,'extern','examples','compiler',
                                     'Data_Handling','user_data.mat');
    LOADFILENAME2=fullfile(matlabroot,'extern','examples','compiler',
                              'Data_Handling','userdata','extra_data.mat');
    LOADFILENAME3=fullfile(matlabroot,'extern','examples','compiler',
                                     'externdata','extern_data.mat');
end
```

```
% Load the data file from current working directory
disp(['Load A from : ',LOADFILENAME1]);
load(LOADFILENAME1,'data1');
disp('A= ');
disp(data1);

% Load the data file from sub directory
disp(['Load B from : ',LOADFILENAME2]);
load(LOADFILENAME2,'data2');
disp('B= ');
disp(data2);

% Load extern data outside of current working directory
disp(['Load extern data from : ',LOADFILENAME3]);
load(LOADFILENAME3);
disp('ext_data= ');
disp(ext_data);

%==== multiple the data matrix by 2 ==============
result = data1*data2;
disp('A * B = ');
disp(result);

%==== save  the new data to a new file ===========
SAVEPATH=strcat(pwd,filesep,'output');
if ( ~isdir(SAVEPATH))
    mkdir(SAVEPATH);
end
SAVEFILENAME=strcat(SAVEPATH,filesep,'saved_data.mat');
disp(['Save the A * B result to : ',SAVEFILENAME]);
save(SAVEFILENAME, 'result');
```

MATLAB Objects

When working with MATLAB objects, remember to include the following statement in your MAT file:

```
%#function class_constructor
```

Using the `%#function` pragma in this manner forces depfun to load needed class definitions, enabling the MCR to successfully load the object.

4

C and C++ Standalone Executable and Shared Library Creation

Supported Compilation Targets

In this section...
"When to Create a Standalone Application" on page 4-2
"What's the Difference Between a Windows Standalone Application and a Console/Standalone Application?" on page 4-2
"When to Create a Shared Library" on page 4-3

When to Create a Standalone Application

Standalone Applications (or *standalones*) are C or C++ executable binaries designed to be executed on demand, usually by a single user or process.

Standalones are self-contained executable binaries and, when bundled with the MATLAB Compiler Runtime (MCR) can be a powerful solution when rolled out to a limited group of users. For example, creating a standalone from the MATLAB function magic enables you (or an end user) to execute that function by simply double-clicking or running magic.exe, created by MATLAB Compiler.

The non-dependent nature of a standalone makes it ideal for deploying to a limited numbers of users, usually in informal, loosely structured test and research development environments. This differs from a shared library, for example, which is more often integrated with enterprise C and C++ applications.

What's the Difference Between a Windows Standalone Application and a Console/Standalone Application?

If you are using a non-Windows operating system, "console applications" are referred to as "standalone applications".

Windows standalones differ from regular standalones in that Windows standalones suppress their MS-DOS window output.

The equivalent method to specify a Windows standalone target on the `mcc` command line is "-e Suppress MS-DOS Command Window" on page 12-28.

Note If you are using a non-Windows operating system, console applications are referred to as standalone applications.

When to Create a Shared Library

A shared library is a file that is intended to be shared by executable files, usually as part of an enterprise or large-scale application.

Creating a shared library allows you to create code that can be integrated into applications coded in either C or C++.

It is an ideal solution for MATLAB programmers who want to share their code with a large number of users, generally in a highly-structured software development environment.

Shared libraries contain C or C++ code that is often run automatically when a program is started. Modules used by a program are loaded from individual shared libraries into memory at load time or run time, rather than being copied statically, by a linker, when it creates a single monolithic executable file for the program.

For example, the MATLAB Compiler Runtime (MCR), which contains numerous executables called by user application code, is made up of shared libraries.

C Shared Libraries

Many procedural or non-object oriented applications run against *C shared libraries*. C is a popular language for interfacing with system infrastructures and other forms of low-level computing. Since C applications often utilize low-level features of computer operating systems, they generally cannot be ported easily to other platforms.

C++ Shared Libraries

C++ shared libraries have the advantage of utilizing object-oriented methodology and can also incorporate C language subroutines, if needed. C++ is generally used to develop enterprise applications that closely mimic the logic of business and real-world systems. Like C applications, however, C++ applications generally cannot be ported easily to other platforms.

Standalone Executable and Shared Library Creation From MATLAB Code

In this section...
"Build Standalone Executables and Shared Libraries Using the Deployment Tool " on page 4-5
"Build Standalone Executables and Shared Libraries Using the Command Line (mcc)" on page 4-5
"Watch a Video" on page 4-7

Build Standalone Executables and Shared Libraries Using the Deployment Tool

For a complete overview of the process of building and integrating an application from start to finish with the graphical Deployment Tool, read in the *MATLAB Compiler User's Guide*.

Build Standalone Executables and Shared Libraries Using the Command Line (mcc)

You can use the command line to execute the Deployment Tool GUI as well run `mcc` command.

- "Building Standalone Applications and Shared Libraries Using the Command Line" on page 4-6

- "Using the Deployment Tool from the Command Line" on page 4-6

Note The Deployment Tool command line interface (CLI) can be run from the MATLAB command line window, the Windows command line, or a UNIX shell.

Building Standalone Applications and Shared Libraries Using the Command Line

Instead of the GUI, you can use the mcc command to run MATLAB Compiler. The following table shows sample commands to create a standalone application or a shared library using mcc at the operating system prompt.

Desired Result	Command
Standalone application from the MATLAB file mymfunction	`mcc -m mymfunction.m`
	Creates a standalone application named mymfunction.exe on Windows platforms and mymfunction on platforms that are not Windows.
C shared library from the MATLAB files file1.m, file2.m, and file3.m	`mcc -B csharedlib:libfiles file1.m file2.m file3.m`
	Creates a shared library named libfiles.dll on Windows, libfiles.so on Linux, and libfiles.dylib on Mac OS X.
C++ shared library from the MATLAB files file1.m, file2.m, and file3.m	`mcc -B cpplib:libfiles file1.m file2.m file3.m`
	Creates a shared library named libfiles.dll on Windows, libfiles.so on Linux, and libfiles.dylib on Mac OS X.

Using the Deployment Tool from the Command Line

Start the Deployment Tool from the command line by using one of the following options.

You can start deploytool in this manner on all platforms supported by MATLAB Compiler.

Desired Results	Command
Start Deployment Tool GUI with the **New/Open** dialog box active	`deploytool` (default) or `deploytool -n`
Start Deployment Tool GUI and load *project_name*	`deploytool project_name.prj`

Desired Results	Command
Start Deployment Tool command line interface and build project_name after initializing	`deploytool -build project_name.prj`
Start Deployment Tool command line interface and package project_name after initializing	`deploytool -package project_name.prj`
Start Deployment Tool and package an existing project from the Command Line Interface. Specifying the *package_name* is optional. By default, a project is packaged into a .zip file. On Windows, if the *package_name* ends with .exe, the project is packaged into a self-extracting .exe.	`deploytool -package project_name.prj package_name`
Display MATLAB Help for the deploytool command	`deploytool -?`

Watch a Video

Watch a video about deploying applications using MATLAB Compiler.

Input and Output Files

In this section...

Standalone Executable

In this example, MATLAB Compiler takes the MATLAB files `foo.m` and `bar.m` as input and generates a standalone called `foo`.

```
mcc -m foo.m bar.m
```

File	Description
foo	The main file of the application. This file reads and executes the content stored in the embedded CTF archive. On Windows, this file is `foo.exe`.
run_*component*.sh	`mcc` generates `run_<component>.sh` file on UNIX (including Mac) systems for standalone applications. It temporarily sets up the environment variables needed at runtime and executes the application. On Windows, `mcc` doesn't generate this run script file, because the environment variables have already been set up by the installer. In this case, you just run your standalone `.exe` file.

C Shared Library

In this example, MATLAB Compiler takes the MATLAB files foo.m and bar.m as input and generates a C shared library called libfoo.

```
mcc -W lib:libfoo -T link:lib foo.m bar.m
```

File	Description
libfoo.c	The library wrapper C source file containing the exported functions of the library representing the C interface to the two MATLAB functions (foo.m and bar.m) as well as library initialization code.
libfoo.h	The library wrapper header file. This file is included by applications that call the exported functions of libfoo.
libfoo_mcc_component_data.c	C source file containing data needed by the MCR to initialize and use the library. This data includes path information, encryption keys, and other initialization for the MCR.
libfoo.exports	The exports file used by mbuild to link the library.
libfoo	The shared library binary file. On Windows, this file is libfoo.dll. **Note** UNIX extensions vary depending on the platform. See the External Interfaces documentation for additional information.

File	Description
libname.exp	Exports file used by the linker. The linker uses the export file to build a program that contains exports, usually a dynamic-link library (.dll). The import library is used to resolve references to those exports in other programs.
libname.lib	Import library. An import library is used to validate that a certain identifier is legal, and will be present in the program when the .dll is loaded. The linker uses the information from the import library to build the lookup table for using identifiers that are not included in the .dll. When an application or .dll is linked, an import library may be generated, which will be used for all future .dlls that depend on the symbols in the application or .dll.

C++ Shared Library

In this example, MATLAB Compiler takes the MATLAB files `foo.m` and `bar.m` as input and generates a C++ shared library called `libfoo`.

```
mcc -W cpplib:libfoo -T link:lib foo.m bar.m
```

File	Description
`libfoo.cpp`	The library wrapper C++ source file containing the exported functions of the library representing the C++ interface to the two MATLAB functions (`foo.m` and `bar.m`) as well as library initialization code.
`libfoo.h`	The library wrapper header file. This file is included by applications that call the exported functions of `libfoo`.
`libfoo_mcc_component_data.c`	C++ source file containing data needed by the MCR to initialize and use the library. This data includes path information, encryption keys, and other initialization for the MCR.
`libfoo.exports`	The exports file used by `mbuild` to link the library.
`libfoo`	The shared library binary file. On Windows, this file is `libfoo.dll`. **Note** UNIX extensions vary depending on the platform. See the External Interfaces documentation for additional information.

File	Description
libname.exp	Exports file used by the linker. The linker uses the export file to build a program that contains exports (usually a dynamic-link library (.dll). The import library is used to resolve references to those exports in other programs.
libname.lib	Import library. An import library is used to validate that a certain identifier is legal, and will be present in the program when the .dll is loaded. The linker uses the information from the import library to build the lookup table for using identifiers that are not included in the .dll. When an application or .dll is linked, an import library may be generated, which will need to be used for all future .dlls that depend on the symbols in the application or .dll.

Macintosh 64 (Maci64)

For 64-bit Macintosh, a Macintosh application bundle is created.

File	Description
foo.app	The bundle created for executable foo. Execution of the bundle occurs through foo.app/Contents/MacOS/foo.
foo	Application
run_component.sh	The generated shell script which executes the application through the bundle.

Dependency Analysis Function (depfun) and User Interaction with the Compilation Path

addpath and rmpath in MATLAB

If you run MATLAB Compiler from the MATLAB prompt, you can use the addpath and rmpath commands to modify the MATLAB path before doing a compilation. There are two disadvantages:

- The path is modified for the current MATLAB session only.

- If MATLAB Compiler is run outside of MATLAB, this doesn't work unless a savepath is done in MATLAB.

Note The path is also modified for any interactive work you are doing in the MATLAB environment as well.

Passing -I <directory> on the Command Line

You can use the -I option to add a folder to the beginning of the list of paths to use for the current compilation. This feature is useful when you are compiling files that are in folders currently not on the MATLAB path.

Passing -N and -p <directory> on the Command Line

There are two MATLAB Compiler options that provide more detailed manipulation of the path. This feature acts like a "filter" applied to the MATLAB path for a given compilation. The first option is -N. Passing -N on the mcc command line effectively clears the path of all folders except the following core folders (this list is subject to change over time):

- *matlabroot*\toolbox\matlab

- *matlabroot*\toolbox\local

- *matlabroot*\toolbox\compiler\deploy

- *matlabroot*\toolbox\compiler

It also retains all subfolders of the above list that appear on the MATLAB path at compile time. Including -N on the command line allows you to replace folders from the original path, while retaining the relative ordering of the included folders. All subfolders of the included folders that appear on the original path are also included. In addition, the -N option retains all folders that the user has included on the path that are not under *matlabroot*\toolbox.

Use the -p option to add a folder to the compilation path in an order-sensitive context, i.e., the same order in which they are found on your MATLAB path. The syntax is

```
p <directory>
```

where <directory> is the folder to be included. If <directory> is not an absolute path, it is assumed to be under the current working folder. The rules for how these folders are included are

- If a folder is included with -p that is on the original MATLAB path, the folder and all its subfolders that appear on the original path are added to the compilation path in an order-sensitive context.

- If a folder is included with -p that is not on the original MATLAB path, that folder is not included in the compilation. (You can use -I to add it.)

- If a path is added with the -I option while this feature is active (-N has been passed) and it is already on the MATLAB path, it is added in the order-sensitive context as if it were included with -p. Otherwise, the folder is added to the head of the path, as it normally would be with -I.

Note The -p option requires the -N option on the mcc command line.

5

Deployment Process

This chapter tells you how to deploy compiled MATLAB code to developers and to end users.

- "Overview" on page 5-2
- "Deploying to Developers" on page 5-3
- "Deploying to End Users" on page 5-9
- "Working with the MCR" on page 5-17
- "Deploy Applications Created Using Parallel Computing Toolbox" on page 5-39
- "Deploying a Standalone Application on a Network Drive (Windows Only)" on page 5-46
- "MATLAB® Compiler™ Deployment Messages" on page 5-48
- "Using MATLAB® Compiler™ Generated DLLs in Windows Services" on page 5-49
- "Reserving Memory for Deployed Applications with MATLAB Memory Shielding" on page 5-50

Overview

After you create a library, a component, or an application, the next step is typically to deploy it to others to use on their machines, independent of the MATLAB environment. These users can be developers who want to use the library or component to develop an application, or end users who want to run a standalone application.

- "Deploying to Developers" on page 5-3
- "Deploying to End Users" on page 5-9

Note When you deploy, you provide the wrappers for the compiled MATLAB code and the software needed to support the wrappers, including the MCR. The MCR is version specific, so you must ensure that developers as well as users have the proper version of the MCR installed on their machines.

Watch a Video

Watch a video about deploying applications using MATLAB Compiler.

Deploying to Developers

In this section...
"Procedure" on page 5-3
"What Software Does a Developer Need?" on page 5-4
"Ensuring Memory for Deployed Applications" on page 5-8

Procedure

Note If you are programming on the same machine where you created the component, you can skip the steps described here.

---

1 Create a package that contains the software necessary to support the compiled MATLAB code. It is frequently helpful to install the MCR on development machines, for testing purposes. See "What Software Does a Developer Need?" on page 5-4

Note You can use the Deployment Tool to create a package for developers. For Windows platforms, the package created by the Deployment Tool is a self-extracting executable. For UNIX platforms, the package created by the Deployment Tool is a zip file that must be decompressed and installed manually. See to get started using the Deployment Tool.

---

2 Write instructions for how to use the package.

   a If your package was created with the Deployment Tool, Windows developers can just run the self-extracting executable created by the Deployment Tool. UNIX developers must unzip and install manually.

   b All developers must set path environment variables properly. See .

3 Distribute the package and instructions.

What Software Does a Developer Need?

The software that you provide to a developer who wants to use compiled MATLAB code depends on which of the following kinds of software the developer will be using:

- "Standalone Application" on page 5-4
- "C or C++ Shared Library" on page 5-5
- ".NET Component" on page 5-6
- "COM Component" on page 5-6
- "Java Component" on page 5-7
- "COM Component to Use with Microsoft® Excel®" on page 5-7

Standalone Application

To distribute a standalone application created with MATLAB Compiler to a development machine, create a package that includes the following files.

Software Module	Description
MCR Installer (Windows)	The MCR Installer is a self-extracting executable that installs the necessary components to develop your application. This file is included with MATLAB Compiler. Run mcrinstaller function to obtain name of executable.
MCR Installer (Linux)	The MCR Installer is a self-extracting executable that installs the necessary components to develop your application on UNIX machines (other than Mac). This file is included with MATLAB Compiler. Run mcrinstaller function to obtain name of binary.
MCR Installer (Mac)	Run mcrinstaller function to obtain name of binary.

Software Module	Description
application_name.exe (Windows)	Application created by MATLAB Compiler. Maci64 must include the bundle directory hierarchy.
application_name (UNIX)	
application_name.app (Maci64)	

Note If you are using a non-Windows operating system, "console applications" are referred to as "standalone applications".

C or C++ Shared Library

To distribute a shared library created with MATLAB Compiler to a development machine, create a package that includes the following files.

Software Module	Description
MCR Installer (Windows)	MATLAB Compiler Runtime library archive; platform-dependent file that must correspond to the end user's platform. Run mcrinstaller function to obtain name of executable.
MCR Installer (Mac)	The MCR Installer is a self-extracting executable that installs the necessary components to develop your application on Mac machines. This file is included with MATLAB Compiler. Run mcrinstaller function to obtain name of binary.
MCR Installer (Linux)	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform. Run mcrinstaller function to obtain name of binary.
libmatrix	Shared library; extension varies by platform, for example, DLL on Windows

Software Module	Description
libmatrix.h	Library header file
libmatrix.lib	Application library file needed to create the driver application for the shared library.

.NET Component

To distribute a .NET component to a development machine, create a package that includes the following files.

Software Module	Description
componentName.xml	Documentation files
componentName.pdb (if **Debug** option is selected)	Program Database File, which contains debugging information
componentName.dll	Component assembly file
MCR Installer	MCR Installer (if not already installed on the target machine). Run mcrinstaller function to obtain name of executable.

COM Component

To distribute a COM component to a development machine, create a package that includes the following files.

Software Module	Description
mwcomutil.dll	Utilities required for array processing. Provides type definitions used in data conversion.
componentname_ version.dll	Component that contains compiled MATLAB code.
MCR Installer	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform.

Software Module	Description
	The MCR Installer installs MATLAB Compiler Runtime (MCR), which users of your component need to install on the target machine once per release. Run `mcrinstaller` function to obtain name of executable.

Java Component

To distribute a Java component to a development machine, create a package that includes the *componentname*.jar file, a Java package containing the Java interface to MATLAB code.

Note For more information, see the `MWArray` Javadoc, which is searchable from the Help or from the MathWorks Web site.

COM Component to Use with Microsoft Excel

To distribute a COM component for Excel to a development machine, create a package that includes the following files.

Software Module	Description
componentname_projectversion.dll	Compiled component.
MCR Installer	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform.
	The MCR Installer installs the MATLAB Compiler Runtime (MCR), which users of your component need to install on the target machine once per release. Run `mcrinstaller`

Software Module	Description
	function to obtain name of executable.
`*.xla`	Any user-created Excel add-in files found in the `<projectdir>\distrib` folder

Ensuring Memory for Deployed Applications

If you are having trouble obtaining memory for your deployed application, use MATLAB Memory Shielding for deployed applications to ensure a maximum amount of contiguous allocated memory. See "Reserving Memory for Deployed Applications with MATLAB Memory Shielding" on page 5-50 for more information.

Deploying to End Users

In this section...
"Steps by the Developer to Deploy to End Users" on page 5-9
"What Software Does the End User Need?" on page 5-12
"Using Relative Paths with Project Files" on page 5-15
"Porting Generated Code to a Different Platform" on page 5-15
"Extracting a CTF Archive Without Executing the Component" on page 5-15
"Ensuring Memory for Deployed Applications" on page 5-16

Steps by the Developer to Deploy to End Users

For an end user to run an application or use a library that contains compiled MATLAB code, there are two sets of tasks. Some tasks are for the developer who developed the application or library, and some tasks are for the end user.

1 Create a package that contains the software needed at run time. See "What Software Does a Developer Need?" on page 5-4 for more details.

Note The package for end users must include the `.ctf` file, which includes all the files in your preferences folder. Be aware of the following with regards to preferences:

- MATLAB preferences set at compile time are inherited by the compiled application. Therefore, include no files in your preferences folder that you do not want exposed to end users.

- Preferences set by a compiled application do not affect the MATLAB preferences, and preferences set in MATLAB do not affect a compiled application until that application is recompiled. MATLAB does not save your preferences folder until you exit MATLAB. Therefore, if you change your MATLAB preferences, stop and restart MATLAB before attempting to recompile using your new preferences.

.

The preferences folder is as follows:

- `$HOME/.matlab/`*current_release* on UNIX

- *system root*`\profiles\`*user*`\application data\mathworks\`
  `matlab\`*current_release* on Windows

The folder will be stored in the CTF archive in a folder with a generated name, such as:

mwapplication`_mcr\`*myapplication*`_7CBEDC3E1DB3D462C18914C13CBFA649`.

---

2 Write instructions for the end user. See "Steps by the End User" on page 5-10.

3 Distribute the package to your end user, along with the instructions.

Steps by the End User

1 Open the package containing the software needed at run time.

2 Run `MCRInstaller` *once* on the target machine, that is, the machine where you want to run the application or library. The `MCRInstaller` opens a

command window and begins preparation for the installation. See "Using the MCR Installer GUI" on page 5-11.

3 If you are deploying a Java application to end users, they must set the class path on the target machine.

Note for Windows® Applications You must have administrative privileges to install the MCR on a target machine since it modifies both the system registry and the system path.

Running the MCRInstaller after the MCR has been set up on the target machine requires only user-level privileges.

Using the MCR Installer GUI

1 When the MCR Installer wizard appears, click **Next** to begin the installation. Click **Next** to continue.

2 In the Select Installation Folder dialog box, specify where you want to install the MCR and whether you want to install the MCR for just yourself or others. Click **Next** to continue.

Note The **Install MATLAB Compiler Runtime for yourself, or for anyone who uses this computer** option is not implemented for this release. The current default is **Everyone**.

3 Confirm your selections by clicking **Next**.

The installation begins. The process takes some time due to the quantity of files that are installed.

The MCR Installer automatically:

- Copies the necessary files to the target folder you specified.
- Registers the components as needed.

- Updates the system path to point to the MCR binary folder, which is
  `<target_directory>\<version>\runtime\win32|win64`.

4 When the installation completes, click **Close** on the Installation Completed
dialog box to exit.

What Software Does the End User Need?

The software required by end users depends on which of the following kinds
of software is to be run by the user:

- "Standalone Compiled Application That Accesses Shared Library" on page
  5-12
- ".NET Application" on page 5-13
- "COM Application" on page 5-13
- "Java Application" on page 5-14
- "Microsoft® Excel® Add-in" on page 5-14

Standalone Compiled Application That Accesses Shared Library

To distribute a shared library created with MATLAB Compiler to end users,
create a package that includes the following files.

Component	Description
MCR Installer (Windows)	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform.
matrixdriver.exe (Windows) matrixdriver (UNIX)	Application
libmatrix	Shared library; extension varies by platform. Extensions are: • Windows — .dll

Component	Description
	• Linux, Linux x86-64 — .so
	• Mac OS X — .dylib

.NET Application

To distribute a .NET application that uses components created with MATLAB Builder NE, create a package that includes the following files.

Software Module	Description
componentName.xml	Documentation files
componentName.pdb (if **Debug** option is selected)	Program Database File, which contains debugging information
componentName.dll	Component assembly file
MCR Installer	MCR Installer (if not already installed on the target machine). Run mcrinstaller function to obtain name of executable.
application.exe	Application

COM Application

To distribute a COM application that uses components created with MATLAB Builder NE or MATLAB Builder EX, create a package that includes the following files.

Software Module	Description
componentname.ctf	Component Technology File (ctf) archive. This is a platform-dependent file that must correspond to the end user's platform.
componentname _version.dll	Component that contains compiled MATLAB code
_install.bat	Script run by the self-extracting executable

Software Module	Description
MCR Installer	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform.
	The MCR Installer installs MATLAB Compiler Runtime (MCR), which users of your component need to install on the target machine once per release. Run `mcrinstaller` function to obtain name of executable.
application.exe	Application

Java Application

To distribute a Java application created with MATLAB Builder JA, create a *componentname*.jar file. To deploy the application on computers without MATLAB, you must include the MCR when creating your Java component.

Microsoft Excel Add-in

To distribute an Excel add-in created with MATLAB Builder EX, create a package that includes the following files.

Software Module	Description
componentname _version.dll	Component that contains compiled MATLAB code
_install.bat	Script run by the self-extracting executable
MCR Installer	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform. Run `mcrinstaller` function to obtain name of executable.
*.xla	Any Excel add-in files found in *projectdirectory*\distrib

Using Relative Paths with Project Files

Project files now support the use of relative paths as of R2007b of MATLAB Compiler, enabling you to share a single project file for convenient deployment over the network. Simply share your project folder and use relative paths to define your project location to your distributed computers.

Porting Generated Code to a Different Platform

You can distribute an application generated by MATLAB Compiler to any target machine that has the same operating system as the machine on which the application was compiled. For example, if you want to deploy an application to a Windows machine, you must use the Windows version of MATLAB Compiler to build the application on a Windows machine.

Note Since binary formats are different on each platform, the components generated by MATLAB Compiler cannot be moved from platform to platform as is.

To deploy an application to a machine with an operating system different from the machine used to develop the application, you must rebuild the application on the desired targeted platform. For example, if you want to deploy a previous application developed on a Windows machine to a Linux machine, you must use MATLAB Compiler on a Linux machine and completely rebuild the application. You must have a valid MATLAB Compiler license on both platforms to do this.

Extracting a CTF Archive Without Executing the Component

CTF archives contain content (MATLAB files and MEX-files) that need to be extracted from the archive before they can be executed. In order to extract the archive you must override the default CTF embedding option (see "MCR Component Cache and CTF Archive Embedding" on page 6-14). To do this, ensure that you compile your component with the "-C Do Not Embed CTF Archive by Default" on page 12-26 option.

The CTF archive automatically expands the first time you run a MATLAB Compiler-based component (a MATLAB Compiler based standalone application or an application that calls a MATLAB Compiler-based shared library, COM, or .NET component).

To expand an archive without running the application, you can use the extractCTF (.exe on Windows) standalone utility provided in the *matlabroot*\toolbox\compiler*arch* folder, where *arch* is your system architecture, Windows = win32|win64, Linux = glnx86, x86-64 = glnxa64, and Mac OS X = mac. This utility takes the CTF archive as input and expands it into the folder in which it resides. For example, this command expands hello.ctf into the folder where it resides:

extractCTF hello.ctf

The archive expands into a folder called hello_mcr. In general, the name of the folder containing the expanded archive is <componentname>_mcr, where componentname is the name of the CTF archive without the extension.

Note To run extractCTF from any folder, you must add *matlabroot*\toolbox\compiler*arch* to your PATH environment variable. Run extractCTF.exe from a system prompt. If you run it from MATLAB, be sure to use the bang (!) operator.

Ensuring Memory for Deployed Applications

If you are having trouble obtaining memory for your deployed application, use MATLAB Memory Shielding for deployed applications to ensure a maximum amount of contiguous allocated memory. See "Reserving Memory for Deployed Applications with MATLAB Memory Shielding" on page 5-50 for more information.

Working with the MCR

In this section...
"About the MATLAB® Compiler™ Runtime (MCR)" on page 5-17
"The MCR Installer" on page 5-18
"Installing the MCR Non-Interactively (Silent Mode)" on page 5-27
"Removing (Uninstalling) the MCR" on page 5-28
"MCR Startup Options" on page 5-31
"Using the MCR User Data Interface" on page 5-34
"Displaying MCR Initialization Start-Up and Completion Messages For Users" on page 5-36

About the MATLAB Compiler Runtime (MCR)

MATLAB Compiler uses the MATLAB Compiler Runtime (MCR), a standalone set of shared libraries that enables the execution of MATLAB files on computers without an installed version of MATLAB.

If you distribute your compiled components to end-users who do not have MATLAB installed on their systems, they must install the MATLAB Compiler Runtime (MCR) on their computers or know the location of a network-installed MCR. When you packaged your compiled component, you have the option of including the MCR in the package you distribute to users, or they can download it from the Web at `http://www.mathworks.com/products/compiler/mcr`. The MCR only needs to be installed once a user system.

Note There is no way to distribute your application with any subset of the files that are installed by the MCR Installer.

See "The MCR Installer" on page 5-18 for more information.

How is the MCR Different from MATLAB?

This MCR differs from MATLAB in several important ways:

- In the MCR, MATLAB files are securely encrypted for portability and integrity.

- MATLAB has a desktop graphical interface. The MCR has all the MATLAB functionality without the graphical interface.

- The MCR is version-specific. You must run your applications with the version of the MCR associated with the version of MATLAB Compiler with which it was created. For example, if you compiled an application using version 4.10 (R2009a) of MATLAB Compiler, users who do not have MATLAB installed must have version 7.10 of the MCR installed. Use `mcrversion` to return the version number of the MCR.

- The MATLAB and Java paths in an MCR instance are fixed and cannot be changed. To change them, you must first customize them within MATLAB.

Performance Considerations and the MCR

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

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

The MCR makes use of thread locking so that only one thread is allowed to access the MCR at a time. As a result, calls into the MCR are threadsafe for MATLAB Compiler generated libraries, COM objects, and .NET objects. On the other hand, this can impact performance.

The MCR Installer

Download the MCR from the Web at
http://www.mathworks.com/products/compiler/mcr.

Installing the MCR

To install the MCR, users of your component must run the MCR Installer. When you packaged your compiled component for distribution, you had the

option to embed the MCR in your application package, or specify the network location of the MCR.

To install the MCR on any computer, perform these steps.

1 Start the MCR Installer. How you accomplish this depends on your computer.

Computer	Steps
Windows	Double-click the compiled component package self-extracting archive file, typically named *my_program*_pkg.exe, where *my_program* is the name of the compiled component. This extracts the MCR Installer from the archive, along with all the files that make up the MCR. Once all the files have been extracted, the MCR Installer starts automatically.
Linux Mac	Extract the contents of the compiled component package, which is a Zip file on Linux systems, typically named, *my_program*_pkg.zip, where *my_program* is the name of the compiled component. Use the unzip command to extract the files from the package. unzip MCRInstaller.zip Run the MCR Installer script, from the directory where you unzipped the package file, by entering: ./install For example, if you unzipped the package and MCR Installer in \home\USER, you run the ./install from \home\USER. **Note** On Mac systems, you may need to enter an administrator username and password after you run ./install.

2 When the MCR Installer starts, it displays the following dialog box. Read the information and then click **Next** to proceed with the installation.
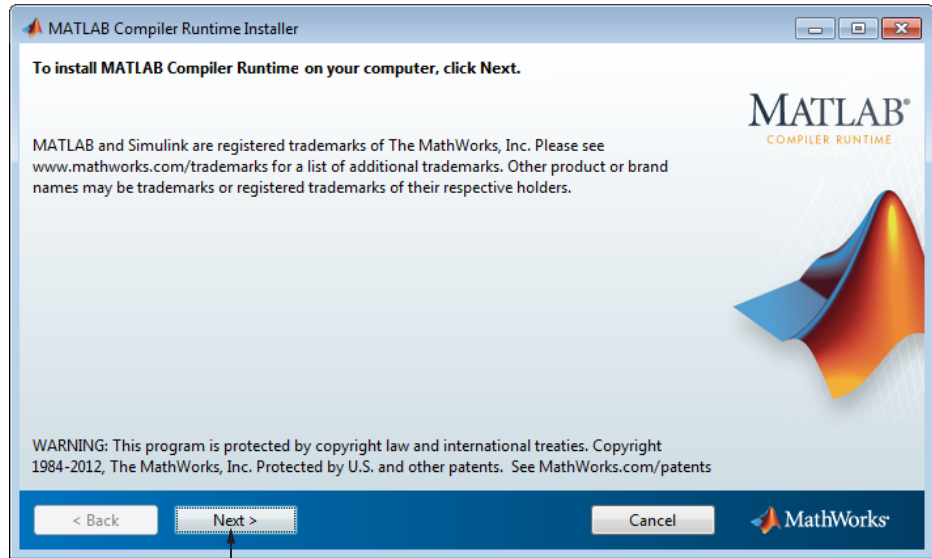
Click Next.

3 Specify the folder in which you want to install the MCR in the Folder Selection dialog box.

Note On Windows systems, you can have multiple installations of different versions of the MCR on your computer but only one installation for any particular version. If you already have an existing installation, the MCR Installer does not display the Folder Selection dialog box because you can only overwrite the existing installation in the same folder. On Linux and Mac systems, you can have multiple installations of the same version of the MCR.

Specify installation folder.
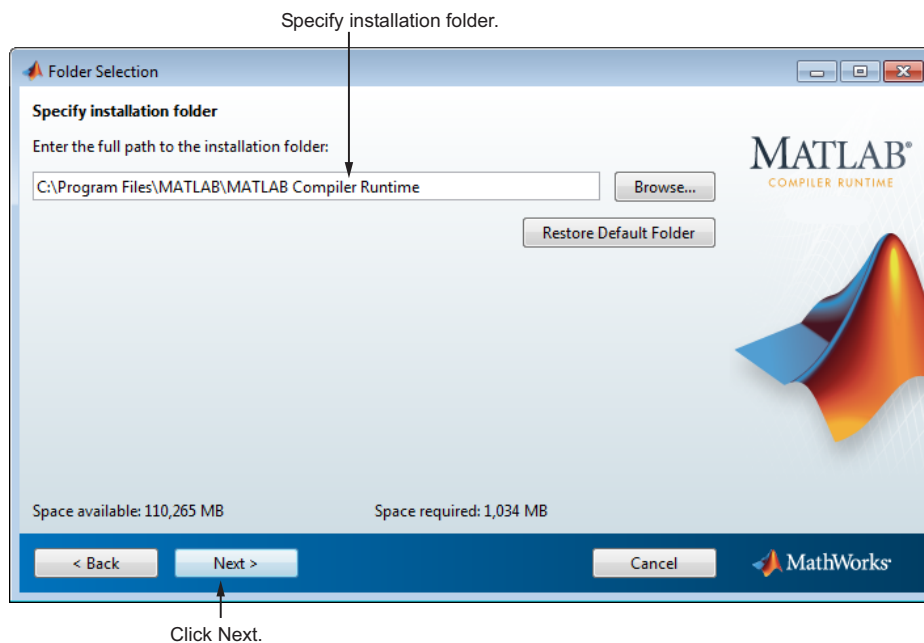
Click Next.

4 Confirm your choices and click **Next**. The MCR Installer starts copying files into the installation folder.

Con firm your choices.

Click Install.
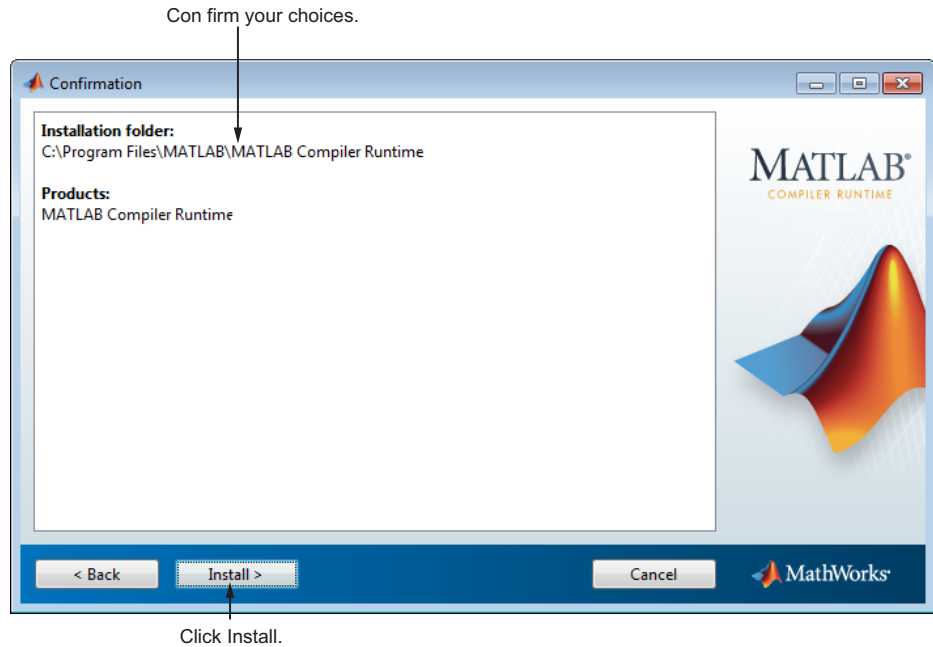
5 On Linux and Macintosh systems, after copying files to your disk, the MCR Installer displays the Product Configuration Notes dialog box. This dialog box contains information necessary for setting your path environment variables. Copy the path information from this dialog box and then click **Next**.
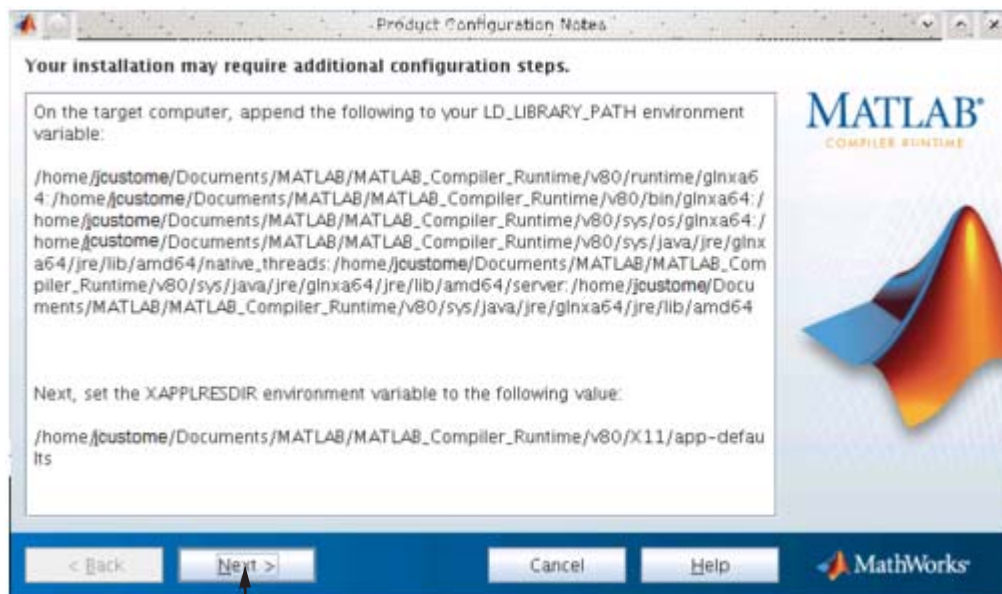
Click Next.

6 Click **Finish** to exit the MCR Installer.

Click Finish.
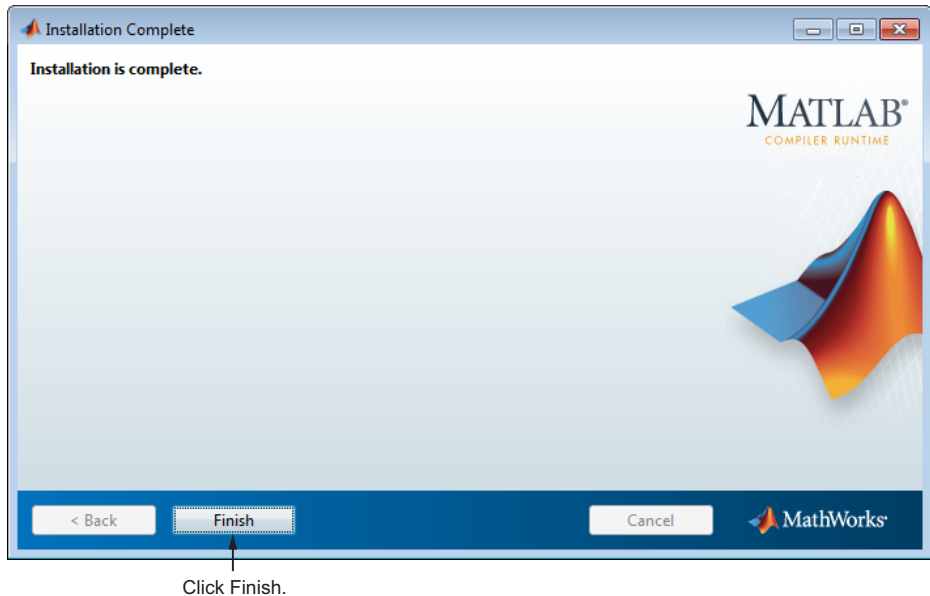
MCR Installer Readme File. A `readme.txt` file is included with the MCR Installer. This file, visible when the MCR Installer is expanded, provides more detailed information about the installer and the switches that can be used with it.

Installing the MCR and MATLAB on the Same Machine

You do not need to install the MCR on your machine if your machine has both MATLAB and MATLAB Compiler installed. The version of MATLAB should be the same as the version of MATLAB that was used to create the deployed component.

You can, however, install the MCR for debugging purposes. See "Modifying the Path" on page 5-25.

Caution   There is a limitation regarding folders on your path. If the target machine has a MATLAB installation, the <mcr_root> folders must be first on the path to run the deployed application. To run MATLAB, the *matlabroot* folders must be first on the path. This restriction only applies to profiles involving an installed MCR and an installed MATLAB on the same machine.

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

- **Windows**

  To run deployed components against the MCR install, *mcr_root**ver*\\runtime\\win32|win64 must appear on your system path before *matlabroot*\\runtime\\win32|win64.

  If *mcr_root**ver*\\runtime*arch* appears first on the compiled application path, the application uses the files in the MCR install area.

  If *matlabroot*\\runtime*arch* appears first on the compiled application path, the application uses the files in the MATLAB Compiler installation area.

- **UNIX**

  To run deployed components against the MCR install, on Linux, Linux x86-64, or the <mcr_root>/runtime/<arch> folder must appear on your LD_LIBRARY_PATH before *matlabroot*/runtime/<arch>. See for the platform-specific commands.

  To run deployed components on Mac OS X, the <mcr_root>/runtime folder must appear on your DYLD_LIBRARY_PATH before *matlabroot*/runtime/<arch>.

  To run MATLAB on Mac OS X or Intel® Mac, *matlabroot*/runtime/<arch> must appear on your DYLD_LIBRARY_PATH before the <mcr_root>/bin folder.

Note For detailed information about setting MCR paths on UNIX variants such as Mac and Linux, see Appendix B, "Using MATLAB® Compiler™ on Mac or Linux" for complete deployment and troubleshooting information.

Installing Multiple MCRs on One Machine

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

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

Note for Mac OS X Users Installing multiple versions of the MCR on the same machine is not supported on Mac OS X. When you receive a new version of MATLAB, you must recompile and redeploy all of your applications and components. Also, when you install a new MCR onto a target machine, you must delete the old version of the MCR and install the new one. You can only have one version of the MCR on the target machine.

Deploying a Recompiled Application. Always run your compiled applications with the version of the MCR that corresponds to the MATLAB version with which your application was built. If you upgrade your MATLAB Compiler software on your development machine and distribute the recompiled application to your users, you should also distribute the corresponding version of the MCR. Users should upgrade their MCR to the new version. If users need to maintain multiple versions of the MCR on their systems, refer to "Installing Multiple MCRs on One Machine" on page 5-26 for more information.

Installing the MCR Non-Interactively (Silent Mode)

To install the MCR without having to interact with the installer dialog boxes, use one of the MCR installer non-interactive modes: silent or automated.

Mode	Description
silent	MCR installer runs as a background task and does not display any dialog boxes.
automated	MCR installer displays the dialog boxes but does not wait for user interaction.

When run in silent or automated mode, the MCR installer uses default values for installation options, such as the name of the destination folder. You can override these defaults by using MCR installer command-line options.

Note If you have already installed the MCR for a particular release in the default location, the installer overwrites the existing installation, when running in silent or automated mode.

1 Extract the contents of the MCR installer file to a temporary folder, called $temp in this documentation.

On Windows systems, double-click the MCR installer self-extracting archive file, MCRinstaller.exe. You might have to first extract the MCR installer from the compiled component archive, if you received a package file from the component developer.

On Linux and Mac systems, use the unzip command:

unzip MCRInstaller.zip

2 Run the MCR installer, specifying the mode option on the command line.

The install script is created in the folder in which you have unzipped the MCR.

Execute the MCR installer, specifying the mode argument.

```
setup.exe -mode silent
```

On a Linux or Mac computer, run the MCR installer script, specifying the mode argument.

```
./install -mode silent
```

3 View a log of the installation.

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

On Linux and Mac systems, the MCR installer displays the log information at the command prompt, unless you redirect it to a file.

Customizing a Silent Installation

If you want to specify the installation folder or the name of the log file, use MCR installer command line options. For example, to specify the installation folder, use the `-destinationFolder` option, as follows:

```
setup.exe -mode silent -destinationFolder C:\my_folder
```

To specify the name and location of the installation log file, use the `-outputFile` option, as follows:

```
setup.exe -mode silent -destinationFolder C:\my_folder
                              -outputFile C:\my_log.txt
```

Removing (Uninstalling) the MCR

The method you use to remove (uninstall) the MCR from your computer varies depending on the type of computer.

You can remove unwanted versions before or after installation of a more recent version of the MCR, as versions can be installed or removed in any order.

Windows

1 Start the uninstaller. From the Windows Start menu, search for the **Add or Remove Programs** control panel, and double-click MATLAB Compiler

Runtime in the list. You can also launch the MCR Uninstaller from the *mcr_root*\uninstall\bin*arch* folder, where *mcr_root* is your MCR installation folder and *arch* is an architecture-specific folder, such as win64.

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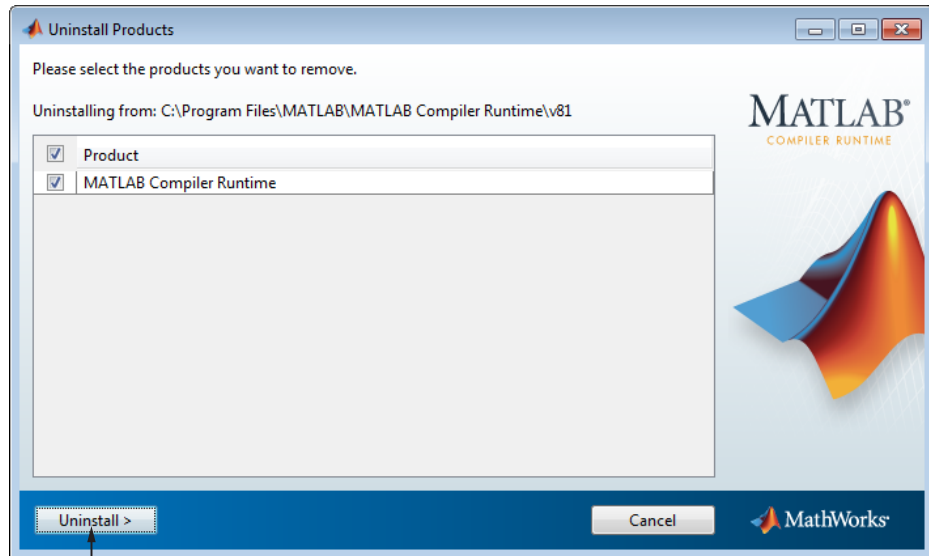
Click Uninstall.

3 After the MCR uninstaller removes the files from your disk, it displays the Uninstallation Complete dialog box. Click **Finish** to exit the MCR uninstaller.
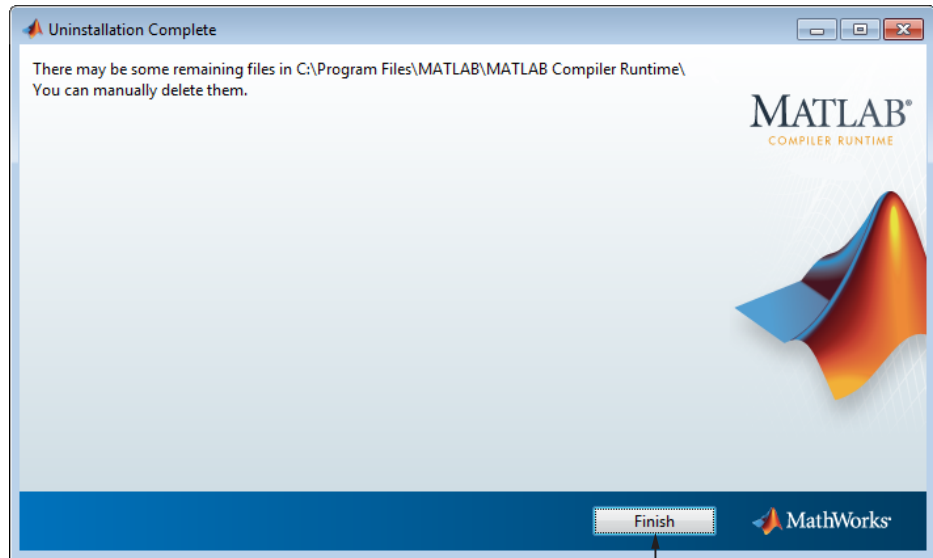
Click Finish.

Linux

1 Exit the application.

2 Enter this command at the Linux prompt:

```
rm -rf mcr_root
```

where *mcr_root* represents the name of your top-level MATLAB installation folder.

Mac

- Exit the application.

- Navigate to your MCR installation folder. For example, the installation folder might be named MATLAB_Compiler_Runtime.app in your Applications folder.

where *mcr_root* represents the name of your top-level MATLAB installation folder.

• Drag your MCR installation folder to the trash, and then select **Empty Trash** from the Finder menu.

MCR Startup Options

Setting MCR Options

Set MCR options, such as -nojvm, -nodisplay, or -logfile by performing either of the following tasks.

• Using the Deployment Tool, click the Settings icon (⚙▾). Using the **Advanced** tab in the Project Settings dialog box, enter the options you want to select in the **MCR Startup Options** area.

• Using the mcc command, specify the -R switch.

Using the Deployment Tool. In the MCR Startup Options area of the Project Settings dialog box, you can set the following options.

Setting MCR Startup Options Using the Deployment Tool GUI

MCR Startup Option	This option...	Set the options by...
-nojvm	Disables the Java Virtual Machine, which is enabled by default. This can help improve MCR performance.	Select the **Disable JVM** checkbox.
-nodisplay	On Linux, launches the MCR without display functionality.	In the **MCR Options** box, enter -nodisplay.
-logfile	Writes information about MCR startup to a logfile.	Select the **Create log file** checkbox. You *must* enter the path to the logfile, including the logfile name, in the **Log File** box.

Setting MCR Startup Options Using the mcc Command Line. When you use the command line, specify the -R switch to invoke the MCR startup options you want to use.

Following are examples of using mcc -R to invoke -nojvm, -nodisplay, and -logfile when building a C standalone (designated by the -m switch).

Setting -nojvm

```
mcc -m -R -nojvm -v foo.m
```

Setting -nodisplay (Linux Only)

```
mcc -m -R -nodisplay -v foo.m
```

Setting -logfile

```
mcc -e -R '-logfile,bar.txt' -v foo.m
```

Setting -nojvm, -nodisplay, and -logfile With One Command

```
mcc -m -R '-logfile,bar.txt,-nojvm,-nodisplay' -v foo.m
```

Retrieving MCR Startup Options (Shared Libraries Only)

Use these functions to return data about MCR state when working with shared libraries (this does not apply to standalone applications).

Function and Signature	When to Use	Return Value
bool mclIsMCRInitialized()	Use mclIsMCRInitialized() to determine whether or not the MCR has been properly initialized.	Boolean (true or false). Returns true if MCR is already initialized, else returns false.
bool mclIsJVMEnabled()	Use mclIsJVMEnabled() to determine if the MCR was launched with an instance of a Java Virtual Machine (JVM).	Boolean (true or false). Returns true if MCR is launched with a JVM instance, else returns false.

Function and Signature	When to Use	Return Value
`const char* mclGetLogFileName()`	Use `mclGetLogFileName()` to retrieve the name of the log file used by the MCR	Character string representing log file name used by MCR
`bool mclIsNoDisplaySet()`	Use `mclIsNoDisplaySet()` to determine if `-nodisplay` option is enabled.	Boolean (`true` or `false`). Returns `true` if `-nodisplay` is enabled, else returns `false`.
		Note `false` is always returned on Windows systems since the `-nodisplay` option is not supported on Windows systems.
		Caution When running on Mac, if `-nodisplay` is used as one of the options included in `mclInitializeApplication`, then the call to `mclInitializeApplication` must occur before calling `mclRunMain`.

Note All of these attributes have properties of write-once, read-only.

Retrieving Information About MCR Startup Options.

```
const char* options[4];
    options[0] = "-logfile";
    options[1] = "logfile.txt";
    options[2] = "-nojvm";
    options[3] = "-nodisplay";
```

5-33

```
if( !mclInitializeApplication(options,4) )
{
    fprintf(stderr,
            "Could not initialize the application.\n");
    return -1;
}
printf("MCR initialized : %d\n", mclIsMCRInitialized());
printf("JVM initialized : %d\n", mclIsJVMEnabled());
printf("Logfile name : %s\n", mclGetLogFileName());
printf("nodisplay set : %d\n", mclIsNoDisplaySet());
fflush(stdout);
```

Using the MCR User Data Interface

The MCR User Data Interface lets you easily access MCR data. It allows keys and values to be passed between an MCR instance, the MATLAB code running on the MCR, and the wrapper code that created the MCR. Through calls to the MCR User Data Interface API, you access MCR data by creating a per-MCR-instance associative array of mxArrays, consisting of a mapping from string keys to mxArray values. Reasons for doing this include, but are not limited to the following:

• You need to supply run-time profile information to a client running an application created with the Parallel Computing Toolbox. You supply and change profile information on a per-execution basis. For example, two instances of the same application may run simultaneously with different profiles. See "Deploy Applications Created Using Parallel Computing Toolbox" on page 5-39 for more information.

• You want to set up a global workspace, a global variable or variables that MATLAB and your client can access.

• You want to store the state of any variable or group of variables.

The API consists of:

• Two MATLAB functions callable from within deployed application MATLAB code

• Four external C functions callable from within deployed application wrapper code

---

Note The MATLAB functions are available to other modules since they are native to MATLAB. These built-in functions are implemented in the MCLMCR module, which lives in the standalone folder.

For implementations using .NET components, Java components, or COM components with Excel, see the MATLAB Builder NE, MATLAB Builder JA, and MATLAB Builder EX documentation, respectively.

MATLAB Functions

Use the MATLAB functions `getmcruserdata` and `setmcruserdata` from deployed MATLAB applications. They are loaded by default only in applications created with the MATLAB Compiler or builder products. See "Using the MCR User Data Interface" on page 5-34 for more information.

Tip When calling the MATLAB functions `getmcruserdata` and `setmcruserdata`, remember the following:

- These functions will produce an `Unknown function` error when called in MATLAB if the MCLMCR module cannot be located. This can be avoided by calling `isdeployed` before calling `getmcruserdata` and `setmcruserdata`. For more information about the `isdeployed` function, see the `isdeployed` reference page.

- The MATLAB functions `getmcruserdata` and `setmcruserdata` can be dragged and dropped (as you would any other MATLAB file), directly to the `deploytool` GUI.

Setting MCR Data for Standalone Executables

MCR data can be set for a standalone executable with the `-mcruserdata` command line argument.

The following example demonstrates how to set MCR user data for use with a Parallel Computing Toolbox profile `.mat` file:

```
parallelapp.exe -mcruserdata
```

ParallelProfile:config.mat

The argument following -mcruserdata is interpreted as a key/value MCR user data pair, where the colon separates the key from the value. The standalone executable accesses this data by using getmcruserdata.

Note A compiled application should set mcruserdata ParallelProfile *before* calling any Parallel Computing Toolbox™ code. Once this code has been called, setting ParallelProfile to point to a different file has no effect.

Setting and Retrieving MCR Data for Shared Libraries

As mentioned in "Using the MCR User Data Interface" on page 5-34, there are many possible scenarios for working with MCR Data. The most general scenario involves setting the MCR with specific data for later retrieval, as follows:

1 In your code, Include the MCR header file and the library header generated by MATLAB Compiler.

2 Properly initialize your application using mclInitializeApplication.

3 After creating your input data, write or "set" it to the MCR with setmcruserdata .

4 After calling functions or performing other processing, retrieve the new MCR data with getmcruserdata.

5 Free up storage memory in work areas by disposing of unneeded arrays with mxDestroyArray.

6 Shut down your application properly with mclTerminateApplication.

Displaying MCR Initialization Start-Up and Completion Messages For Users

You can display a console message for end users that informs them when MCR initialization starts and completes.

To create these messages, use the -R option of the `mcc` command.

You have the following options:

- Use the default start-up message only (`Initializing MATLAB Compiler Runtime version *x.xx*`)

- Customize the start-up or completion message with text of your choice. The default start-up message will also display prior to displaying your customized start-up message.

Some examples of different ways to invoke this option follow:

This command:	Displays:
`mcc -R -startmsg`	Default start-up message `Initializing MATLAB Compiler Runtime version` *x.xx*
`mcc -R -startmsg,'`*user customized message*`'`	Default start-up message `Initializing MATLAB Compiler Runtime version` *x.xx* and *user customized message* for start-up
`mcc -R -completemsg,'`*user customized message*`'`	Default start-up message `Initializing MATLAB Compiler Runtime version` *x.xx* and *user customized message* for completion
`mcc -R -startmsg,'`*user customized message*`' -R -completemsg,'`*user customized message*`"`	Default start-up message `Initializing MATLAB Compiler Runtime version` *x.xx* and *user customized message* for both start-up and completion by specifying `-R` before each option
`mcc -R -startmsg,'`*user customized message*`',-completemsg,'`*user customized message*`'`	Default start-up message `Initializing MATLAB Compiler Runtime version` *x.xx* and *user customized message* for both start-up and completion by specifying `-R` only once

Best Practices

Keep the following in mind when using `mcc -R`:

- When calling `mcc` in the MATLAB Command Window, place the comma inside the single quote. For example:

  ```
  mcc -m hello.m -R '-startmsg,"Message_Without_Space"'
  ```

- If your initialization message has a space in it, call `mcc` from the system console or use `!mcc` from MATLAB.

Deploy Applications Created Using Parallel Computing Toolbox

For information about using the MCR User Data Interface see "Improving Data Access Using the MCR User Data Interface" in the MATLAB Builder JA, MATLAB Builder NE, and MATLAB Builder EX User's Guides.

Compile and Deploy a Standalone Application with the Parallel Computing Toolbox

Standalone Applications with Profile Passed at Run-Time

This example shows how to compile a standalone application, using functions from the Parallel Computing Toolbox, with deploytool. In this case, the profile is passed at runtime.

Note Standalone executables and shared libraries generated from MATLAB Compiler for parallel applications can now launch up to twelve local workers without MATLAB Distributed Computing Server™.

Step 1: Write Your Parallel Computing Toolbox Code

1 Compile sample_pct.m in MATLAB.

This example code uses the cluster defined in the default profile.

The output assumes that the default profile is local.

```
function speedup = sample_pct (n)
warning off all;
tic
if(ischar(n))
    n=str2double(n);
end
for ii = 1:n
   (cov(sin(magic(n)+rand(n,n))));
end
```

```
time1 =toc;
matlabpool('open');
tic
parfor ii = 1:n
    (cov(sin(magic(n)+rand(n,n))));
end
time2 =toc;
disp(['Normal loop times: ' num2str(time1) ...
    ',parallel loop time: ' num2str(time2) ]);
disp(['parallel speedup:  ' num2str(1/(time2/time1)) ...
    ' times faster than normal']);
matlabpool('close');
disp('done');
speedup = (time1/time2);
```

2 Run the code as follows after changing the default profile to `local`, if needed.

```
a = sample_pct(200)
```

3 Verify that you get the following results;

```
Starting matlabpool using the 'local'
                    profile ... connected to 4 labs.
Normal loop times: 1.4625, parallel loop time: 0.82891
parallel speedup:  1.7643 times faster than normal
Sending a stop signal to all the labs ... stopped.
done
a =
    1.7643
```

Step 2: Export Your Profile

As mentioned above, to compile and deploy Parallel Computing Toolbox code, you must have access to the MDCS cluster. This step shows you how to export the cluster profile.

1 In MATLAB, click **Parallel > Manage Cluster Profiles**. The Manage Cluster Profiles dialog opens.

2 Select your profile (representing the cluster), right click, and select **Export** to export the profile.

Step 3: Compile and Deploy Your Application

1 Follow the steps in to compile your application. When the compilation finishes, a new folder (with the same name as the project) is created. This folder contains two subfolders: distrib and src.

Project Name	pct_Compiled
File to compile (add to **Main File** area)	sample_pct.m

Note If you are using the GPU feature of Parallel Computing Toolbox, you need to manually add the PTX and CU files.

If you are using a Deployment Tool project, click **Add files/directories** on the **Build** tab.

If you are using the mcc command, use the -a option.

2 To deploy the compiled application, copy the distrib folder (containing the executable), the MCR Installer, and the profile, with the cluster information, to your end users. The packaging function of deploytool offers a convenient way to do this.

Note The end-user's target machine must have access to the cluster.

3 To run the deployed application, do the following:

a On the end-user machine, navigate to the folder containing the EXE file.

b Issue the following command:

```
pct_Compiled.exe 200
    -mcruserdata
```

```
ParallelProfile:C:\work9b\pctdeploytool\
                   pct_Compiled\distrib\myprofile.settings
```

Note As of R2012a, Parallel Configurations and MAT files have been replaced with Parallel Profiles. For more information, see the release notes for the Deployment products and Parallel Computing Toolbox.

To use existing MAT files and ensure backward compatibility with this change, issue a command such as the following, in the above example:

```
pct_Compiled.exe 200 -mcruserdata
    ParallelProfile:C:\work9b\pctdeploytool\pct_Compiled\distrib\myconfig.mat
```

If you continue to use MAT files, remember to specify the full path to the MAT file.

---

c Verify the output is as follows:

```
Starting matlabpool using the 'myprofile'
               profile ... connected
               to 4 labs.
Normal loop times: 1.5712, parallel loop time: 0.90766
parallel speedup:  1.7311 times faster than normal
Sending a stop signal to all the labs ... stopped.
Did not find any pre-existing parallel
jobs created by matlabpool.
done
```

Standalone Applications with Embedded Profile

This example shows how to compile a standalone application, using functions from the Parallel Computing Toolbox, with deploytool. In this case, the profile information is embedded in the CTF archive, and you set up the MCR to work with Parallel Computing Toolbox using the setmcruserdata function in MATLAB.

Note Standalone executables and shared libraries generated from MATLAB Compiler for parallel applications can now launch up to twelve local workers without MATLAB Distributed Computing Server.

Step 1: Write Your Parallel Computing Toolbox Code

Compile the following two files in MATLAB. Note that, in this example, run_sample_pct calls sample_pct and supplies the profile directly as MCR user data.

```
function speedup = sample_pct (n)
warning off all;
if(ischar(n))
    n=str2double(n);
end
tic
for ii = 1:n
   (cov(sin(magic(n)+rand(n,n))));
end
time1 =toc;
matlabpool('open');
tic
parfor ii = 1:n
   (cov(sin(magic(n)+rand(n,n))));
end
time2 =toc;
disp(['Normal loop times: ' num2str(time1) ...
    ', parallel loop time: ' num2str(time2) ]);
disp(['parallel speedup:  ' num2str(1/(time2/time1)) ...
    ' times faster than normal']);
matlabpool('close');
disp('done');
speedup = (time1/time2);
```

Step 2: Compile and Deploy Your Application

1 Follow the steps in to compile your application. When the compilation finishes, a new folder (with the same name as the project) is created. This folder contains two subfolders: distrib and src.

Project Name	pct_CompiledWithProfile
File to compile (add to **Main File** area)	run_sample_pct.m
Shared Resource and Helper Files	myprofile.settings sample_pct.m

2 To deploy the compiled application, copy and distribute the distrib folder (containing the executable) and the MCR Installer to your end users. The packaging function of deploytool offers a convenient way to do this.

Note The end-user's target machine must have access to the cluster.

---

3 To run the deployed application, do the following:

a On the end-user machine, navigate to directory containing the EXE file.

b Issue the following command. Note that, in this case, you are not supplying the profile explicitly, since it is included in the CTF archive.

```
pct_CompiledWithProfile.exe 200
```

c Verify the output is as follows:

```
Starting matlabpool using the 'myprofile'
                ... connected to 4 labs.
Normal loop times: 2.1289, parallel loop time: 1.227
parallel speedup:  1.735 times faster than normal
Sending a stop signal to all the labs ... stopped.
done
speedup =
    1.7350
```

Compile and Deploy a Shared Library with the Parallel Computing Toolbox

The process of deploying a C or C++ shared library with the Parallel Computing Toolbox is similar to deploying a standalone application.

1 Compile the shared library using the Deployment Tool.

2 Set the file in the C or C++ driver code using the `setmcruserdata` function. See the `setmcruserdata` function reference page for an example.

Note Standalone executables and shared libraries generated from MATLAB Compiler for parallel applications can now launch up to twelve local workers without MATLAB Distributed Computing Server.

Deploying a Standalone Application on a Network Drive (Windows Only)

You can deploy a compiled standalone application to a network drive so that it can be accessed by all network users without having them install the MCR on their individual machines.

Note There is no need to perform these steps on a Linux system.

There is no requirement for vcredist on Linux, and the component registration is in support of MATLAB Builder EX and MATLAB COM Builder, which both run on Windows only.

Distributing to a Linux network file system is exactly the same as distributing to a local file system. You only need to set up the LD_LIBRARY_PATH or use scripts which points to the MCR installation.

1 On any Windows machine, run mcrinstaller function to obtain name of the MCR Installer executable.

2 Copy the entire MCR folder (the folder where MCR is installed) onto a network drive.

3 Copy the compiled application into a separate folder in the network drive and add the path <mcr_root>\<ver>\runtime\<arch> to all client machines. All network users can then execute the application.

4 Run vcredist_x86.exe on for 32-bit clients; run vcredist_x64.exe for 64-bit clients.

5 If you are using MATLAB Builder EX, register mwcomutil.dll and mwcommgr.dll on every client machine.

If you are using MATLAB Builder NE (to create COM objects), register mwcomutil.dll on every client machine.

To register the DLLs, at the DOS prompt enter

```
mwregsvr <fully_qualified_pathname\dllname.dll>
```

These DLLs are located in *<mcr_root>**<ver>*\runtime*<arch>*.

Note These libraries are automatically registered on the machine on which the installer was run.

MATLAB Compiler Deployment Messages

To enable display of MATLAB Compiler deployment messages, see the *MATLAB Desktop Tools and Environment* documentation.

Using MATLAB Compiler Generated DLLs in Windows Services

If you have a Windows service that is built using DLL files generated by MATLAB Compiler, do the following to ensure stable performance:

1 Create a file named `java.opts`.

2 Add the following line to the file:

```
-Xrs
```

3 Save the file to: *MCRROOT**version*\runtime\win32|win64, where *MCRROOT* is the installation folder of the MATLAB Compiler Runtime and *version* is the MCR version (for example, `v74` for MATLAB Compiler 4.4 (R2006a)).

Caution   Failure to create the `java.opts` file using these steps may result in unpredictable results such as premature termination of Windows services.

Reserving Memory for Deployed Applications with MATLAB Memory Shielding

In this section...
"What Is MATLAB Memory Shielding and When Should You Use It?" on page 5-50
"Requirements for Using MATLAB Memory Shielding" on page 5-51
"Invoking MATLAB Memory Shielding for Your Deployed Application" on page 5-51

What Is MATLAB Memory Shielding and When Should You Use It?

Occasionally you encounter problems ensuring that you have the memory needed to run deployed applications. These problems often occur when:

- Your data set is large

- You are trying to compensate for the memory limitations inherent in a 32-bit Windows system

- The computer available to you has limited resources

- Network resources are restrictive

Use MATLAB Memory Shielding to ensure that you obtain the maximum amount of contiguous memory to run your deployed application successfully.

MATLAB Memory Shielding provides the specified level of protection of the address space used by MATLAB. When you use this feature, it reserves the largest contiguous block of memory available for your application after startup.

Memory shielding works by ensuring that resources, such as DLLs, load into locations that will not fragment the address space of the system. The feature provides the specified amount of contiguous address space you specify, up to the maximum available on the system.

For example, on a 32-bit Windows system, MATLAB defaults to memory shielding for virtual addresses `0x50000000-0x70000000`. At the point where your application runs, the shield lowers, allowing allocation of that virtual address space.

Note This topic describes how to invoke the shielding function for deployed applications, not the MATLAB workspace. To learn more about invoking memory shielding for MATLAB workspaces, see the discussion of the start-up option `matlab` *shieldOption* in the *MATLAB Function Reference Guide*.

Requirements for Using MATLAB Memory Shielding

Before using MATLAB Memory Shielding for your deployed applications, verify that you meet the following requirements:

- Your deployed application is failing because it cannot find the proper amount of memory and not for another unrelated reason. As a best practice, let the operating system attempt to satisfy runtime memory requests, if possible. See "What Is MATLAB Memory Shielding and When Should You Use It?" on page 5-50 for examples of cases where you can benefit by using MATLAB Memory Shielding

- Your application runs on a Windows 32-bit system. While MATLAB Memory Shielding runs on 64-bit Windows systems without failing, it has no effect on your application.

- You are running with a standalone application or Windows executable. MATLAB Memory Shielding does not work with shared libraries, .NET components or Java components.

- You have run the MCR Installer on your system to get the MATLAB Compiler Runtime (MCR). The memory shielding feature is installed with the MCR.

Invoking MATLAB Memory Shielding for Your Deployed Application

Invoke memory shielding by using either the command-line syntax or the GUI. Each approach has appropriate uses based on your specific memory reservation needs.

Using the Command Line

Use the command line if you want to invoke memory shielding only with the various *shield_level* values (not specific address ranges).

The base command-line syntax is:

```
MemShieldStarter [-help] [-gui]
     [-shield shield_level]
    fully-qualified_app_path
   [user-defined_app_arguments]
```

1 Run your application using the default level of memory shielding. Use the command:

   ```
   MemShieldStarter fully-qualified_app_path
                 [user-defined_app_arguments]
   ```

2 If your application runs successfully, try the next highest shield level to guarantee more contiguous memory, if needed.

   - A higher level of protection does not always provide a larger size block and can occasionally cause start-up problems. Therefore, start with a lower level of protection and be conservative when raising the level of protection.

   - Use only memory shielding levels that guarantee a successful execution of your application. See the table MemShieldStarter Options on page 5-53 for more details on which shield options to choose.

   - Contact your system administrator for further advice on successfully running your application.

3 If your application fails to start, disable memory shielding:

   a To disable memory shielding after you have enabled it, run the following command:

   ```
   MemShieldStarter -shield none
    fully-qualified_app_path
   [user-defined_app_arguments]
   ```

b Contact your system administrator for further advice on successfully running your application.

MemShieldStarter Options

Option	Description
`-help`	Invokes help for `MemShieldStarter`
`-gui`	Starts the Windows graphical interface for `MemShieldStarter.exe`. See "Using the GUI" on page 5-54 for more details.
`-shield` *shield_level*	See "Shield Level Options" on page 5-53.
fully-qualified_application_path	The fully qualified path to your user application
user-defined_application_arguments	Arguments passed to your user application. `MemShieldStarter.exe` only passes user arguments. It does not alter them.

Shield Level Options. *shield_level* options are as follows:

- `none` — This value completely disables memory shielding. Use this value if your application fails to start successfully with the default (`-shield minimum`) option.

- `minimum` — The option defaults to this setting. Minimum shielding protects the range `0x50000000` to `0x70000000` during startup until just before processing `matlabrc`. This value ensures at least approximately 500 MB of contiguous memory available up to this point.

   When experimenting with a shielding level. start with `minimum`. To use the default, do not specify a shield option upon startup. If your application fails to start successfully using `minimum`, use `-shield none`. If your application starts successfully with the default value for *shield_level*, try using the `-shield medium` option to guarantee more memory.

- `medium` — This value protects the same range as `minimum`, `0x50000000` to `0x70000000`, but protects the range until just after startup processes `matlabrc`. It ensures that there is at least approximately 500 MB of

contiguous memory up to this point. If MATLAB fails to start successfully with the -shield medium option, use the default option (-shield minimum). If MATLAB starts successfully with the -shield medium option and you want to try to ensure an even larger contiguous block after startup, try using the -shield maximum option.

- maximum — This value protects the maximum range, which can be up to approximately 1.5 GB, until just after startup processes matlabrc. The default memory shielding range for maximum covers 0x10000000 to 0x78000000. If MATLAB fails to start successfully with the -shield maximum option, use the -shield medium option.

Note The shielding range may vary in various locales. Contact your system administrator for further details.

Using the GUI

Use the graphical interface to invoke memory shielding for specific address ranges as well as with specific *shield_level* values.

1 To start the GUI, run the following at the system command prompt:

```
MemShieldStarter.exe -gui
```

The Memory Shielding Starter dialog box opens:
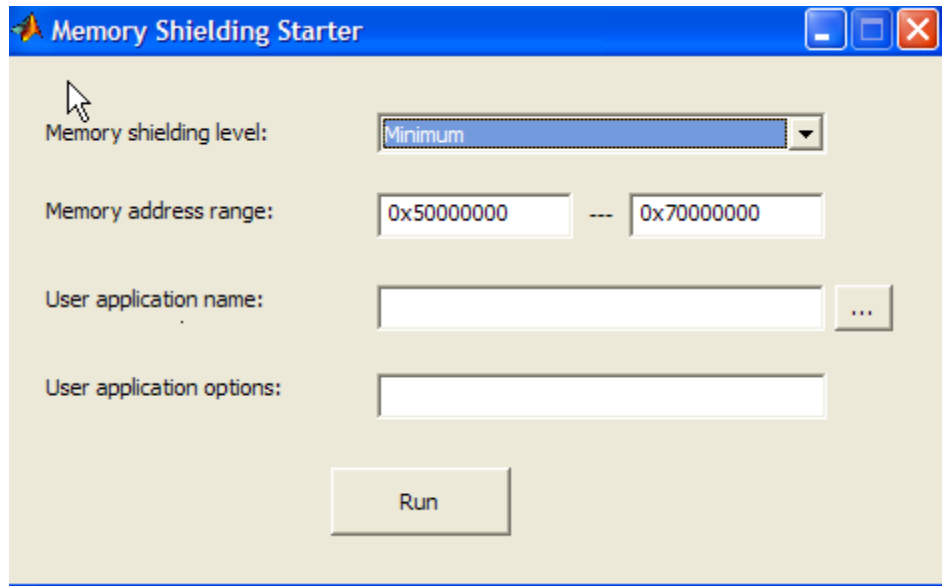
2 Enter the appropriate values as described in MemShieldStarter Options on page 5-53. Use the default **Memory shielding level** minimum.

You can specify a specific address range in the **Memory address range** fields. Specifying a range override the default 0x50000000 through 0x70000000 address range values required for the *shield_level* minimum, for example.

3 Click **Run**.

4 If your application runs successfully, try the next highest shield level to guarantee more contiguous memory, if needed.

- A higher level of protection does not always provide a larger size block and can occasionally cause startup problems. Therefore, start with a lower level of protection and use only what is necessary to guarantee a successful execution of your application.

- See the table MemShieldStarter Options on page 5-53 for more details on appropriate shield options for various situations.

Compiler Commands

This chapter describes mcc, which is the command that invokes MATLAB Compiler.

Command Overview

In this section...
"Compiler Options" on page 6-2
"Combining Options" on page 6-2
"Conflicting Options on the Command Line" on page 6-3
"Using File Extensions" on page 6-3
"Interfacing MATLAB Code to C/C++ Code" on page 6-4

Compiler Options

mcc is the MATLAB command that invokes MATLAB Compiler. You can issue the mcc command either from the MATLAB command prompt (MATLAB mode) or the DOS or UNIX command line (standalone mode).

You may specify one or more MATLAB Compiler option flags to mcc. Most option flags have a one-letter name. You can list options separately on the command line, for example,

```
mcc -m -v myfun
```

Macros are MathWorks supplied MATLAB Compiler options that simplify the more common compilation tasks. Instead of manually grouping several options together to perform a particular type of compilation, you can use a simple macro option. You can always use individual options to customize the compilation process to satisfy your particular needs. For more information on macros, see "Simplify Compilation Using Macros" on page 6-5.

Combining Options

You can group options that do not take arguments by preceding the list of option flags with a single dash (-), for example:

```
mcc -mv myfun
```

Options that take arguments cannot be combined unless you place the option with its arguments last in the list. For example, these formats are valid:

```
mcc -v -W main -T link:exe myfun    % Options listed separately
mcc -vW main -T link:exe myfun      % Options combined
```

This format is *not* valid:

```
mcc -Wv main -T link:exe myfun
```

In cases where you have more than one option that takes arguments, you can only include one of those options in a combined list and that option must be last. You can place multiple combined lists on the mcc command line.

If you include any C or C++ file names on the mcc command line, the files are passed directly to mbuild, along with any MATLAB Compiler generated C or C++ files.

Conflicting Options on the Command Line

If you use conflicting options, MATLAB Compiler resolves them from left to right, with the rightmost option taking precedence. For example, using the equivalencies in "Macro Options" on page 6-5,

```
mcc -m -W none test.m
```

is equivalent to:

```
mcc -W main -T link:exe -W none test.m
```

In this example, there are two conflicting -W options. After working from left to right, MATLAB Compiler determines that the rightmost option takes precedence, namely, -W none, and the product does not generate a wrapper.

Caution   Macros and regular options may both affect the same settings and may therefore override each other depending on their order in the command line.

Using File Extensions

The valid, recommended file extension for a file submitted to MATLAB Compiler is .m. Always specify the complete file name, including the .m

extension, when compiling with `mcc` or you may encounter unpredictable results.

Note P-files (`.p`) have precedence over MATLAB files, therefore if both P-files and MATLAB files reside in a folder, and a file name is specified without an extension, the P-file will be selected.

Interfacing MATLAB Code to C/C++ Code

To designate code to be compiled with C or C++, rewrite the C or C++ function as a MEX-file and simply call it from your application. The `%#EXTERNAL` pragma is no longer supported.

You can control whether the MEX-file or a MATLAB stub gets called by using the `isdeployed` function.

Code Proper Return Types From C and C++ Methods

To avoid potential problems, ensure all C methods you write (and reference from within MATLAB code) return a `bool` return type indicating the status. C++ methods should return nothing (`void`).

Simplify Compilation Using Macros

In this section...
"Macro Options" on page 6-5
"Working With Macro Options" on page 6-5

Macro Options

MATLAB Compiler, through its exhaustive set of options, gives you access to the tools you need to do your job. If you want a simplified approach to compilation, you can use one simple option, i.e., *macro*, that allows you to quickly accomplish basic compilation tasks. Macros let you group several options together to perform a particular type of compilation.

This table shows the relationship between the macro approach to accomplish a standard compilation and the multioption alternative.

Macro Option	Bundle File	Creates	Option Equivalence Function Wrapper | Output Stage | |
-l	macro_option_l	Library	-W lib -T link:lib
-m	macro_option_m	Standalone application	-W main -T link:exe

Working With Macro Options

The -m option tells MATLAB Compiler to produce a standalone application. The -m macro is equivalent to the series of options

```
-W main -T link:exe
```

This table shows the options that compose the -m macro and the information that they provide to MATLAB Compiler.

-m Macro

Option	Function
-W main	Produce a wrapper file suitable for a standalone application.
-T link:exe	Create an executable link as the output.

Changing Macro Options

You can change the meaning of a macro option by editing the corresponding macro_option bundle file in *matlabroot*\toolbox\compiler\bundles. For example, to change the -m macro, edit the file macro_option_m in the bundles folder.

Note This changes the meaning of -m for all users of this MATLAB installation.

Specifying Default Macro Options

As the MCCSTARTUP functionality has been replaced by bundle file technology, the macro_default file that resides in toolbox\compiler\bundles can be used to specify default options to the compiler.

For example, adding -mv to the macro_default file causes the command:

```
 mcc foo.m
```

to execute as though it were:

```
mcc -mv foo.m
```

Similarly, adding -v to the macro_default file causes the command:

```
mcc -W 'lib:libfoo' -T link:lib foo.m
```

to behave as though the command were:

```
mcc -v -W 'lib:libfoo' -T link:lib foo.m
```

Invoke MATLAB Build Options

Specifying Full Path Names to Build MATLAB Code

If you specify a full path name to a MATLAB file on the mcc command line, MATLAB Compiler

 1 Breaks the full name into the corresponding path name and file names (<path> and <file>).

 2 Replaces the full path name in the argument list with "-I <path> <file>".

Specifying Full Paths Names

For example:

```
mcc -m /home/user/myfile.m
```

would be treated as

```
mcc -m -I /home/user myfile.m
```

In rare situations, this behavior can lead to a potential source of confusion. For example, suppose you have two different MATLAB files that are both named myfile.m and they reside in /home/user/dir1 and /home/user/dir2. The command

```
mcc -m -I /home/user/dir1 /home/user/dir2/myfile.m
```

would be equivalent to

```
mcc -m -I /home/user/dir1 -I /home/user/dir2 myfile.m
```

MATLAB Compiler finds the `myfile.m` in `dir1` and compiles it instead of the one in `dir2` because of the behavior of the `-I` option. If you are concerned that this might be happening, you can specify the `-v` option and then see which MATLAB file MATLAB Compiler parses. The `-v` option prints the full path name to the MATLAB file during the dependency analysis phase.

Note MATLAB Compiler produces a warning (`specified_file_mismatch`) if a file with a full path name is included on the command line and MATLAB Compiler finds it somewhere else.

Using Bundle Files to Build MATLAB Code

Bundle files provide a convenient way to group sets of MATLAB Compiler options and recall them as needed. The syntax of the bundle file option is:

```
-B <filename>[:<a1>,<a2>,...,<an>]
```

When used on the `mcc` command line, the bundle option `-B` replaces the entire string with the contents of the specified file. The file should contain only `mcc` command-line options and corresponding arguments and/or other file names. The file may contain other `-B` options.

A bundle file can include replacement parameters for MATLAB Compiler options that accept names and version numbers. For example, there is a bundle file for C shared libraries, `csharedlib`, that consists of:

```
-W lib:%1% -T link:lib
```

To invoke MATLAB Compiler to produce a C shared library using this bundle, you can use:

```
mcc -B csharedlib:mysharedlib myfile.m myfile2.m
```

In general, each `%n%` in the bundle file will be replaced with the corresponding option specified to the bundle file. Use `%%` to include a `%` character. It is an error to pass too many or too few options to the bundle file.

Note You can use the -B option with a replacement expression as is at the DOS or UNIX prompt. To use -B with a replacement expression at the MATLAB prompt, you must enclose the expression that follows the -B in single quotes when there is more than one parameter passed. For example,

```
>>mcc -B csharedlib:libtimefun weekday data tic calendar toc
```

can be used as is at the MATLAB prompt because libtimefun is the only parameter being passed. If the example had two or more parameters, then the quotes would be necessary as in

```
>>mcc -B 'cexcel:component,class,1.0' ...
weekday data tic calendar toc
```

Bundle Files Available with MATLAB Compiler

See the following table for a list of bundle files available with MATLAB Compiler.

Bundle File	Creates	Contents
cpplib	C++ Library	-W cpplib:<shared_library_name> -T link:lib
csharedlib	C Shared Library	-W lib:<shared_library_name> -T link:lib

---

Note Additional bundle files are available when you have a license for products layered on MATLAB Compiler. For example, if you have a license for MATLAB Builder NE , you can use the mcc command with bundle files that create COM objects and .NET objects.

What Are Wrapper Files?

Wrapper files encapsulate, or wrap, the MATLAB files in your application with an interface that enables the MATLAB files to operate in a given target environment.

To provide the required interface, the wrapper does the following:

• Performs wrapper-specific initialization and termination

• Provides the dispatching of function calls to the MCR

To specify the type of wrapper to generate, use the following syntax:

```
-W <type>
```

The following sections detail the available wrapper types.

Wrapper Files

• "Main File Wrapper" on page 6-11

• "C Library Wrapper" on page 6-12

• "C++ Library Wrapper" on page 6-12

Main File Wrapper

The `-W main` option generates wrappers that are suitable for building standalone applications. These POSIX-compliant main wrappers accept strings from the POSIX shell and return a status code. They pass these command-line strings to the MATLAB file function(s) as MATLAB strings. They are meant to translate "command-like" MATLAB files into POSIX main applications.

POSIX Main Wrapper. Consider this MATLAB file, `sample.m`.

```
function y = sample(varargin)
varargin{:}
y = 0;
```

You can compile `sample.m` into a POSIX main application. If you call `sample` from MATLAB, you get

```
sample hello world
ans =
hello

ans =
world
```

6-11

```
ans =
     0
```

If you compile `sample.m` and call it from the DOS shell, you get

```
C:\> sample hello world

ans =
hello

ans =
world

C:\>
```

The difference between the MATLAB and DOS/UNIX environments is the handling of the return value. In MATLAB, the return value is handled by printing its value; in the DOS/UNIX shell, the return value is handled as the return status code. When you compile a function into a POSIX main application, the return status is set to 0 if the compiled MATLAB file is executed without errors and is nonzero if there are errors.

C Library Wrapper

The `-l` option, or its equivalent `-W lib:libname`, produces a C library wrapper file. This option produces a shared library from an arbitrary set of MATLAB files. The generated header file contains a C function declaration for each of the compiled MATLAB functions. The export list contains the set of symbols that are exported from a C shared library.

Note You must generate a library wrapper file when calling any MATLAB Compiler generated code from a larger application.

C++ Library Wrapper

The `-W cpplib:libname` option produces the C++ library wrapper file. This option allows the inclusion of an arbitrary set of MATLAB files into a library.

The generated header file contains all of the entry points for all of the compiled MATLAB functions.

Note You must generate a library wrapper file when calling any MATLAB Compiler generated code from a larger application.

MCR Component Cache and CTF Archive Embedding

In this section...
"Overriding Default Behavior" on page 6-15
"For More Information" on page 6-16

CTF data is automatically embedded directly in the C/C++, `main` and `Winmain`, shared libraries and standalones by default. It is also extracted by default to a temporary folder.

Automatic embedding enables usage of MCR Component Cache features through environment variables.

These variables allow you to specify the following:

- Define the default location where you want the CTF archive to be automatically extracted
- Add diagnostic error printing options that can be used when automatically extracting the CTF, for troubleshooting purposes
- Tuning the MCR component cache size for performance reasons.

Use the following environment variables to change these settings.

Environment Variable	**Purpose**	**Notes**
MCR_CACHE_ROOT	When set to the location of where you want the CTF archive to be extracted, this variable overrides the default per-user component cache location.	Does not apply
MCR_CACHE_VERBOSE	When set to any value, this variable prints logging details about the component cache for diagnostic reasons. This can be very helpful if problems	Logging details are turned off by default (for example, when this variable has no value).

Environment Variable	Purpose	Notes
	are encountered during CTF archive extraction.	
MCR_CACHE_SIZE	When set, this variable overrides the default component cache size.	The initial limit for this variable is 32M (megabytes). This may, however, be changed after you have set the variable the first time. Edit the file .max_size, which resides in the file designated by running the mcrcachedir command, with the desired cache size limit.

You can override this automatic embedding and extraction behavior by compiling with the -C option. See "Overriding Default Behavior" on page 6-15 for more information.

Caution If you run mcc specifying conflicting wrapper and target types, the CTF will not be embedded into the generated component. For example, if you run:

```
mcc -W lib:myLib -T link:exe test.m test.c
```

the generated test.exe will not have the CTF embedded in it, as if you had specified a -C option to the command line.

Overriding Default Behavior

To extract the CTF archive in a manner prior to R2008a, alongside the compiled shared library or executable, compile using the option "-C Do Not Embed CTF Archive by Default" on page 12-26.

You can also implement this override by checking the appropriate **Option** in the Deployment Tool.

You might want to use this option to troubleshoot problems with the CTF archive, for example, as the log and diagnostic messages are much more visible.

For More Information

For more information about the CTF archive, see "Component Technology File (CTF Archive)" on page 3-8.

Explicitly Including a File for Compilation Using the %#function Pragma

In this section...
"Using feval" on page 6-17
"Using %#function" on page 6-17

Using feval

In standalone mode, the pragma

```
%#function <function_name-list>
```

informs MATLAB Compiler that the specified function(s) should be included in the compilation, whether or not the MATLAB Compiler dependency analysis detects it. Without this pragma, the MATLAB Compiler dependency analysis will not be able to locate and compile all MATLAB files used in your application. This pragma adds the top-level function as well as all the local functions in the file to the compilation.

You cannot use the %#function pragma to refer to functions that are not available in MATLAB code.

Using %#function

A good coding technique involves using %#function in your code wherever you use feval statements. This example shows how to use this technique to help MATLAB Compiler find the appropriate files during compile time, eliminating the need to include all the files on the command line.

```
function ret = mywindow(data,filterName)
%MYWINDOW Applies the window specified on the data.
%

% Get the length of the data.
N= length(data);

% List all the possible windows.
```

```
% Note the list of functions in the following function pragma is
% on a single line of code.
%#function bartlett, barthannwin, blackman, blackmanharris,
bohmanwin, chebwin, flattopwin, gausswin, hamming, hann, kaiser,
nuttallwin, parzenwin, rectwin, tukeywin, triang

window = feval(filterName,N);
% Apply the window to the data.
ret = data.*window;
```

Use the mxArray API to Work with MATLAB Types

For full documentation on the mxArray API, see the *MATLAB C and Fortran API Reference* documentation.

For a complete description of data types used with mxArray, see *MATLAB External Interfaces* documentation.

For general information on data handling, see *MATLAB External Interfaces* documentation.

Script Files

Converting Script MATLAB Files to Function MATLAB Files

MATLAB provides two ways to package sequences of MATLAB commands:

- Function MATLAB files
- Script MATLAB files

Some things to remember about script and function MATLAB files:

- Variables used inside function MATLAB files are local to that function; you cannot access these variables from the MATLAB interpreter's workspace unless they are passed back by the function. By contrast, variables used inside script MATLAB files are shared with the caller's workspace; you can access these variables from the MATLAB interpreter command line.

- Variables that are declared as persistent in a MEX-file may not retain their values through multiple calls from MATLAB.

MATLAB Compiler can compile script MATLAB files or can compile function MATLAB files that call scripts. You can either specify an script MATLAB file explicitly on the `mcc` command line, or you can specify function MATLAB files that include scripts.

Converting a script into a function is usually fairly simple. To convert a script to a function, simply add a `function` line at the top of the MATLAB file.

Running this script MATLAB file from a MATLAB session creates variables `m` and `t` in your MATLAB workspace browser.

If desired, convert this script MATLAB file into a function MATLAB file by simply adding a `function` header line.

```
function houdini(sz)
m = magic(sz); % Assign magic square to m.
t = m .^ 3;    % Cube each element of m.
disp(t)        % Display the value of t.
```

MATLAB Compiler can now compile `houdini.m`. However, because this makes `houdini` a function, running the function no longer creates variables `m` and `t` in the MATLAB workspace browser. If it is important to have `m` and `t` accessible from the MATLAB workspace browser, you can change the beginning of the function to

```
function [m,t] = houdini(sz)
```

The function now returns the values of `m` and `t` to its caller.

Including Script Files in Deployed Applications

Compiled applications consist of two layers of MATLAB files. The top layer is the interface layer and consists of those functions that are directly accessible from C or C++.

In standalone applications, the interface layer consists of only the main MATLAB file. In libraries, the interface layer consists of the MATLAB files specified on the `mcc` command line.

The second layer of MATLAB files in compiled applications includes those MATLAB files that are called by the functions in the top layer. You can include scripts in the second layer, but not in the top layer.

For example, you can produce an application from the `houdini.m` script MATLAB file by writing a new MATLAB function that calls the script, rather than converting the script into a function.

```
function houdini_fcn
 houdini;
```

To produce the `houdini_fcn`, which will call the `houdini.m` script MATLAB file, use

```
mcc -m houdini_fcn
```

Compiler Tips

In this section...

Calling a Function from the Command Line

You can make a MATLAB function into a standalone that is directly callable from the system command line. All the arguments passed to the MATLAB function from the system command line are strings. Two techniques to work with these functions are:

- Modify the original MATLAB function to test each argument and convert the strings to numbers.

- Write a wrapper MATLAB function that does this test and then calls the original MATLAB function.

  For example:

  ```
  function x=foo(a, b)
   if (ischar(a)), a = str2num(a), end;
   if (ischar(b)), b = str2num(b), end;

  % The rest of your MATLAB code here...
  ```

You only do this if your function expects numeric input. If your function expects strings, there is nothing to do because that's the default from the command line.

Using winopen in a Deployed Application

winopen is a function that depends closely on a computer's underlying file system. You need to specify a path to the file you want to open, either absolute or relative.

When using winopen in deployed mode:

1 Verify that the file being passed to the command exists on the MATLAB path.

2 Use the which command to return an absolute path to the file.

3 Pass the path to winopen.

Using MAT-Files in Deployed Applications

To use a MAT-file in a deployed application, use the MATLAB Compiler -a option to include the file in the CTF archive. For more information on the -a option, see "-a Add to Archive" on page 12-21.

Compiling a GUI That Contains an ActiveX Control

When you save a GUI that contains ActiveX components, GUIDE creates a file in the current folder for each such component. The file name consists of the name of the GUI followed by an underscore (_) and activex*n*, where *n* is a sequence number. For example, if the GUI is named ActiveXcontrol then the file name would be ActiveXcontrol_activex1. The file name does not have an extension.

If you use MATLAB Compiler mcc command to compile a GUIDE-created GUI that contains an ActiveX component, you must use the -a option to add the ActiveX control files that GUIDE saved in the current folder to the CTF archive. Your command should be similar to

```
mcc -m mygui -a mygui_activex1
```

where `mygui_activex1` is the name of the file. If you have more than one such file, use a separate `-a` option for each file.

Debugging MATLAB Compiler Generated Executables

As of MATLAB Compiler 4, it is no longer possible to debug your entire program using a C/C++ debugger; most of the application is MATLAB code, which can only be debugged in MATLAB. Instead, run your code in MATLAB and verify that it produces the desired results. Then you can compile it. The compiled code will produce the same results.

Deploying Applications That Call the Java Native Libraries

If your application interacts with Java, you need to specify the search path for native method libraries by editing `librarypath.txt` and deploying it.

1 Copy `librarypath.txt` from *matlabroot*/toolbox/local/librarypath.txt.

2 Place `librarypath.txt` in <mcr_root>/<ver>/toolbox/local.

   <mcr_root> refers to the complete path where the MCR library archive files are installed on your machine.

3 Edit `librarypath.txt` by adding the folder that contains the native library that your application's Java code needs to load.

Locating .fig Files in Deployed Applications

MATLAB Compiler locates `.fig` files automatically when there is a MATLAB file with the same name as the `.fig` file in the same folder. If the `.fig` file does not follow this rule, it must be added with the `-a` option.

Terminating Figures by Force In a Console Application

The purpose of `mclWaitForFiguresToDie` is to block execution of a calling program as long as figures created in encapsulated MATLAB code are displayed. `mclWaitForFiguresToDie` takes no arguments. Your application

can call `mclWaitForFiguresToDie` any time during execution. Typically you use `mclWaitForFiguresToDie` when:

- There are one or more figures you want to remain open.

- The function that displays the graphics requires user input before continuing.

- The function that calls the figures was called from `main()` in a console program.

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

Both MATLAB Builder NE and MATLAB Builder JA use `mclWaitForFiguresToDie` through the use of wrapper methods. See "Blocking Execution of a Console Application That Creates Figures" in the MATLAB Builder NE User's Guide and "Blocking Execution of a Console Application that Creates Figures" in the MATLAB Builder JA User's Guide for more details and code fragment examples.

Caution   Use caution when calling the `mclWaitForFiguresToDie` function. Calling this function from an interactive program like Excel can hang the application. This function should be called *only* from console-based programs.

Using `mclWaitForFiguresToDie` with Standalone Applications
Standalone applications will terminate when all of the following are true:

- The deployed main MATLAB function has returned

- There are no open visible figures for at least four seconds

Passing Arguments to and from a Standalone Application
To pass input arguments to a MATLAB Compiler generated standalone application, you pass them just as you would to any console-based application. For example, to pass a file called `helpfile` to the compiled function called `filename`, use

```
filename helpfile
```

To pass numbers or letters (e.g., 1, 2, and 3), use

```
filename 1 2 3
```

Do not separate the arguments with commas.

To pass matrices as input, use

```
filename "[1 2 3]" "[4 5 6]"
```

You have to use the double quotes around the input arguments if there is a space in it. The calling syntax is similar to the `dos` command. For more information, see the MATLAB `dos` command.

The things you should keep in mind for your MATLAB file before you compile are:

- The input arguments you pass to your application from a system prompt are considered as string input. If, in your MATLAB code before compilation, you are expecting the data in different format, say double, you will need to convert the string input to the required format. For example, you can use `str2num` to convert the string input to numerical data. You can determine at run time whether or not to do this by using the `isdeployed` function. If your MATLAB file expects numeric inputs in MATLAB, the code can check whether it is being run as a standalone application. For example:

```
function myfun (n1, n2)
if (isdeployed)
 n1 = str2num(n1);
 n2 = str2num(n2);
end
```

- You cannot return back values from your standalone application to the user. The only way to return values from compiled code is to either display it on the screen or store it in a file. To display your data on the screen, you either need to unsuppress (do not use semicolons) the commands whose results yield data you want to return to the screen or, use the `disp` command to

display the value. You can then redirect these outputs to other applications using output redirection (> operator) or pipes (only on UNIX systems).

Passing Arguments to a Double-Clickable Application

On Windows, if you want to run the standalone application by double-clicking it, you can create a batch file that calls this standalone application with the specified input arguments. Here is an example of the batch file:

```
rem main.bat file that calls sub.exe with input parameters
sub "[1 2 3]" "[4 5 6]"
@echo off
pause
```

The last two lines of code keep your output on the screen until you press a key. If you save this file as `main.bat`, you can run your code with the specified arguments by double-clicking the `main.bat` icon.

Using Graphical Applications in Shared Library Targets

When deploying a GUI as a shared library to a C/C++ application, use `mclWaitForFiguresToDie` to display the GUI until it is explicitly terminated.

Using the VER Function in a Compiled MATLAB Application

When you use the VER function in a compiled MATLAB application, it will perform with the same functionality as if you had called it from MATLAB. However, be aware that when using VER in a compiled MATLAB application, only version information for toolboxes which the compiled application uses will be displayed.

Standalone Applications

This chapter describes how to use MATLAB Compiler to code and build standalone applications. You can distribute standalone applications to users who do not have MATLAB software on their systems.

Introduction

Suppose you want to create an application that calculates the rank of a large magic square. One way to create this application is to code the whole application in C or C++; however, this would require writing your own magic square, rank, and singular value routines. An easier way to create this application is to write it as one or more MATLAB files, taking advantage of the power of MATLAB and its tools.

You can create MATLAB applications that take advantage of the mathematical functions of MATLAB, yet do not require that end users own MATLAB. Standalone applications are a convenient way to package the power of MATLAB and to distribute a customized application to your users.

The source code for standalone applications consists either entirely of MATLAB files or some combination of MATLAB files and MEX-files.

MATLAB Compiler takes your MATLAB files and generates a standalone executable that allows your MATLAB application to be invoked from outside of interactive MATLAB.

You can call MEX-files from MATLAB Compiler generated standalone applications. The MEX-files will then be loaded and called by the standalone code.

Deploying Standalone Applications

In this section...
"Compiling the Application" on page 7-3
"Testing the Application" on page 7-3
"Deploying the Application" on page 7-4
"Running the Application" on page 7-6

Compiling the Application

This example takes a MATLAB file, magicsquare.m, and creates a standalone application, magicsquare.

1 Copy the file magicsquare.m from

   matlabroot\extern\examples\compiler

   to your work folder.

2 To compile the MATLAB code, use

   mcc -mv magicsquare.m

   The -m option tells MATLAB Compiler (mcc) to generate a standalone application. The -v option (verbose) displays the compilation steps throughout the process and helps identify other useful information such as which third-party compiler is used and what environment variables are referenced.

   This command creates the standalone application called magicsquare and additional files. The Windows platform appends the .exe extension to the name. See the table in "Standalone Executable" on page 4-8 for the complete list of files created.

Testing the Application

These steps test your standalone application on your development machine.

Note Testing your application on your development machine is an important step to help ensure that your application is compilable. To verify that your application compiled properly, you must test all functionality that is available with the application. If you receive an error message similar to `Undefined function` or `Attempt to execute script` *script_name* `as a function`, it is likely that the application will not run properly on deployment machines. Most likely, your CTF archive is missing some necessary functions. Use `-a` to add the missing functions to the archive and recompile your code.

---

1 Update your path as described in

2 Run the standalone application from the system prompt (shell prompt on UNIX or DOS prompt on Windows) by typing the application name.

```
magicsquare.exe 4                             (On Windows)
magicsquare 4                                 (On UNIX)
magicsquare.app/Contents/MacOS/magicsquare    (On Maci64)
```

The results are:

```
ans =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Deploying the Application

You can distribute a MATLAB Compiler generated standalone application to any target machine that has the same operating system as the machine on which the application was compiled.

For example, if you want to deploy an application to a Windows machine, you must use MATLAB Compiler to build the application on a Windows machine. If you want to deploy the same application to a UNIX machine, you must use MATLAB Compiler on the same UNIX platform and completely rebuild the application. To deploy an application to multiple platforms requires MATLAB and MATLAB Compiler licenses on all the desired platforms.

Windows

Gather and package the following files and distribute them to the deployment machine.

Component	Description
MCR Installer	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform. Run the `mcrinstaller` command to obtain name of executable.
magicsquare	Application; `magicsquare.exe` for Windows

UNIX

Distribute and package your standalone application on UNIX by packaging the following files and distributing them to the deployment machine.

Component	Description
MCR Installer	MATLAB Compiler Runtime library archive; platform-dependent file that must correspond to the end user's platform. Run the `mcrinstaller` command to obtain name of the binary.
magicsquare	Application

Maci64

Distribute and package your standalone application on 64-bit Macintosh by copying, tarring, or zipping as described in the following table.

Component	Description
MCR Installer	MATLAB Compiler Runtime library archive; platform-dependent file that must correspond to the end user's platform. Run the mcrinstaller command to obtain name of the binary.
magicsquare	Application

magicsquare.app	Application bundle
	
	Assuming foo is a folder within your current folder:
	
	• Distribute by copying:
	
	```cp -R myapp.app foo```
	
	• Distribute by tarring:
	
	``tar -cvf myapp.tar myapp.app
	  cd foo
	  tar -xvf../ myapp.tar```
	
	• Distribute by zipping:
	
	``zip -ry myapp myapp.app
	  cd foo
	  unzip ..\myapp.zip```

Running the Application

These steps describe the process that end users must follow to install and run the application on their machines.

Preparing Your Machines

Install the MCR by running the mcrinstaller command to obtain name of the executable or binary. For more information on running the MCR Installer utility and modifying your system paths, see "Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)" on page 1-23.

Executing the Application

Run the `magicsquare` standalone application from the system prompt and provide a number representing the size of the desired magic square, for example, 4.

```
magicsquare 4
```

The results are displayed as:

```
ans =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Note Input arguments you pass to and from a system prompt are treated as string input and you need to consider that in your application. For more information, see "Passing Arguments to and from a Standalone Application" on page 6-26.

Note Before executing your MATLAB Compiler generated executable, set the `LD_PRELOAD` environment variable to `\lib\libgcc_s.so.1`.

Executing the Application on 64-Bit Macintosh (Maci64). For 64-bit Macintosh, you run the application through the bundle:

```
magicsquare.app/Contents/MacOS/magicsquare
```

Working with Standalone Applications and Arguments

In this section...
"Overview" on page 7-8
"Passing File Names, Numbers or Letters, Matrices, and MATLAB Variables" on page 7-8
"Running Standalone Applications that Use Arguments" on page 7-9

Overview

You usually create a standalone to simply run the application without passing or retrieving any arguments to or from it.

However, arguments can be passed to standalone applications created using MATLAB Compiler in the same way that input arguments are passed to any console-based application.

The following are example commands used to execute an application called `filename` from a DOS or Linux command prompt with different types of input arguments.

Passing File Names, Numbers or Letters, Matrices, and MATLAB Variables

To Pass....	Use This Syntax....	Notes
A file named `helpfile`	`filename helpfile`	
Numbers or letters	`filename 1 2 3 a b c`	Do *not* use commas or other separators between the numbers and letters you pass.

To Pass....	Use This Syntax....	Notes
Matrices as input	`filename "[1 2 3]" "[4 5 6]"`	Place double quotes around input arguments to denote a blank space.
MATLAB variables	`for k=1:10 cmd = ['filename ',num2str(k)]; system(cmd); end`	To pass a MATLAB variable to a program as input, you must first convert it to a string.

Running Standalone Applications that Use Arguments

You call a standalone application that uses arguments from MATLAB with any of the following commands:

- SYSTEM
- DOS
- UNIX
- !

To pass the contents of a MATLAB variable to the program as an input, the variable must first be converted to a string. For example:

Using SYSTEM, DOS, or UNIX

Specify the entire command to run the application as a string (including input arguments). For example, passing the numbers and letters 1 2 3 a b c could be executed using the SYSTEM command, as follows:

```
system('filename 1 2 3 a b c')
```

Using the ! (bang) Operator

You can also use the ! (bang) operator, from within MATLAB, as follows:

```
!filename 1 2 3 a b c
```

When you use the ! (bang) operator, the remainder of the input line is interpreted as the SYSTEM command, so it is not possible to use MATLAB variables.

Using a Windows System

To run a standalone application by double clicking on it, you create a batch file that calls the standalone application with the specified input arguments. For example:

```
rem This is main.bat file which calls
rem filename.exe with input parameters

filename "[1 2 3]" "[4 5 6]"
@echo off
pause
```

The last two lines of code in `main.bat` are added so that the window displaying your output stays open until you press a key.

Once you save this file, you run your code with the arguments specified above by double clicking on the icon for `main.bat`.

Using a MATLAB File You Plan to Deploy

When running MATLAB files that use arguments that you also plan to deploy with MATLAB Compiler, keep the following in mind:

- The input arguments you pass to your executable from a system prompt will be received as string input. Thus, if you expect the data in a different format (for example, double), you must first convert the string input to the required format in your MATLAB code. For example, you can use STR2NUM to convert the string input to numerical data.

- You cannot return values from your standalone application to the user. The only way to return values from compiled code is to either display it on the screen or store it in a file.

  In order to have data displayed back to the screen, do one of the following:

- Unsuppress the commands that yield your return data. Do not use semicolons to unsuppress.

- Use the DISP command to display the variable value, then redirect the outputs to other applications using redirects (the > operator) or pipes (||) on non-Windows systems.

Taking Input Arguments and Displaying to a Screen Using a MATLAB File. Here are two ways to use a MATLAB file to take input arguments and display data to the screen:

Method 1

```
function [x,y]=foo(z);

if ischar(z)
z=str2num(z);
else
z=z;
end
x=2*z % Omit the semicolon after calculation to display the value on the sc
y=z^2;
disp(y) %Use DISP command to display the value of a variable explicitly
```

Method 2

```
function [x,y]=foo(z);

if isdeployed
z=str2num(z);
end
x=2*z % Omit the semicolon after calculation to display the value on the sc
y=z^2;
disp(y) % Use DISP command to display the value of a variable explicitly
```

Combining Your MATLAB and C/C++ Code

To deploy an application that mixes MATLAB code with C or C++ code, simply created a shared library target and compile and link as you normally would using `deploytool` or `mcc`.

See the in this User's Guide for more information.

8

Libraries

This chapter describes how to use MATLAB Compiler to create libraries.

Introduction

You can use MATLAB Compiler to create C or C++ shared libraries (DLLs on Microsoft Windows) from your MATLAB algorithms. You can then write C or C++ programs that can call the MATLAB functions in the shared library, much like calling the functions from the MATLAB command line.

Addressing mwArrays Above the 2 GB Limit

In R2007b, you had to define `MX_COMPAT_32_OFF` in the `mbuild` step to address `MWArrays` above the 2 GB limit on 64-bit architectures. If you did not define `MX_COMPAT_32_OFF`, the compile time variable `MX_COMPAT_32` was defined for you, limiting you to using smaller arrays on all architectures.

In R2008a, the default definition of `MX_COMPAT_32` was removed, and large array support is now the default for both C and C++ code. This default may, in some cases, cause compiler warnings and errors. You can define `MX_COMPAT_32` in your `mbuild` step to return to the previously default behavior.

Code compiled with `MX_COMPAT_32` is *not* 64-bit aware. In addition, `MX_COMPAT_32` controls the behavior of some type definitions. For instance, when `MX_COMPAT_32` is defined, `mwSize` and `mwIndex` are defined to `ints`. When `MX_COMPAT_32` is not defined, `mwSize` and `mwIndex` are defined to `size_t`. This can lead to compiler warnings and errors with respect to signed and unsigned mismatches.

In R2008b, all support for MX_COMPAT_32 was removed.

See Appendix D, "C++ Utility Library Reference", for detailed changes to `mwArray` classes and method signatures.

Integrate C Shared Libraries

C Shared Library Wrapper

The C library wrapper option allows you to create a shared library from an arbitrary set of MATLAB files on both Microsoft Windows and UNIX operating systems. MATLAB Compiler generates a wrapper file, a header file, and an export list. The header file contains all of the entry points for all of the compiled MATLAB functions. The export list contains the set of symbols that are exported from a C shared library.

Note Even if you are not producing a shared library, you must use `-W lib` or `-W cpplib` when including any MATLAB Compiler generated code into a larger application.

C Shared Library Example

This example takes several MATLAB files and creates a C shared library. It also includes a standalone driver application to call the shared library.

Building the Shared Library

1 Copy the following files from *matlabroot*\extern\examples\compiler to your work directory:

```
matlabroot\extern\examples\compiler\addmatrix.m
matlabroot\extern\examples\compiler\multiplymatrix.m
matlabroot\extern\examples\compiler\eigmatrix.m
matlabroot\extern\examples\compiler\matrixdriver.c
```

> **Note** `matrixdriver.c` contains the standalone application's `main` function.

2 To create the shared library, enter the following command on a single line:

```
mcc -B csharedlib:libmatrix addmatrix.m multiplymatrix.m
eigmatrix.m -v
```

The `-B csharedlib` option is a bundle option that expands into

```
-W lib:<libname> -T link:lib
```

The `-W lib:<libname>` option tells MATLAB Compiler to generate a function wrapper for a shared library and call it `libname`. The `-T link:lib` option specifies the target output as a shared library. Note the directory where the product puts the shared library because you will need it later on.

Writing a Driver Application for a Shared Library

> **Note** You must call `mclInitializeApplication` once at the beginning of your driver application, before calling any other MathWorks functions (for example, before accessing an `MWArray`). See "Calling a Shared Library" on page 8-12 for complete details on using a MATLAB Compiler generated library in your application.
>
> You can use your operating system's `loadlibrary` (the Windows `loadlibrary` function, for example) to call a MATLAB Compiler shared library function as long as you first call the initialization and termination functions `mclInitializeApplication()` and `mclTerminateApplication()`.

All programs that call MATLAB Compiler generated shared libraries have roughly the same structure:

1 Declare variables and process/validate input arguments.

2 Call `mclInitializeApplication`, and test for success. This function sets up the global MCR state and enables the construction of MCR instances.

> **Caution** Avoid issuing `cd` commands from the driver application prior to calling `mclInitializeApplication`. Failure to do so can cause a failure in MCR initialization.

3 Call, once for each library, `<libraryname>Initialize`, to create the MCR instance required by the library.

4 Invoke functions in the library, and process the results. (This is the main body of the program.)

> **Note** If your driver application displays MATLAB figure windows, you should include a call to `mclWaitForFiguresToDie(NULL)` before calling the `Terminate` functions and `mclTerminateApplication` in the following two steps.

5 Call, once for each library, `<lib>Terminate`, to destroy the associated MCR.

> **Caution** `<lib>Terminate` will bring down enough of the MCR address space that the same library (or any other library) cannot be initialized. Issuing a `<lib>Initialize` call after a `<lib>Terminate` call causes unpredictable results. Instead, use the following structure:
>
> ```
> ...code...
> mclInitializeApplication();
> lib1Initialize();
> lib2Initialize();
>
> lib1Terminate();
> lib2Terminate();
> mclTerminateApplication();
> ...code...
> ```

6 Call `mclTerminateApplication` to free resources associated with the global MCR state.

7 Clean up variables, close files, etc., and exit.

This example uses `matrixdriver.c` as the driver application.

How the `mclmcrrt` Proxy Layer Handles Loading of Libraries. All application and software components generated by MATLAB Compiler and the associated builder products need to link against only one MathWorks library, `mclmcrrt`. This library provides a proxy API for all the public functions in MATLAB libraries used for matrix operations, MAT-file access, utility and memory management, and application runtime. The `mclmcrrt` library lies between deployed components and these other version-dependent libraries, providing the following functionality:

- Ensures that multiple versions of the MATLAB Compiler Runtime can coexist
- Provides a layer of indirection
- Ensures applications are thread-safe
- Loads the dependent (re-exported) libraries dynamically

The relationship between `mclmcrrt` and other MATLAB libraries is shown in the following figure.
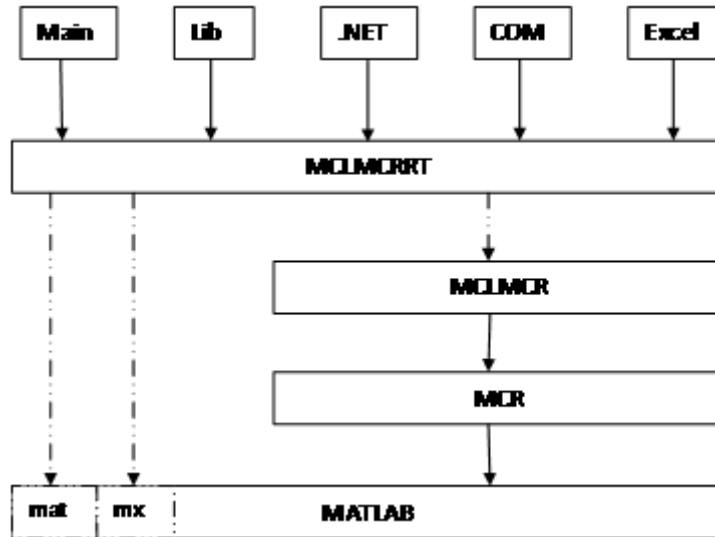
The MCLMCRRT Proxy Layer

In the figure, solid arrows designate static linking and dotted arrows designate dynamic linking. The figure illustrates how the mclmcrrt library layer sits above the mclmcr and mcr libraries. The mclmcr library contains the run-time functionality of the deployed components. The mcr module ensures each deployed component runs in its own context at runtime. The mclmcrrt proxy layer, in addition to loading the mclmcr, also dynamically loads the MX and MAT modules, primarily for mxArray manipulation. For more information, see the MathWorks Support database and search for information on the MSVC shared library.

Caution Deployed applications must only link to the mclmcrrt proxy layer library (mclmcrrt.lib on Windows, mclmcrrt.so on Linux, and mclmcrrt.dylib on Macintosh). Do not link to the other libraries shown in the figure, such as mclmcr, libmx, and so on.

Compiling the Driver Application

To compile the driver code, `matrixdriver.c`, you use your C/C++ compiler. Execute the following `mbuild` command that corresponds to your development platform. This command uses your C/C++ compiler to compile the code.

```
mbuild matrixdriver.c libmatrix.lib    (Windows)
mbuild matrixdriver.c -L. -lmatrix -I. (UNIX)
```

Note This command assumes that the shared library and the corresponding header file created from step 2 are in the current working directory.

On UNIX, if this is not the case, replace the "`.`" (dot) following the `-L` and `-I` options with the name of the directory that contains these files, respectively.

On Windows, if this is not the case, specify the full path to `libmatrix.lib`, and use a `-I` option to specify the directory containing the header file.

This generates a standalone application, `matrixdriver.exe`, on Windows, and `matrixdriver`, on UNIX.

Difference in the Exported Function Signature. The interface to the `mlf` functions generated by MATLAB Compiler from your MATLAB file routines has changed from earlier versions of the product. The generic signature of the exported `mlf` functions is

- MATLAB functions with no return values

```
bool MW_CALL_CONV mlf<function-name>
        (<list_of_input_variables>);
```

- MATLAB functions with at least one return value

```
bool MW_CALL_CONV
     mlf<function-name>(int number_of_return_values,
<list_of_pointers_to_return_variables>,
<list_of_input_variables>);
```

Refer to the header file generated for your library for the exact signature of the exported function. For example, in the library created in the previous section, the signature of the exported addmatrix function is

```
void mlfAddmatrix(int nlhs,mxArray **a,mxArray *a1,mxArray *a2);
```

Testing the Driver Application

These steps test your standalone driver application and shared library on your development machine.

Note Testing your application on your development machine is an important step to help ensure that your application is compilable. To verify that your application compiled properly, you must test all functionality that is available with the application. If you receive an error message similar to `Undefined function` or `Attempt to execute script` *script_name* `as a function`, it is likely that the application will not run properly on deployment machines. Most likely, your CTF archive is missing some necessary functions. Use `-a` to add the missing functions to the archive and recompile your code.

---

1 To run the standalone application, add the directory containing the shared library that was created in step 2 in "Building the Shared Library" on page 8-4 to your dynamic library path.

2 Update the path for your platform by following the instructions in .

3 Run the driver application from the prompt (DOS prompt on Windows, shell prompt on UNIX) by typing the application name.

```
matrixdriver.exe                              (On Windows)
matrixdriver                             (On UNIX)
matrixdriver.app/Contents/MacOS/matrixdriver (On Maci64)
```

The results are displayed as

```
The value of added matrix is:
2.00  8.00  14.00
4.00  10.00  16.00
6.00  12.00  18.00
```

```
The value of the multiplied matrix is:
30.00  66.00  102.00
36.00  81.00  126.00
42.00  96.00  150.00

The eigenvalues of the first matrix are:
16.12  -1.12  -0.00
```

Creating Shared Libraries from C with mbuild

mbuild can also create shared libraries from C source code. If a file with the extension .exports is passed to mbuild, a shared library is built. The .exports file must be a text file, with each line containing either an exported symbol name, or starting with a # or * in the first column (in which case it is treated as a comment line). If multiple .exports files are specified, all symbol names in all specified .exports files are exported.

Deploying Standalone Applications That Call MATLAB Compiler Based Shared Libraries

Gather and package the following files and distribute them to the deployment machine.

Component	Description
MCR Installer	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform. Run the mcrinstaller command to obtain name of executable.
matrixdriver	Application; matrixdriver.exe for Windows matrixdriver.app for Maci64 (bundle directory structure must be deployed)
libmatrix	Shared library; extension varies by platform. Extensions are: • Windows — .dll • Linux, Linux x86-64 — .so • Mac OS X — .dylib

Note You can distribute a MATLAB Compiler generated standalone application to any target machine that has the same operating system as the machine on which the application was compiled. If you want to deploy the same application to a different platform, you must use MATLAB Compiler on the different platform and completely rebuild the application.

Deploying Shared Libraries to Be Used with Other Projects

To distribute the shared library for use with an external application, you need to distribute the following.

Component	Description
MCR Installer	(Windows) Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform. Run the `mcrinstaller` command to obtain name of executable.
libmatrix	Shared library; extension varies by platform, for example, `DLL` on Windows
libmatrix.h	Library header file

Calling a Shared Library

At runtime, there is an MCR instance associated with each individual shared library. Consequently, if an application links against two MATLAB Compiler generated shared libraries, there will be two MCR instances created at runtime.

You can control the behavior of each MCR instance by using MCR options. The two classes of MCR options are global and local. Global MCR options are identical for each MCR instance in an application. Local MCR options may differ for MCR instances.

To use a shared library, you must use these functions:

• `mclInitializeApplication`

- `mclTerminateApplication`

Initializing and Terminating Your Application with mclInitializeApplication and mclTerminateApplication

`mclInitializeApplication` allows you to set the global MCR options. They apply equally to all MCR instances. You must set these options before creating your first MCR instance.

These functions are necessary because some MCR options such as whether or not to start Java, whether or not to use the MATLAB JIT feature, and so on, are set when the first MCR instance starts and cannot be changed by subsequent instances of the MCR.

Caution You must call `mclInitializeApplication` once at the beginning of your driver application. You must make this call before calling any other MathWorks functions. This also applies to shared libraries. Avoid calling `mclInitializeApplication` multiple times in an application as it will cause the application to hang.

After you call `mclTerminateApplication`, you may not call `mclInitializeApplication` again. No MathWorks functions may be called after `mclTerminateApplication`.

Avoid issuing `cd` commands from the driver application prior to calling `mclInitializeApplication`. Failure to do so can cause a failure in MCR initialization.

The function signatures are

```
bool mclInitializeApplication(const char **options, int count);
bool mclTerminateApplication(void);
```

mclInitializeApplication. Takes an array of strings (options) that you set (the same options that can be provided to `mcc` via the `-R` option) and a count of the number of options (the length of the option array). Returns `true` for success and `false` for failure.

mclTerminateApplication. Takes no arguments and can *only* be called
after all MCR instances have been destroyed. Returns true for success and
false for failure.

The following code example is from matrixdriver.c:

```c
int main(){

    mxArray *in1, *in2; /* Define input parameters */
    mxArray *out = NULL;/* and output parameters to pass to
                            the library functions */

    double data[] = {1,2,3,4,5,6,7,8,9};

    /* Call library initialization routine and make sure that
       the library was initialized properly */
    mclInitializeApplication(NULL,0);
    if (!libmatrixInitialize()){
        fprintf(stderr,"could not initialize the library
                        properly\n");
        return -1;
    }

    /* Create the input data */
    in1 = mxCreateDoubleMatrix(3,3,mxREAL);
    in2 = mxCreateDoubleMatrix(3,3,mxREAL);
    memcpy(mxGetPr(in1), data, 9*sizeof(double));
    memcpy(mxGetPr(in2), data, 9*sizeof(double));

    /* Call the library function */
    mlfAddmatrix(1, &out, in1, in2);
    /* Display the return value of the library function */
    printf("The value of added matrix is:\n");
    display(out);
    /* Destroy return value since this variable will be reused
       in next function call. Since we are going to reuse the
       variable, we have to set it to NULL. Refer to MATLAB
       Compiler documentation for more information on this. */
    mxDestroyArray(out); out=0;
    mlfMultiplymatrix(1, &out, in1, in2);
```

```
    printf("The value of the multiplied matrix is:\n");
    display(out);
    mxDestroyArray(out); out=0;
    mlfEigmatrix(1, &out, in1);
    printf("The Eigen value of the first matrix is:\n");
    display(out);
    mxDestroyArray(out); out=0;

    /* Call the library termination routine */
    libmatrixTerminate();

    /* Free the memory created */
    mxDestroyArray(in1); in1=0;
    mxDestroyArray(in2); in2 = 0;
    mclTerminateApplication();
    return 0;
}
```

Caution `mclInitializeApplication` can only be called *once* per application. Calling it a second time generates an error, and will cause the function to return `false`. This function must be called before calling any C MEX function or MAT-file API function.

Using a Shared Library

To use a MATLAB Compiler generated shared library in your application, you must perform the following steps:

1 Include the generated header file for each library in your application. Each MATLAB Compiler generated shared library has an associated header file named *libname*.h, where *libname* is the library's name that was passed in on the command line when the library was compiled.

2 Initialize the MATLAB libraries by calling the `mclInitializeApplication` API function. You must call this function once per application, and it must be called before calling any other MATLAB API functions, such as C-MEX functions or C MAT-file functions. `mclInitializeApplication` must be called before calling any functions in a MATLAB Compiler generated

shared library. You may optionally pass in application-level options to this function. `mclInitializeApplication` returns a Boolean status code. A return value of `true` indicates successful initialization, and `false` indicates failure.

3 For each MATLAB Compiler generated shared library that you include in your application, call the library's initialization function. This function performs several library-local initializations, such as unpacking the CTF archive, and starting an MCR instance with the necessary information to execute the code in that archive. The library initialization function will be named *libname*`Initialize()`, where *libname* is the library's name that was passed in on the command line when the library was compiled. This function returns a Boolean status code. A return value of `true` indicates successful initialization, and `false` indicates failure.

Note On Windows, if you want to have your shared library call a MATLAB shared library (as generated by MATLAB Compiler), the MATLAB library initialization function (e.g., `<libname>Initialize`, `<libname>Terminate`, `mclInitialize`, `mclTerminate`) cannot be called from your shared library during the `DllMain(DLL_ATTACH_PROCESS)` call. This applies whether the intermediate shared library is implicitly or explicitly loaded. You must place the call somewhere after `DllMain()`.

---

4 Call the exported functions of each library as needed. Use the C MEX API to process input and output arguments for these functions.

5 When your application no longer needs a given library, call the library's termination function. This function frees the resources associated with its MCR instance. The library termination function will be named `<libname>Terminate()`, where `<libname>` is the library's name that was passed in on the command line when the library was compiled. Once a library has been terminated, that library's exported functions should not be called again in the application.

6 When your application no longer needs to call any MATLAB Compiler generated libraries, call the `mclTerminateApplication` API function. This function frees application-level resources used by the MCR. Once you call

this function, no further calls can be made to MATLAB Compiler generated libraries in the application.

Restrictions When using MATLAB Function loadlibrary

You can not use the MATLAB function `loadlibrary` inside of MATLAB to load a C shared library built with MATLAB Compiler.

For more information about using `loadlibrary`, see "Load MATLAB Libraries using loadlibrary" on page 3-19.

Using C Shared Libraries On a Mac OS X System

For information on using C shared libraries on a Macintosh system, see "Using C/C++ Shared Libraries on a Mac OS X System" on page 8-21.

Integrate C++ Shared Libraries

In this section...

C++ Shared Library Wrapper

The C++ library wrapper option allows you to create a shared library from an arbitrary set of MATLAB files. MATLAB Compiler generates a wrapper file and a header file. The header file contains all of the entry points for all of the compiled MATLAB functions.

Note Even if you are not producing a shared library, you must use -W lib or -W cpplib when including any MATLAB Compiler generated code into a larger application. For more information, refer to "Combining Your MATLAB and C/C++ Code" on page 7-12.

C++ Shared Library Example

This example rewrites the previous C shared library example using C++. The procedure for creating a C++ shared library from MATLAB files is identical to the procedure for creating a C shared library, except you use the cpplib wrapper. Enter the following command on a single line:

```
mcc -W cpplib:libmatrixp -T link:lib addmatrix.m multiplymatrix.m eigmatrix.m -v
```

The -W cpplib:<libname> option tells MATLAB Compiler to generate a function wrapper for a shared library and call it <libname>. The -T link:lib option specifies the target output as a shared library. Note the directory where the product puts the shared library because you will need it later.

Writing the Driver Application

Note Due to name mangling in C++, you must compile your driver application with the same version of your third-party compiler that you use to compile your C++ shared library.

In the C++ version of the `matrixdriver` application `matrixdriver.cpp`, arrays are represented by objects of the class `mwArray`. Every `mwArray` class object contains a pointer to a MATLAB array structure. For this reason, the attributes of an `mwArray` object are a superset of the attributes of a MATLAB array. Every MATLAB array contains information about the size and shape of the array (i.e., the number of rows, columns, and pages) and either one or two arrays of data. The first array stores the real part of the array data and the second array stores the imaginary part. For arrays with no imaginary part, the second array is not present. The data in the array is arranged in column-major, rather than row-major, order.

Caution   Avoid issuing `cd` commands from the driver application prior to calling `mclInitializeApplication`. Failure to do so can cause a failure in MCR initialization.

Caution   to Mac users: when running the `matrixdriver` example, invoke `mclInitializeApplication` prior to `mclRunMain`.

For information about how the compiler uses a proxy layer for the libraries that an application must link, see "How the `mclmcrrt` Proxy Layer Handles Loading of Libraries" on page 8-7.

Compiling the Driver Application

To compile the `matrixdriver.cpp` driver code, you use your C++ compiler. By executing the following `mbuild` command that corresponds to your development platform, you will use your C++ compiler to compile the code.

```
mbuild matrixdriver.cpp libmatrixp.lib            (Windows)
```

```
mbuild matrixdriver.cpp -L. -lmatrixp -I.          (UNIX)
```

Note This command assumes that the shared library and the corresponding header file are in the current working directory.

On Windows, if this is not the case, specify the full path to `libmatrixp.lib`, and use a `-I` option to specify the directory containing the header file.

On UNIX, if this is not the case, replace the "`.`" (dot) following the `-L` and `-I` options with the name of the directory that contains these files, respectively.

Incorporating a C++ Shared Library into an Application

To incorporate a C++ shared library into your application, you will, in general, follow the steps in "Using a Shared Library" on page 8-15. There are two main differences to note when using a C++ shared library:

- Interface functions use the `mwArray` type to pass arguments, rather than the `mxArray` type used with C shared libraries.
- C++ exceptions are used to report errors to the caller. Therefore, all calls must be wrapped in a `try-catch` block.

Exported Function Signature

The C++ shared library target generates two sets of interfaces for each MATLAB function. The first set of exported interfaces is identical to the `mlx` signatures that are generated in C shared libraries. The second set of interfaces is the C++ function interfaces. The generic signature of the exported C++ functions is as follows:

MATLAB Functions with No Return Values.

```
bool MW_CALL_CONV <function-name>(<mwArray_lists>);
```

MATLAB Functions with at Least One Return Value.

```
bool MW_CALL_CONV <function-name>(int <number_of_return_values>,
    <mxArray_pointers>, <mwArray_lists>);
```

In this case, *mwArray_lists* represents a comma-separated list of type const mwArray& and *mxArray_pointers* represents a comma-separated list of pointers of type mwArray&. For example, in the libmatrix library, the C++ interfaces to the addmatrix MATLAB function is generated as:

```
void addmatrix(int nargout, mwArray& a , const mwArray& a1,
               const mwArray& a2);
```

Error Handling

C++ interface functions handle errors during execution by throwing a C++ exception. Use the mwException class for this purpose. Your application can catch mwExceptions and query the what() method to get the error message. To correctly handle errors when calling the C++ interface functions, wrap each call inside a try-catch block.

```
 try
{
  ...
  (call function)
  ...
}
catch (const mwException& e)
{
  ...
  (handle error)
  ...
}
```

The matrixdriver.cpp application illustrates the typical way to handle errors when calling the C++ interface functions.

Using C/C++ Shared Libraries on a Mac OS X System

To use a MATLAB Compiler generated library on a Mac OS X system, a separate thread needs to be created.

The initialization of the shared library and subsequent calls to that library's functions is performed by this thread. The function mclRunMain, provided by MATLAB Compiler, takes care of the thread creation process.

The main thread of the application is the thread that calls your driver program's `main()` function. The body of your `main()` function should call the `mclRunMain` function, passing to it the address of another function. This function should contain the library initialization routines and necessary calls to the shared library generated by MATLAB Compiler.

The `matrixdriver.c` example illustrates this procedure. This example rewrites the C shared library example from this chapter for use on Mac OS X. Follow the same procedure as in "C Shared Library Example" on page 8-4 to build and run this application.

The Mac version of the `matrixdriver` application differs from the version on other platforms. The `run_main()` function performs the basic tasks of initialization, calling the library's functions, and termination. Compare this function with the `matrixdriver main()` function on other platforms, listed in the earlier example.

Working with C++ Shared Libraries and Sparse Arrays

The MATLAB Compiler API includes static factory methods for working with sparse arrays.

For a complete list of the methods, see "Static Factory Methods for Sparse Arrays" on page D-33.

Call MATLAB Compiler API Functions (mcl*) from C/C++ Code

In this section...
"Functions in the Shared Library" on page 8-23
"Type of Application" on page 8-23
"Structure of Programs That Call Shared Libraries" on page 8-25
"Library Initialization and Termination Functions" on page 8-26
"Print and Error Handling Functions" on page 8-27
"Functions Generated from MATLAB Files" on page 8-29
"Retrieving MCR State Information While Using Shared Libraries" on page 8-34

Functions in the Shared Library

A shared library generated by MATLAB Compiler contains at least seven functions. There are three generated functions to manage library initialization and termination, one each for printed output and error messages, and two generated functions for each MATLAB file compiled into the library.

To generate the functions described in this section, first copy sierpinski.m, main_for_lib.c, main_for_lib.h, and triangle.c from *matlabroot*\extern\examples\compiler into your directory, and then execute the appropriate MATLAB Compiler command.

Type of Application

For a C Application on Windows

```
mcc -W lib:libtriangle -T link:lib sierpinski.m
mbuild triangle.c main_for_lib.c libtriangle.lib
```

For a C Application on UNIX

```
mcc -W lib:libtriangle -T link:lib sierpinski.m
mbuild triangle.c main_for_lib.c -L. -ltriangle -I.
```

For a C++ Application on Windows

```
mcc -W cpplib:libtrianglep -T link:lib sierpinski.m
mbuild triangle.cpp main_for_lib.c libtrianglep.lib
```

For a C++ Application on UNIX

```
mcc -W cpplib:libtriangle -T link:lib sierpinski.m
mbuild triangle.cpp main_for_lib.c -L. -ltriangle -I.
```

These commands create a main program named `triangle`, and a shared library named `libtriangle`. The library exports a single function that uses a simple iterative algorithm (contained in `sierpinski.m`) to generate the fractal known as Sierpinski's Triangle. The main program in `triangle.c` or `triangle.cpp` can optionally take a single numeric argument, which, if present, specifies the number of points used to generate the fractal. For example, `triangle 8000` generates a diagram with 8,000 points.
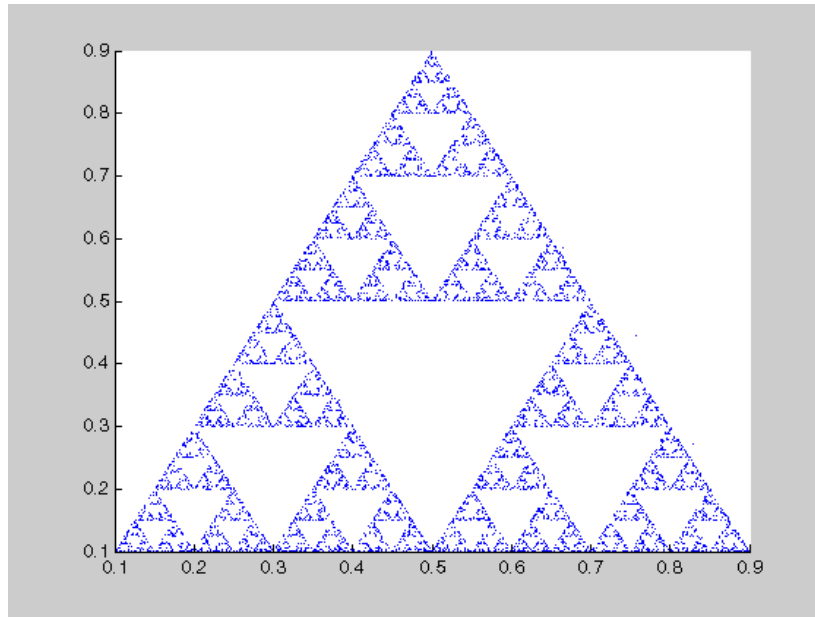
In this example, MATLAB Compiler places all of the generated functions into the generated file libtriangle.c or libtriangle.cpp.

Structure of Programs That Call Shared Libraries

All programs that call MATLAB Compiler generated shared libraries have roughly the same structure:

1 Declare variables and process/validate input arguments.

2 Call mclInitializeApplication, and test for success. This function sets up the global MCR state and enables the construction of MCR instances.

3 Call, once for each library, <libraryname>Initialize, to create the MCR instance required by the library.

4 Invoke functions in the library, and process the results. (This is the main body of the program.)

5 Call, once for each library, `<libraryname>Terminate`, to destroy the associated MCR.

6 Call `mclTerminateApplication` to free resources associated with the global MCR state.

7 Clean up variables, close files, etc., and exit.

To see these steps in an actual example, review the main program in this example, `triangle.c`.

Library Initialization and Termination Functions

The library initialization and termination functions create and destroy, respectively, the MCR instance required by the shared library. You must call the initialization function before you invoke any of the other functions in the shared library, and you should call the termination function after you are finished making calls into the shared library (or you risk leaking memory).

There are two forms of the initialization function and one type of termination function. The simpler of the two initialization functions takes no arguments; most likely this is the version your application will call. In this example, this form of the initialization function is called `libtriangleInitialize`.

```
bool libtriangleInitialize(void)
```

This function creates an MCR instance using the default print and error handlers, and other information generated during the compilation process.

However, if you want more control over how printed output and error messages are handled, you may call the second form of the function, which takes two arguments.

```
bool libtriangleInitializeWithHandlers(
    mclOutputHandlerFcn error_handler,
    mclOutputHandlerFcn print_handler
)
```

By calling this function, you can provide your own versions of the print and error handling routines called by the MCR. Each of these routines has the same signature (for complete details, see "Print and Error Handling

Functions" on page 8-27). By overriding the defaults, you can control how output is displayed and, for example, whether or not it goes into a log file.

Note Before calling either form of the library initialization routine, you must first call `mclInitializeApplication` to set up the global MCR state. See "Calling a Shared Library" on page 8-12 for more information.

On Microsoft Windows platforms, MATLAB Compiler generates an additional initialization function, the standard Microsoft DLL initialization function `DllMain`.

```
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason,
                    void *pv)
```

The generated `DllMain` performs a very important service; it locates the directory in which the shared library is stored on disk. This information is used to find the CTF archive, without which the application will not run. If you modify the generated `DllMain` (not recommended), make sure you preserve this part of its functionality.

Library termination is simple.

```
void libtriangleTerminate(void)
```

Call this function (once for each library) before calling `mclTerminateApplication`.

Print and Error Handling Functions

By default, MATLAB Compiler generated applications and shared libraries send printed output to standard output and error messages to standard error. MATLAB Compiler generates a default print handler and a default error handler that implement this policy. If you'd like to change this behavior, you must write your own error and print handlers and pass them in to the appropriate generated initialization function.

You may replace either, both, or neither of these two functions. The MCR sends all regular output through the print handler and all error output

through the error handler. Therefore, if you redefine either of these functions, the MCR will use your version of the function for all the output that falls into class for which it invokes that handler.

The default print handler takes the following form.

```
static int mclDefaultPrintHandler(const char *s)
```

The implementation is straightforward; it takes a string, prints it on standard output, and returns the number of characters printed. If you override or replace this function, your version must also take a string and return the number of characters "handled." The MCR calls the print handler when an executing MATLAB file makes a request for printed output, e.g., via the MATLAB function `disp`. The print handler does not terminate the output with a carriage return or line feed.

The default error handler has the same form as the print handler.

```
static int mclDefaultErrorHandler(const char *s)
```

However, the default implementation of the print handler is slightly different. It sends the output to the standard error output stream, but if the string does not end with carriage return, the error handler adds one. If you replace the default error handler with one of your own, you should perform this check as well, or some of the error messages printed by the MCR will not be properly formatted.

Caution   The error handler, despite its name, does not handle the actual errors, but rather the message produced after the errors have been caught and handled inside the MCR. You cannot use this function to modify the error handling behavior of the MCR -- use the `try` and `catch` statements in your MATLAB files if you want to control how a MATLAB Compiler generated application responds to an error condition.

Note If you provide alternate C++ implementations of either `mclDefaultPrintHandler` or `mclDefaultErrorHandler`, then functions must be declared `extern "C"`. For example:

```
extern "C" int myPrintHandler(const char *s);
```

Functions Generated from MATLAB Files

For each MATLAB file specified on the MATLAB Compiler command line, the product generates two functions, the `mlx` function and the `mlf` function. Each of these generated functions performs the same action (calls your MATLAB file function). The two functions have different names and present different interfaces. The name of each function is based on the name of the first function in the MATLAB file (`sierpinski`, in this example); each function begins with a different three-letter prefix.

Note For C shared libraries, MATLAB Compiler generates the `mlx` and `mlf` functions as described in this section. For C++ shared libraries, the product generates the `mlx` function the same way it does for the C shared library. However, the product generates a modified `mlf` function with these differences:

- The `mlf` before the function name is dropped to keep compatibility with R13.

- The arguments to the function are `mwArray` instead of `mxArray`.

---

mlx Interface Function

The function that begins with the prefix `mlx` takes the same type and number of arguments as a MATLAB MEX-function. (See the External Interfaces documentation for more details on MEX-functions.) The first argument, `nlhs`, is the number of output arguments, and the second argument, `plhs`, is a pointer to an array that the function will fill with the requested number of return values. (The "`lhs`" in these argument names is short for "left-hand side" -- the output variables in a MATLAB expression are those on the

left-hand side of the assignment operator.) The third and fourth parameters
are the number of inputs and an array containing the input variables.

```
void mlxSierpinski(int nlhs, mxArray *plhs[], int nrhs,
                   mxArray *prhs[])
```

mlf Interface Function

The second of the generated functions begins with the prefix `mlf`. This
function expects its input and output arguments to be passed in as individual
variables rather than packed into arrays. If the function is capable of
producing one or more outputs, the first argument is the number of outputs
requested by the caller.

```
void mlfSierpinski(int nargout, mxArray** x, mxArray** y,
                   mxArray* iterations, mxArray* draw)
```

In both cases, the generated functions allocate memory for their return
values. If you do not delete this memory (via `mxDestroyArray`) when you are
done with the output variables, your program will leak memory.

Your program may call whichever of these functions is more convenient, as
they both invoke your MATLAB file function in an identical fashion. Most
programs will likely call the `mlf` form of the function to avoid managing the
extra arrays required by the `mlx` form. The example program in `triangle.c`
calls `mlfSierpinski`.

```
mlfSierpinski(2, &x, &y, iterations, draw);
```

In this call, the caller requests two output arguments, `x` and `y`, and provides
two inputs, `iterations` and `draw`.

If the output variables you pass in to an `mlf` function are not NULL, the `mlf`
function will attempt to free them using `mxDestroyArray`. This means that
you can reuse output variables in consecutive calls to `mlf` functions without
worrying about memory leaks. It also implies that you must pass either NULL
or a valid MATLAB array for all output variables or your program will fail
because the memory manager cannot distinguish between a non-initialized
(invalid) array pointer and a valid array. It will try to free a pointer that is
not NULL -- freeing an invalid pointer usually causes a segmentation fault
or similar fatal error.

Using varargin and varargout in a MATLAB Function Interface

If your MATLAB function interface uses `varargin` or `varargout`, you must pass them as cell arrays. For example, if you have N `varargins`, you need to create one cell array of size `1-by-N`. Similarly, `varargouts` are returned back as one cell array. The length of the `varargout` is equal to the number of return values specified in the function call minus the number of actual variables passed. As in the MATLAB software, the cell array representing `varagout` has to be the last return variable (the variable preceding the first input variable) and the cell array representing `varargins` has to be the last formal parameter to the function call.

For information on creating cell arrays, refer to the C MEX function interface in the External Interfaces documentation.

For example, consider this MATLAB file interface:

```
[a,b,varargout] = myfun(x,y,z,varargin)
```

The corresponding C interface for this is

```
void mlfMyfun(int numOfRetVars, mxArray **a, mxArray **b,
             mxArray **varargout, mxArray *x, mxArray *y,
             mxArray *z, mxArray *varargin)
```

In this example, the number of elements in `varargout` is (`numOfRetVars - 2`), where `2` represents the two variables, `a` and `b`, being returned. Both `varargin` and `varargout` are single row, multiple column cell arrays.

Caution   The C++ shared library interface does not support `varargin` with zero (0) input arguments. Calling your program using an empty `mwArray` results in the compiled library receiving an empty array with `nargin = 1`. The C shared library interface allows you to call `mlfFOO(NULL)` (the compiled MATLAB code interprets this as `nargin=0`). However, calling `FOO((mwArray)NULL)` with the C++ shared library interface causes the compiled MATLAB code to see an empty array as the first input and interprets `nargin=1`.

For example, compile some MATLAB code as a C++ shared library using `varargin` as the MATLAB function's list of input arguments. Have the MATLAB code display the variable `nargin`. Call the library with function *FOO*() and it won't compile, producing this error message:

```
... 'FOO' : function does not take 0 arguments
```

Call the library as:

```
  mwArray junk;
  FOO(junk);
```

or

```
 FOO((mwArray)NULL);
```

At runtime, `nargin=1`. In MATLAB, *FOO*() is `nargin=0` and `FOO([])` is `nargin=1`.

C++ Interfaces for MATLAB Functions Using varargin and varargout.
The C++ `mlx` interface for MATLAB functions does not change even if the functions use `varargin` or `varargout`. However, the C++ function interface (the second set of functions) changes if the MATLAB function is using `varargin` or `varargout`.

For examples, view the generated code for various MATLAB function signatures that use `varargin` or `varargout`.

> **Note** For simplicity, only the relevant part of the generated C++ function signature is shown in the following examples.

function varargout = foo(varargin)

For this MATLAB function, the following C++ overloaded functions are generated:

```
No input no output:
void foo()

Only inputs:
void foo(const mwArray& varargin)

Only outputs:
void foo(int nargout, mwArray& varargout)

Most generic form that has both inputs and outputs:
void foo(int nargout, mwArray& varargout,
                      const mwArray& varargin)
```

function varargout = foo(i1, i2, varargin)

For this MATLAB function, the following C++ overloaded functions are generated:

```
Most generic form that has outputs and all the inputs
void foo(int nargout, mwArray& varargout, const
                            mwArray& i1, const
                            mwArray& i2, const
                             mwArray& varargin)

Only inputs:
void foo(const mwArray& i1,
      const mwArray& i2, const mwArray& varargin)
```

function [o1, o2, varargout] = foo(varargin)

For this MATLAB function, the following C++ overloaded functions are generated:

```
Most generic form that has all the outputs and inputs
void foo(int nargout, mwArray& o1, mwArray& o2,
                            mwArray& varargout,
                        const mwArray& varargin)


Only outputs:
void foo(int nargout, mwArray& o1, mwArray& o2,
                                mwArray& varargout)
```

function [o1, o2, varargout] = foo(i1, i2, varargin)

For this MATLAB function, the following C++ overloaded function is generated:

```
Most generic form that has all the outputs and
                                        all the inputs
void foo(int nargout, mwArray& o1, mwArray& o2,

mwArray& varargout,
        const mwArray& i1, const mwArray& i2,
                      const mwArray& varargin)
```

Retrieving MCR State Information While Using Shared Libraries

When using shared libraries (note this does not apply to standalone applications), you may call functions to retrieve specific information from MCR state. For details, see "MCR Startup Options" on page 5-31.

About Memory Management and Cleanup

In this section...
"Overview" on page 8-35
"Passing mxArrays to Shared Libraries" on page 8-35

Overview

Generated C++ code provides consistent garbage collection via the object destructors and the MCR's internal memory manager optimizes to avoid heap fragmentation.

If memory constraints are still present on your system, try preallocating arrays in MATLAB. This will reduce the number of calls to the memory manager, and the degree to which the heap fragments.

Passing mxArrays to Shared Libraries

When an `mxArray` is created in an application which uses the MCR, it is created in the managed memory space of the MCR.

Therefore, it is very important that you never create `mxArrays` (or call any other MathWorks function) before calling `mclInitializeApplication`.

It is safe to call `mxDestroyArray` when you no longer need a particular `mxArray` in your code, even when the input has been assigned to a persistent or global variable in MATLAB. MATLAB uses reference counting to ensure that when `mxDestroyArray` is called, if another reference to the underlying data still exists, the memory will not be freed. Even if the underlying memory is not freed, the `mxArray` passed to `mxDestroyArray` will no longer be valid.

For more information about `mclInitializeApplication` and `mclTerminateApplication`, see "Calling a Shared Library" on page 8-12.

For more information about `mxArray`, see "Use the mxArray API to Work with MATLAB Types" on page 6-19.

9

Troubleshooting

Introduction

MATLAB Compiler software converts your MATLAB programs into self-contained applications and software components and enables you to share them with end users who do not have MATLAB installed. MATLAB Compiler takes MATLAB applications (MATLAB files, MEX-files, and other MATLAB executable code) as input and generates redistributable standalone applications or shared libraries. The resulting applications and components are platform specific.

Another use of MATLAB Compiler is to build C or C++ shared libraries (DLLs on Windows) from a set of MATLAB files. You can then write C or C++ programs that can call the functions in these libraries. The typical workflow for building a shared library is to compile your MATLAB code on a development machine, write a C/C++ driver application, build an executable from the driver code, test the resulting executable on that machine, and deploy the executable and MCR to a test or customer machine without MATLAB.

Compiling a shared library is very similar to compiling an executable. The command line differs as shown:

```
mcc -B csharedlib:hellolib hello.m
```

or

```
mcc -B cpplib:hellolib hello.m
```

Once you have compiled a shared library, the next step is to create a driver application that initializes and terminates the shared library as well as invokes method calls. This driver application can be compiled and linked with your shared library with the mbuild command. For example:

```
mbuild helloapp.c hellolib.lib
```

or

```
mbuild helloapp.cpp hellolib.lib
```

The only header file that needs to be included in your driver application is the one generated by your mcc command (hellolib.h in the above example). See "Integrate C Shared Libraries" on page 8-4 and "Integrate C++ Shared

Libraries" on page 8-18 for examples of how to correctly access a shared library.

Common Issues

Some of the most common issues encountered when using MATLAB Compiler generated standalone executables or shared libraries are:

- **Compilation fails with an error message.** This can indicate a failure during any one of the internal steps involved in producing the final output.

- **Compilation succeeds but the application does not execute because required DLLs are not found.** All shared libraries required for your standalone executable or shared library are contained in the MATLAB Compiler Runtime (MCR). Installing the MCR is required for any of the deployment targets.

- **Compilation succeeds, and the resultant file starts to execute but then produces errors and/or generates a crash dump.**

- **The compiled program executes on the machine where it was compiled but not on other machines.**

- **The compiled program executes on some machines and not others.**

If any of these issues apply to you, search "Failure Points and Possible Solutions" on page 9-5 for common solutions.

Failure Points and Possible Solutions

In this section...

How to Use this Section

Use the following list of questions to diagnose some of the more common issues associated with using MATLAB Compiler software.

Does the Failure Occur During Compilation?

You typically compile your MATLAB code on a development machine, test the resulting executable on that machine, and deploy the executable and MATLAB Compiler Runtime (MCR) to a test or customer machine without MATLAB. The compilation process performs dependency analysis on your MATLAB code, creates an encrypted archive of your code and required toolbox code, generates wrapper code, and compiles the wrapper code into an executable. If your application fails to build an executable, the following questions may help you isolate the problem.

Is your selected compiler supported by MATLAB Compiler?

See the current list of supported compilers at
http://www.mathworks.com/support/compilers/current_release/.

Are error messages produced at compile time?

See error messages in "MATLAB® Compiler™" on page 9-17.

Did you compile with the verbose flag?

Compilation can fail in MATLAB because of errors encountered by the system compiler when the generated wrapper code is compiled into an executable. Additional errors and warnings are printed when you use the verbose flag as such:

```
mcc -mv myApplication.m
```

In this example, -m tells MATLAB Compiler to create a standalone application and -v tells MATLAB Compiler and other processors to display messages about the process.

Are you compiling within or outside of MATLAB?

mcc can be invoked from the operating system command line or from the MATLAB prompt. When you run mcc inside the MATLAB environment, MATLAB will modify environment variables in its environment as necessary so mcc will run. Issues with PATH, LD_LIBRARY_PATH, or other environment variables seen at the operating system command line are often not seen at the MATLAB prompt. The environment that MATLAB uses for mcc can be listed at the MATLAB prompt. For example:

```
>>!set
```

lists the environment on Windows platforms.

```
>>!printenv
```

lists the environment on UNIX platforms. Using this path allows you to use mcc from the operating system command line.

Does a simple read/write application such as "Hello World" compile successfully?

Sometimes applications won't compile because of MEX-file issues, other toolboxes, or other dependencies. Compiling a helloworld application can determine if MATLAB Compiler is correctly set up to produce any executable. For example, try compiling:

```
function helloworld
```

```
 disp('hello world')
```

with:

```
>>mcc -mv helloworld.m
```

Have you tried to compile any of the examples in MATLAB Compiler help?

The source code for all examples is provided with MATLAB Compiler and is located in *matlabroot*\extern\examples\compiler, where *matlabroot* is the root folder of your MATLAB installation.

Did the MATLAB code compile successfully before this failure?

The three most common reasons for MATLAB code to stop compiling are:

- Upgrading to MATLAB without running mbuild -setup — Running mbuild -setup is required after any upgrade to MATLAB Compiler.

- A change in the selection of the system compiler — It is possible to inadvertently change the system compiler for versions of MATLAB that store preferences in a common folder. For example, MATLAB 7.0.1 (R14SP1) and MATLAB 7.0.4 (R14SP2) store their preferences in the same folder. Changing the system compiler in R14SP1 will also change the system compiler in R14SP2.

- An upgrade to MATLAB that didn't include an upgrade to MATLAB Compiler — The versions of MATLAB Compiler and MATLAB must be the same in order to work together. It is possible to see conflicts in installations where the MATLAB installation is local and the MATLAB Compiler installation is on a network or vice versa.

Are you receiving errors when trying to compile a standalone executable?

If you are not receiving error messages to help you debug your standalone, write an application to display the warnings or error messages (a console application).

Are you receiving errors when trying to compile a shared library?

Errors at compile time can indicate issues with either `mcc` or `mbuild`. For troubleshooting `mcc` issues, see the previous section on compile time issues. It is recommended that your driver application be compiled and linked using `mbuild`. `mbuild` can be executed with the `-v` switch to provide additional information on the compilation process. If you receive errors at this stage, ensure that you are using the correct header files and/or libraries produced by `mcc`, in your C or C++ driver. For example:

```
mcc -B csharedlib:hellolib hello.m
```

produces `hellolib.h`, which is required to be included in your C/C++ driver, and `hellolib.lib` or `hellolib.so`, which is required on the `mbuild` command line.

Is your MATLAB object failing to load?

If your MATLAB object fails to load, it is typically a result of the MCR not finding required class definitions.

When working with MATLAB objects, remember to include the following statement in your MAT file:

```
%#function class_constructor
```

Using the `%#function` pragma in this manner forces depfun to load needed class definitions, enabling the MCR to successfully load the object.

If you are compiling a driver application, are you using mbuild?

MathWorks recommends and supports using `mbuild` to compile your driver application. `mbuild` is designed and tested to correctly build driver applications. It will ensure that all MATLAB header files are found by the C/C++ compiler, and that all necessary libraries are specified and found by the linker.

Are you trying to compile your driver application using Microsoft Visual Studio or another IDE?

If using an IDE, in addition to linking to the generated export library, you need to include an additional dependency to `mclmcrrt.lib`. This library is provided for all supported third-party compilers in *matlabroot*\extern\lib*vendor-name*.

Are you importing the correct versions of import libraries?

If you have multiple versions of MATLAB installed on your machine, it is possible that an older or incompatible version of the library is referenced. Ensure that the only MATLAB library that you are linking to is `mclmcrrt.lib` and that it is referenced from the appropriate vendor folder. Do not reference libraries as `libmx` or `libut`. In addition, verify that your library path references the version of MATLAB that your shared library was built with.

Are you able to compile the matrixdriver example?

Typically, if you cannot compile the examples in the documentation, it indicates an issue with the installation of MATLAB or your system compiler. See "Integrate C Shared Libraries" on page 8-4 and "Integrate C++ Shared Libraries" on page 8-18 for these examples.

Do you get the `MATLAB:I18n:InconsistentLocale` Warning?

The warning message

```
MATLAB:I18n:InconsistentLocale - The system locale setting,
system_locale_name, is different from the user locale
setting, user_locale_name
```

indicates a mismatch between locale setting on Microsoft Windows systems. This may affect your ability to display certain characters. For information about changing the locale settings, see your operating system Help.

Does the Failure Occur When Testing Your Application?

After you have successfully compiled your application, the next step is to test it on a development machine and deploy it on a target machine. Typically the target machine does not have a MATLAB installation and requires that the MATLAB Compiler Runtime (MCR) be installed. A distribution includes all of the files that are required by your application to run, which include the executable, CTF archive and the MCR.

See "Deploying to Developers" on page 5-3 and "Deploying to End Users" on page 5-9 for information on distribution contents for specific application types and platforms.

Test the application on the development machine by running the application against the MCR shipped with MATLAB Compiler. This will verify that library dependencies are correct, that the CTF archive can be extracted and that all MATLAB code, MEX—files and support files required by the application have been included in the archive. If you encounter errors testing your application, the questions in the column to the right may help you isolate the problem.

Are you able to execute the application from MATLAB?

On the development machine, you can test your application's execution by issuing !*application-name* at the MATLAB prompt. If your application executes within MATLAB but not from outside, this can indicate an issue with the system PATH variable. For more information, see .

Does the application begin execution and result in MATLAB or other errors?

Ensure that you included all necessary files when compiling your application (see the readme.txt file generated with your compilation for more details).

Functions that are called from your main MATLAB file are automatically included by MATLAB Compiler; however, functions that are not explicitly called, for example through EVAL, need to be included at compilation using the -a switch of the mcc command. Also, any support files like .mat, .txt, or .html files need to be added to the archive with the -a switch. There is a

limitation on the functionality of MATLAB and associated toolboxes that can be compiled. Check the documentation to see that the functions used in your application's MATLAB files are valid. Check the file `mccExcludedFiles.log` on the development machine. This file lists all functions called from your application that cannot be compiled.

Does the application emit a warning like "MATLAB file may be corrupt"?

See the listing for this error message in "MATLAB® Compiler™" on page 9-17 for possible solutions.

Do you have multiple MATLAB versions installed?

Executables generated by MATLAB Compiler are designed to run in an environment where multiple versions of MATLAB are installed. Some older versions of MATLAB may not be fully compatible with this architecture.

On Windows, ensure that the *matlabroot*\runtime\win32|win64 of the version of MATLAB in which you are compiling appears ahead of *matlabroot*\runtime\win32|win64 of other versions of MATLAB installed on the PATH environment variable on your machine.

Similarly, on UNIX, ensure that the dynamic library paths (LD_LIBRARY_PATH on Linux) match. Do this by comparing the outputs of !printenv at the MATLAB prompt and printenv at the shell prompt. Using this path allows you to use mcc from the operating system command line.

If you are testing a standalone executable or shared library and driver application, did you install the MCR?

All shared libraries required for your standalone executable or shared library are contained in the MATLAB Compiler Runtime (MCR). Installing the MCR is required for any of the deployment targets.

Do you receive an error message about a missing DLL?

Error messages indicating missing DLLs such as `mclmcrrt7x.dll` or `mclmcrrt7x.so` are generally caused by incorrect installation of the MCR. It is also possible that the MCR is installed correctly, but that the

PATH,LD_LIBRARY_PATH, or DYLD_LIBRARY_PATH variables are set incorrectly. For information on installing the MCR on a deployment machine, refer to "Working with the MCR" on page 5-17.

Caution   Do not solve these problems by moving libraries or other files within the MCR folder structure. The run-time system is designed to accommodate different MCR versions operating on the same machine. The folder structure is an important part of this feature.

Are you receiving errors when trying to run the shared library application?

Calling MATLAB Compiler generated shared libraries requires correct initialization and termination in addition to library calls themselves. For information on calling shared libraries, see "Call MATLAB® Compiler™ API Functions (mcl*) from C/C++ Code" on page 8-23.

Some key points to consider to avoid errors at run time:

- Ensure that the calls to mclinitializeApplication and *libname*Initialize are successful. The first function enables construction of MCR instances. The second creates the MCR instance required by the library named *libname*. If these calls are not successful, your application will not execute.

- Do not use any mw- or mx-functions before calling mclinitializeApplication. This includes static and global variables that are initialized at program start. Referencing mw- or mx-functions before initialization results in undefined behavior.

- Do not re-initialize (call mclinitializeApplication) after terminating it with mclTerminateApplication. The mclinitializeApplication and*libname*Initialize functions should be called only once.

- Ensure that you do not have any library calls after mclTerminateApplication.

- Ensure that you are using the correct syntax to call the library and its functions.

Does the Failure Occur When Deploying the Application to End Users?

After the application is working on the test machine, failures can be isolated in end-user deployment. The end users of your application need to install the MATLAB Compiler Runtime (MCR) on their machines. The MCR includes a set of shared libraries that provides support for all features of MATLAB. If your application fails during end-user deployment, the following questions in the column to the right may help you isolate the problem.

Note There are a number of reasons why your application might not deploy to end users, after running successfully in a test environment. For a detailed list of guidelines for writing MATLAB code that can be consumed by end users, see "Write Deployable MATLAB Code" on page 3-12

Is the MCR installed?

All shared libraries required for your standalone executable or shared library are contained in the MCR. Installing the MCR is required for any of the deployment targets. See"Working with the MCR" on page 5-17 for complete information.

If running on UNIX or Mac, did you update the dynamic library path after installing the MCR?

For information on installing the MCR on a deployment machine, refer to "Working with the MCR" on page 5-17.

Do you receive an error message about a missing DLL?

Error messages indicating missing DLLs such as `mclmcrrt7x.dll` or `mclmcrrt7x.so` are generally caused by incorrect installation of the MCR. It is also possible that the MCR is installed correctly, but that the `PATH`, `LD_LIBRARY_PATH`, or `DYLD_LIBRARY_PATH` variables are set incorrectly. For information on installing the MCR on a deployment machine, refer to "Working with the MCR" on page 5-17.

Caution   Do not solve these problems by moving libraries or other files within the MCR folder structure. The run-time system is designed to accommodate different MCR versions operating on the same machine. The folder structure is an important part of this feature.

Do you have write access to the directory the application is installed in?

The first operation attempted by a compiled application is extraction of the CTF archive. If the archive is not extracted, the application cannot access the compiled MATLAB code and the application fails. If the application has write access to the installation folder, a subfolder named *application-name*_mcr is created the first time the application is run. After this subfolder is created, the application no longer needs write access for subsequent executions.

Are you executing a newer version of your application?

When deploying a newer version of an executable, both the executable needs to be redeployed, since it also contains the embedded CTF file. The CTF file is keyed to a specific compilation session. Every time an application is recompiled, a new, matched CTF file is created. As above, write access is required to expand the new CTF file. Deleting the existing *application-name*_mcr folder and running the new executable will verify that the application can expand the new CTF file.

Troubleshooting mbuild

This section identifies some of the more common problems that might occur when configuring mbuild to create standalone applications.

Options File Not Writable. When you run mbuild -setup, mbuild makes a copy of the appropriate options file and writes some information to it. If the options file is not writable, you are asked if you want to overwrite the existing options file. If you choose to do so, the existing options file is copied to a new location and a new options file is created.

Directory or File Not Writeable. If a destination folder or file is not writable, ensure that the permissions are properly set. In certain cases, make sure that the file is not in use.

mbuild Generates Errors. If you run mbuild filename and get errors, it may be because you are not using the proper options file. Run mbuild -setup to ensure proper compiler and linker settings.

Compiler and/or Linker Not Found. On Windows, if you get errors such as unrecognized command or file not found, make sure the command-line tools are installed and the path and other environment variables are set correctly in the options file. For Microsoft Visual Studio®, for example, make sure to run vcvars32.bat (MSVC 6.x and earlier) or vsvars32.bat (MSVC 8.x and later).

mbuild Not a Recognized Command. If mbuild is not recognized, verify that *matlabroot*\bin is in your path. On UNIX, it may be necessary to rehash.

mbuild Works from the Shell But Not from MATLAB (UNIX). If the command

```
mcc -m hello
```

works from the UNIX command prompt but not from the MATLAB prompt, you may have a problem with your .cshrc file. When MATLAB launches a new C shell to perform compilations, it executes the .cshrc script. If this script causes unexpected changes to the PATH environment variable, an error

may occur. You can test this before starting MATLAB by performing the following:

```
setenv SHELL /bin/sh
```

If this works correctly, then you should check your .cshrc file for problems setting the PATH environment variable.

Cannot Locate Your Compiler (Windows). If mbuild has difficulty locating your installed compilers, it is useful to know how it finds compilers. mbuild automatically detects your installed compilers by first searching for locations specified in the following environment variables:

• MSVCDIR for Microsoft Visual C++, Version 6.0 or 8.0

Next, mbuild searches the Windows registry for compiler entries.

Internal Error when Using mbuild -setup (Windows). Some antivirus software packages may conflict with the mbuild -setup process. If you get an error message during mbuild -setup of the following form

```
mex.bat: internal error in sub get_compiler_info(): don't
recognize <string>
```

then you need to disable your antivirus software temporarily and rerun mbuild -setup. After you have successfully run the setup option, you can re-enable your antivirus software.

Verification of mbuild Fails. If none of the previous solutions addresses your difficulty with mbuild, contact Technical Support at MathWorks at http://www.mathworks.com/contact_TS.html.

MATLAB Compiler

Typically, problems that occur when building standalone applications involve mbuild. However, it is possible that you may run into some difficulty with MATLAB Compiler. A good source for additional troubleshooting information for the product is the MATLAB Compiler Product Support page at the MathWorks Web site.

libmwlapack: load error: stgsy2_. This error occurs when a customer has both the R13 and the R14 version of MATLAB or MCR/MGL specified in the folder path and the R14 version fails to load because of a lapack incompatibility.

Licensing Problem. If you do not have a valid license for MATLAB Compiler , you will get an error message similar to the following when you try to access MATLAB Compiler:

```
Error: Could not check out a Compiler License:
No such feature exists.
```

If you have a licensing problem, contact MathWorks. A list of contacts at MathWorks is provided at the beginning of this document.

loadlibrary usage (MATLAB loadlibrary command). The following are common error messages encountered when attempting to compile the MATLAB loadlibrary function or run an application that uses the MATLAB loadlibrary function with MATLAB Compiler:

- Output argument 'notfound' was not assigned during call to 'loadlibrary'.

-
  ```
  Warning: Function call testloadlibcompile
  invokes inexact match
  d:\work\testLoadLibCompile_mcr\
  testLoadLibCompile\testLoadLibCompile.m.

  ??? Error using ==> loadlibrary
  Call to Perl failed. Possible error processing header file.
  Output of Perl command:
  Error using ==> perl
  ```

```
All input arguments must be valid strings.

Error in ==> testLoadLibCompile at 4
```

•
```
MATLAB:loadlibrary:cannotgeneratemfile
There was an error running the loader mfile.
Use the mfilename option
to produce a file that you can debug and fix.
Please report this
error to the MathWorks so we can improve this
function.
??? Error using ==> feval
Undefined function or variable 'GHlinkTest_proto'.

Error in ==> loadtest at 6
```

For information about how to properly invoke the MATLAB `loadlibrary` function with MATLAB Compiler, see "Load MATLAB Libraries using loadlibrary" on page 3-19 in the Deploying MATLAB Code section in your product user's guide.

MATLAB Compiler Does Not Generate the Application. If you experience other problems with MATLAB Compiler, contact Technical Support at MathWorks at `http://www.mathworks.com/contact_TS.html`.

"MATLAB file may be corrupt" Message Appears. If you receive the message

```
This MATLAB file does not have proper version information and
may be corrupt. Please delete the extraction directory and
rerun the application.
```

when you run your standalone application that was generated by MATLAB Compiler, you should check the following:

• Do you have a `startup.m` file that calls `addpath`? If so, this will cause run-time errors. As a workaround, use `isdeployed` to have the `addpath` command execute only from MATLAB. For example, use a construct such as:

```
if ~isdeployed
 addpath(path);
end
```

- Verify that the .ctf archive file self extracted and that you have write permission to the folder.

- Verify that none of the files in the <application name>_mcr folder have been modified or removed. Modifying this folder is not supported, and if you have modified it, you should delete it and redeploy or restart the application.

- If none of the above possible causes apply, then the error is likely caused by a corruption. Delete the <application name>_mcr folder and run the application.

Missing Functions in Callbacks. If your application includes a call to a function in a callback string or in a string passed as an argument to the feval function or an ODE solver, and this is the only place in your MATLAB file this function is called, MATLAB Compiler will not compile the function. MATLAB Compiler does not look in these text strings for the names of functions to compile. See "Fixing Callback Problems: Missing Functions" on page 10-3 for more information.

"MCRInstance not available" Message Appears. If you receive the message MCRInstance not available when you try to run a standalone application that was generated with MATLAB Compiler, it can be that the MCR is not located properly on your path or the CTF file is not in the proper folder (if you extracted it from your binary).
The UNIX verification process is the same, except you use the appropriate UNIX path information.

To verify that the MCR is properly located on your path, from a development Windows machine, confirm that *matlabroot*\runtime\win32|win64, where *matlabroot* is your root MATLAB folder, appears on your system path ahead of any other MATLAB installations.

From a Windows target machine, verify that <mcr_root>\<ver>\runtime\win32|win64, where <mcr_root> is your root MCR folder, appears on your system path. To verify that the CTF file that MATLAB Compiler generated in the build process resides in the same folder

as your program's file, look at the folder containing the program's file and make sure the corresponding `.ctf` file is also there.

No Info.plist file in application bundle or no... . On 64-bit Macintosh, indicates the application is not being executed through the bundle.

Deployed Applications

Failed to decrypt file. The MATLAB file "<ctf_root>\toolbox\compiler\deploy\matlabrc.m" cannot be executed. The application is trying to use a CTF archive that does not belong to it. Applications and CTF archives are tied together at compilation time by a unique cryptographic key, which is recorded in both the application and the CTF archive. The keys must match at run time. If they don't match, you will get this error.

To work around this, delete the *_mcr folder corresponding to the CTF archive and then rerun the application. If the same failure occurs, you will likely need to recompile the application using MATLAB Compiler and copy both the application binary and the CTF archive into the installation folder.

This application has requested the run time to terminate in an unusual way. This indicates a segmentation fault or other fatal error. There are too many possible causes for this message to list them all.

To try to resolve this problem, run the application in the debugger and try to get a stack trace or locate the line on which the error occurs. Fix the offending code, or, if the error occurs in a MathWorks library or generated code, contact MathWorks technical support.

Checking access to X display <IP-address>:0.0 . . .
If no response hit ^C and fix host or access control to host.
Otherwise, checkout any error messages that follow and fix . . .
Successful. . . . . This message can be ignored.

**??? Error: File: /home/username/<MATLAB file_name>
Line: 1651 Column: 8
Arguments to IMPORT must either end with ".*"
or else specify a fully qualified class name:
"<class_name>" fails this test.** The import statement is referencing a
Java class (<class_name>) that MATLAB Compiler (if the error occurs at
compile time) or the MCR (if the error occurs at run time) cannot find.

To work around this, ensure that the JAR file that contains
the Java class is stored in a folder that is on the Java class
path. (See *matlabroot*/toolbox/local/classpath.txt for
the class path.) If the error occurs at run time, the classpath
is stored in *matlabroot*/toolbox/local/classpath.txt
when running on the development machine. It is stored in
<mcr_root>/toolbox/local/classpath.txt when running on a target
machine.

**Warning: Unable to find Java library:
matlabroot\sys\java\jre\win32|win64\jre<version>\bin\client\jvm.dll
Warning: Disabling Java support.** This warning indicates that a
compiled application can not find the Java virtual machine, and therefore, the
compiled application cannot run any Java code. This will affect your ability to
display graphics.

To resolve this, ensure that jvm.dll is in the
matlabroot\sys\java\jre\win32|win64\jre<version>\bin\client folder
and that this folder is on your system path.

**Warning: matlabroot\toolbox\local\pathdef.m not found.
Toolbox Path Cache is not being used. Type 'help toolbox_path_cache'
for more info.** The pathdef.m file defines the MATLAB startup path.
MATLAB Compiler does not include this file in the generated CTF archive
because the MCR path is a subset of the full MATLAB path.
This message can be ignored.

Undefined function or variable 'matlabrc'. When MATLAB or the MCR
starts, they attempt to execute the MATLAB file matlabrc.m. This message
means that this file cannot be found.

To work around this, try each of these suggestions in this order:

- Ensure that your application runs in MATLAB (uncompiled) without this error.

- Ensure that MATLAB starts up without this error.

- Verify that the generated CTF archive contains a file called `matlabrc.m`.

- Verify that the generated code (in the `*_mcc_component_data.c*` file) adds the CTF archive folder containing `matlabrc.m` to the MCR path.

- Delete the `*_mcr` folder and rerun the application.

- Recompile the application.

This MATLAB file does not have proper version information and may be corrupt. Please delete the extraction directory and rerun the application. The MATLAB file <MATLAB file> cannot be executed. MATLAB:err_parse_cannot_run_m_file. This message is an indication that the MCR has found nonencrypted MATLAB files on its path and has attempted to execute them. This error is often caused by the use of `addpath`, either explicitly in your application, or implicitly in a `startup.m` file. If you use `addpath` in a compiled application, you must ensure that the added folders contain only data files. (They cannot contain MATLAB files, or you'll get this error.)

To work around this, protect your calls to `addpath` with the `isdeployed` function.

This application has failed to start because mclmcrrt7x.dll was not found. Re-installing the application may fix this problem. `mclmcrrt7x.dll` contains the public interface to the MCR. This library must be present on all machines that run applications generated by MATLAB Compiler. Typically, this means that either the MCR is not installed on this machine, or that the PATH does not contain the folder where this DLL is located.

To work around this, install the MCR or modify the path appropriately. The path must contain `<mcr_root>\<version>\runtime\<arch>`, for example: `c:\mcr\v73\runtime\win32|win64`.

Linker cannot find library and fails to create standalone application (win32 and win64). If you try building your standalone application without `mbuild`, you must link to the following dynamic library:

`mclmcrrt.lib`

This library is found in one of the following locations, depending on your architecture:

matlabroot\extern\lib\win32*arch*
matlabroot\extern\lib\win64*arch*

where *arch* is `microsoft` or `watcom`.

Version 'GCC_4.2.0' not found. When running on Linux platforms, users may report that a run time error occurs that states that the GCC_4.2.0 library is not found by applications built with MATLAB Compiler.

To resolve this error, do the following:

1 Navigate to *matlabroot*/sys/os/glnx86.

2 Rename the following files with a prefix of `old_`:

- `libgcc_s.so.1`
- `libstdc++.so.6.0.8`
- `libgfortran.so.1.0.0`

For example, rename `libgcc_s.so.1` to `old_libgcc_s.so.1`. you must rename all three of the above files. Alternately, you can create a subfolder named `old` and move the files there.

Error: library mclmcrrt76.dll not found. This error can occur for the following reasons:

- The machine on which you are trying to run the application an different, incompatible version of the MCR installed on it than the one the application was originally built with.
- You are not running a version of MATLAB Compiler compatible with the MCR version the application was built with.

To solve this problem, on the deployment machine, install the version of MATLAB you used to build the application.

Invalid .NET Framework.*n* **Either the specified framework was not found or is not currently supported.** This error occurs when the .NET Framework version your application is specifying (represented by *n*) is not supported by the current version of MATLAB Compiler. See the *MATLAB Builder NE Release Notes* for a list of supported .NET Framework versions.

`MATLAB:I18n:InconsistentLocale.` The warning message

```
MATLAB:I18n:InconsistentLocale - The system locale setting,
system_locale_name, is different from the user locale
setting, user_locale_name
```

indicates a mismatch between locale setting on Microsoft Windows systems. This may affect your ability to display certain characters. For information about changing the locale settings, see your operating system Help.

`System.AccessViolationException:  Attempted to read or write protected memory.` The message:

```
System.ArgumentException: Generate Queries
                                 threw General Exception:
System.AccessViolationException: Attempted to
                          read or write protected memory.
 This is often an indication that other memory is corrupt.
```

indicates a library initialization error caused by a Microsoft Visual Studio project linked against a `MCLMCRRT7XX.DLL` placed outside *matlabroot*.

10

Limitations and Restrictions

MATLAB Compiler Limitations

Compiling MATLAB and Toolboxes

MATLAB Compiler supports the full MATLAB language and almost all toolboxes based on MATLAB. However, some limited MATLAB and toolbox functionality is not licensed for compilation.

- Most of the prebuilt graphical user interfaces included in MATLAB and its companion toolboxes will not compile.

- Functionality that cannot be called directly from the command line will not compile.

- Some toolboxes, such as Symbolic Math Toolbox™, will not compile.

Compiled applications can only run on operating systems that run MATLAB. Also, since the MCR is approximately the same size as MATLAB, applications built with MATLAB Compiler need specific storage memory and RAM to operate. For the most up-to-date information about system requirements, go to the MathWorks Web site.

To see a full list of MATLAB Compiler limitations, visit http://www.mathworks.com/products/compiler/compiler_support.html.

Note See "MATLAB Functions That Cannot Be Compiled" on page 10-10 for a list of functions that cannot be compiled.

Fixing Callback Problems: Missing Functions

When MATLAB Compiler creates a standalone application, it compiles the MATLAB file(s) you specify on the command line and, in addition, it compiles any other MATLAB files that your MATLAB file(s) calls. MATLAB Compiler uses a dependency analysis, which determines all the functions on which the supplied MATLAB files, MEX-files, and P-files depend. The dependency analysis may not locate a function if the only place the function is called in your MATLAB file is a call to the function either

- In a callback string
- In a string passed as an argument to the `feval` function or an ODE solver

---

Tip Dependent functions can also be hidden from `depfun` in `.mat` files that get loaded by compiled applications. Use the `mcc -a` argument or the `%#function` pragma to identify `.mat` file classes or functions that should be supported by the `load` command.

MATLAB Compiler does not look in these text strings for the names of functions to compile.

Symptom

Your application runs, but an interactive user interface element, such as a push button, does not work. The compiled application issues this error message:

```
An error occurred in the callback: change_colormap
The error message caught was   : Reference to unknown function
                change_colormap from FEVAL in stand-alone mode.
```

Workaround

There are several ways to eliminate this error:

- Using the `%#function` pragma and specifying callbacks as strings

- Specifying callbacks with function handles

- Using the `-a` option

Specifying Callbacks as Strings. Create a list of all the functions that are specified only in callback strings and pass these functions using separate `%#function` pragma statements. This overrides the product's dependency analysis and instructs it to explicitly include the functions listed in the `%#function` pragmas.

For example, the call to the `change_colormap` function in the sample application, `my_test`, illustrates this problem. To make sure MATLAB Compiler processes the `change_colormap` MATLAB file, list the function name in the `%#function` pragma.

```
function my_test()
% Graphics library callback test application

%#function change_colormap

peaks;

p_btn = uicontrol(gcf,...
                  'Style', 'pushbutton',...
                  'Position',[10 10 133 25 ],...
                  'String', 'Make Black & White',...
```

```
                      'CallBack','change_colormap');
```

Specifying Callbacks with Function Handles. To specify the callbacks with function handles, use the same code as in the example above and replace the last line with

```
'CallBack',@change_colormap);
```

For more information on specifying the value of a callback, see the MATLAB Programming Fundamentals documentation.

Using the -a Option. Instead of using the `%#function` pragma, you can specify the name of the missing MATLAB file on the MATLAB Compiler command line using the `-a` option.

Finding Missing Functions in a MATLAB File

To find functions in your application that may need to be listed in a `%#function` pragma, search your MATLAB file source code for text strings specified as callback strings or as arguments to the `feval`, `fminbnd`, `fminsearch`, `funm`, and `fzero` functions or any ODE solvers.

To find text strings used as callback strings, search for the characters "Callback" or "fcn" in your MATLAB file. This will find all the `Callback` properties defined by Handle Graphics® objects, such as `uicontrol` and `uimenu`. In addition, this will find the properties of figures and axes that end in `Fcn`, such as `CloseRequestFcn`, that also support callbacks.

Suppressing Warnings on the UNIX System

Several warnings may appear when you run a standalone application on the UNIX system. This section describes how to suppress these warnings.

To suppress the `libjvm.so` warning, make sure you set the dynamic library path properly for your platform. See .

You can also use the MATLAB Compiler option `-R -nojvm` to set your application's `nojvm` run-time option, if the application is capable of running without Java.

Cannot Use Graphics with the -nojvm Option

If your program uses graphics and you compile with the `-nojvm` option, you will get a run-time error.

Cannot Create the Output File

If you receive the error

```
Can't create the output file filename
```

there are several possible causes to consider:

- Lack of write permission for the folder where MATLAB Compiler is attempting to write the file (most likely the current working folder).
- Lack of free disk space in the folder where MATLAB Compiler is attempting to write the file (most likely the current working folder).
- If you are creating a standalone application and have been testing it, it is possible that a process is running and is blocking MATLAB Compiler from overwriting it with a new version.

No MATLAB File Help for Compiled Functions

If you create a MATLAB file with self-documenting online help by entering text on one or more contiguous comment lines beginning with the second line of the file and then compile it, the results of the command

```
help filename
```

will be unintelligible.

Note Due to performance reasons, MATLAB file comments are stripped out before MCR encryption.

No MCR Versioning on Mac OS X

The feature that allows you to install multiple versions of the MCR on the same machine is currently not supported on Mac OS X. When you receive a new version of MATLAB , you must recompile and redeploy all of your

applications and components. Also, when you install a new MCR onto a target machine, you must delete the old version of the MCR and install the new one. You can only have one version of the MCR on the target machine.

Older Neural Networks Not Deployable with MATLAB Compiler

Loading networks saved from older Neural Network Toolbox versions requires some initialization routines that are not deployable. Therefore, these networks cannot be deployed without first being updated.

For example, deploying with Neural Network Toolbox Version 5.0.1 (2006b) and MATLAB Compiler Version 4.5 (R2006b) yields the following errors at run time:

```
??? Error using ==> network.subsasgn
"layers{1}.initFcn" cannot be set to non-existing
 function "initwb".
Error in ==> updatenet at 40
Error in ==> network.loadobj at 10

??? Undefined function or method 'sim' for input
arguments of type 'struct'.
Error in ==> mynetworkapp at 30
```

Restrictions on Calling PRINTDLG with Multiple Arguments in Compiled Mode

In compiled mode, only one argument can be present in a call to the MATLAB printdlg function (for example, printdlg(gcf)).

You will not receive an error when making at call to printdlg with multiple arguments. However, when an application containing the multiple-argument call is compiled, the compile will fail with the following error message:

```
Error using = => printdlg at 11
PRINTDLG requires exactly one argument
```

Compiling a Function with WHICH Does Not Search Current Working Directory

Using which, as in this example:

```
function pathtest
which myFile.mat
open('myFile.mat')
```

does not cause the current working folder to be searched in deployed applications. In addition, it may cause unpredictable behavior of the open function.

Use one of the following solutions as alternatives to using which:

- Use the pwd function to explicitly point to the file in the current folder, as follows:

  ```
  open([pwd 'myFile.mat'])
  ```

- Rather than using the general open function, use load or other specialized functions for your particular file type, as load explicitly checks for the file in the current folder. For example:

  ```
  load myFile.mat
  ```

- Include your file using the **Other Files** area of your project using deploytool (and the -a flag using mcc).

Restrictions on Using C++ SETDATA to Dynamically Resize an MWArray

You cannot use the C++ SETDATA function to dynamically resize MWArrays.

For instance, if you are working with the following array:

```
[1 2 3 4]
```

you cannot use SETDATA to increase the size of the array to a length of five elements.

Licensing Terms and Restrictions on Compiled Applications

Applications you build with a trial MATLAB Compiler license are valid for thirty (30) days only.

Applications you build with a purchased license of MATLAB Compiler have no expiration date.

MATLAB Functions That Cannot Be Compiled

> **Note** Due to the number of active and ever-changing list of MathWorks products and functions, this is not a complete list of functions that can not be compiled. If you have a question as to whether a specific MathWorks product's function is able to be compiled or not, the definitive source is that product's documentation, not the MATLAB Compiler documentation.

Some functions are not supported in standalone mode; that is, you cannot compile them with MATLAB Compiler. These functions are in the following categories:

- Functions that print or report MATLAB code from a function, for example, the MATLAB `help` function or debug functions, will not work.

- Simulink® functions, in general, will not work.

- Functions that require a command line, for example, the MATLAB `lookfor` function, will not work.

- `clc`, `home`, and `savepath` will not do anything in deployed mode.

- Tools that allow run-time manipulation of figures

Returned values from standalone applications will be 0 for successful completion or a nonzero value otherwise.

In addition, there are functions and programs that have been identified as nondeployable due to licensing restrictions.

`mccExcludedFiles.log` lists all the functions and files excluded by `mcc` if they can not be compiled. It is created after each attempted build if there are functions or files that cannot be compiled.

List of Unsupported Functions and Programs

```
add_block
add_line
```

List of Unsupported Functions and Programs (Continued)

applescript

checkcode

close_system

colormapeditor

commandwindow

Control System Toolbox™ prescale GUI

createClassFromWsdl

dbclear

dbcont

dbdown

dbquit

dbstack

dbstatus

dbstep

dbstop

dbtype

dbup

delete_block

delete_line

depfun

doc

echo

edit

fields

figure_palette

get_param

help

List of Unsupported Functions and Programs (Continued)

home

inmem

keyboard

linkdata

linmod

mislocked

mlock

more

munlock

new_system

open_system

pack

pcode

plotbrowser

plotedit

plottools

profile

profsave

propedit

propertyeditor

publish

rehash

restoredefaultpath

run

segment

set_param

List of Unsupported Functions and Programs (Continued)

```
sim
simget
simset
sldebug
type
```

Reference Information

MCR Path Settings for Development and Testing

Overview

The following information is for developers developing applications that use libraries or components that contain compiled MATLAB code. These settings are required on the machine where you are developing your application. Other settings required by end users at run time are described in .

Note For *matlabroot*, substitute the MATLAB root folder on your system. Type matlabroot to see this folder name.

Path for Java Development on All Platforms

There are additional requirements when programming in the Java programming language. See "Deploying Applications That Call the Java Native Libraries" on page 6-25.

Path Modifications Required for Accessibility

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

```
JavaAccessBridge.dll
WindowsAccessBridge.dll
```

You may not be able to use such technologies without doing so.

Windows Settings for Development and Testing

When programming with components that are generated with MATLAB Compiler, add the following folder to your system PATH environment variable:

matlabroot\runtime\win32|win64

Linux Settings for Development and Testing

Add the following platform-specific folders to your dynamic library path.

Note For readability, the following commands appear on separate lines, but you must enter each setenv command on one line.

Linux (64-bit)

```
setenv LD_LIBRARY_PATH
   matlabroot/runtime/glnxa64:
   matlabroot/bin/glnxa64:
   matlabroot/sys/os/glnxa64:
   matlabroot/sys/java/jre/glnxa64/jre/lib/amd64/native_threads:
   matlabroot/sys/java/jre/glnxa64/jre/lib/amd64/server:
   matlabroot/sys/java/jre/glnxa64/jre/lib/amd64:
```

Mac Settings for Development and Testing

Add the following platform-specific folders to your dynamic library path.

Note For readability, the following commands appear on separate lines, but you must enter each setenv command on one line.

Mac

```
setenv DYLD_LIBRARY_PATH
   matlabroot/runtime/maci64:
   matlabroot/bin/maci64:
   matlabroot/sys/os/maci64:
```

MCR Path Settings for Run-time Deployment

In this section...
"General Path Guidelines" on page 11-4
"Path for Java Applications on All Platforms" on page 11-4
"Windows Path for Run-Time Deployment" on page 11-4
"Linux Paths for Run-Time Deployment" on page 11-5
"Mac Paths for Run-Time Deployment" on page 11-5

General Path Guidelines

Regardless of platform, be aware of the following guidelines with regards to placing specific folders on the path:

- Always avoid including `bin` or `arch` on the path. Failure to do so may inhibit ability to run multiple MCR instances.

- Ideally, set the environment in a separate shell script to avoid run-time errors caused by path-related issues.

Path for Java Applications on All Platforms

When your users run applications that contain compiled MATLAB code, you must instruct them to set the path so that the system can find the MCR.

Note When you deploy a Java application to end users, they must set the class path on the target machine.

The system needs to find `.jar` files containing the MATLAB libraries. To tell the system how to locate the `.jar` files it needs, specify a `classpath` either in the `javac` command or in your system environment variables.

Windows Path for Run-Time Deployment

The following folder should be added to the system path:

mcr_root*version*\runtime\win32|win64

where *mcr_root* refers to the complete path where the MCR library archive files are installed on the machine where the application is to be run.

mcr_root is version specific; you must determine the path after you install the MCR.

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

Linux Paths for Run-Time Deployment

Use these setenv commands to set your MCR run-time paths.

If you are unfamiliar with these commands, see "Set MCR Paths on Mac or Linux with Scripts" on page B-12 for a detailed procedural and troubleshooting guide, as well as pointers to other information about installing, configuring, building, deploying, and integrating your MATLAB code on Linux.

Note For readability, the following commands appear on separate lines, but you must enter each setenv command on one line. The setenv command is specific to the C shell (csh). See *MATLAB External Interfaces* for more information.

Linux (64-bit)

```
setenv LD_LIBRARY_PATH
    mcr_root/version/runtime/glnxa64:
    mcr_root/version/bin/glnxa64:
    mcr_root/version/sys/os/glnxa64:
    mcr_root/version/sys/java/jre/glnxa64/jre/lib/amd64/native_threads:
    mcr_root/version/sys/java/jre/glnxa64/jre/lib/amd64/server:
    mcr_root/version/sys/java/jre/glnxa64/jre/lib/amd64:
```

Mac Paths for Run-Time Deployment

Use these setenv commands to set your MCR run-time paths.

If you are unfamiliar with these commands, see "Set MCR Paths on Mac or Linux with Scripts" on page B-12 for a detailed procedural and troubleshooting guide, as well as pointers to other information about installing, configuring, building, deploying, and integrating your MATLAB code on a Mac.

Note For readability, the following commands appear on separate lines, but you must enter each `setenv` command on one line. The `setenv` command is specific to the C shell (csh). See *MATLAB External Interfaces* for more information.

Mac

```
setenv DYLD_LIBRARY_PATH
   mcr_root/version/runtime/maci64:
   mcr_root/version/bin/maci64:
   mcr_root/version/sys/os/maci64:
```

MATLAB Compiler Licensing

Using MATLAB Compiler Licenses for Development

You can run MATLAB Compiler from the MATLAB command prompt (MATLAB mode) or the DOS/UNIX prompt (standalone mode).

MATLAB Compiler uses a *lingering license*. This means that when the MATLAB Compiler license is checked out, a timer is started. When that timer reaches 30 minutes, the license key is returned to the license pool. The license key will not be returned until that 30 minutes is up, regardless of whether mcc has exited or not.

Each time a compiler command is issued, the timer is reset.

Running MATLAB Compiler in MATLAB Mode

When you run MATLAB Compiler from "inside" of the MATLAB environment, that is, you run mcc from the MATLAB command prompt, you hold the MATLAB Compiler license as long as MATLAB remains open. To give up the MATLAB Compiler license, exit MATLAB.

Running MATLAB Compiler in Standalone Mode

If you run MATLAB Compiler from a DOS or UNIX prompt, you are running from "outside" of MATLAB. In this case, MATLAB Compiler

- Does not require MATLAB to be running on the system where MATLAB Compiler is running
- Gives the user a dedicated 30-minute time allotment during which the user has complete ownership over a license to MATLAB Compiler

Each time a user requests MATLAB Compiler , the user begins a 30-minute time period as the sole owner of the MATLAB Compiler license. Anytime during the 30-minute segment, if the same user requests MATLAB Compiler , the user gets a new 30-minute allotment. When the 30-minute interval has elapsed, if a different user requests MATLAB Compiler , the new user gets the next 30-minute interval.

When a user requests MATLAB Compiler and a license is not available, the user receives the message

```
Error: Could not check out a Compiler License.
```

This message is given when no licenses are available. As long as licenses are available, the user gets the license and no message is displayed. The best way to guarantee that all MATLAB Compiler users have constant access to MATLAB Compiler is to have an adequate supply of licenses for your users.

Application Deployment Terms

Glossary of Deployment Product Terms

A

Add-in — A Microsoft Excel add-in is an executable piece of code that can be actively integrated into a Microsoft Excel application. Add-ins are front-ends for COM components, usually written in some form of Microsoft Visual Basic®.

API — Application program interface. An implementation of the proxy software design pattern. See *MWArray*.

Application — An end user-system into which a deployed functions or solution is ultimately integrated. Typically, the end goal for the Deployment customer is integration of a deployed MATLAB function into a larger enterprise environment application. The deployment products prepare the MATLAB function for integration by wrapping MATLAB code with enterprise-compatible source code, such as C, C++, C# (.NET), F#, and Java code.

Assembly — An executable bundle of code, especially in .NET. For example, after building a deployable .NET component with MATLAB Builder NE, the .NET developer integrates the resulting .NET assembly into a larger enterprise C# application. See *Executable*.

B

Binary — See *Executable*.

Boxed Types — Data types used to wrap opaque C structures.

Build — See *Compile*.

C

Class — A user-defined type used in C++, C#, and Java, among other object-oriented languages that is a prototype for an object in an object-oriented language. It is analogous to a derived type in a procedural language. A class is a set of objects which share a common structure and behavior. Classes

relate in a class hierarchy. One class is a specialization (a *subclass*) of another (one of its *superclasses*) or comprises other classes. Some classes use other classes in a client-server relationship. Abstract classes have no members, and concrete classes have one or more members. Differs from a *MATLAB class*

Compile — In MATLAB Compiler terminology, to compile a component involves generating a binary that wraps around MATLAB code, enabling it to execute in various computing environments. For example, when MATLAB code builds with MATLAB Builder JA, a Java wrapper provides Java code that enables the MATLAB code to execute in a Java environment.

COM component — In MATLAB Builder EX, the executable back-end code behind a Microsoft Excel add-in. In MATLAB Builder NE, an executable component, to be integrated with Microsoft COM applications.

Component — In MATLAB, a generic term used to describe the wrappered MATLAB code produced by MATLAB Compiler. You can plug these self-contained bundles of code you plug into various computing environments. The wrapper enables the compatibility between the computing environment and your code.

Console application — Any application that is executed from a system command prompt window. If you are using a non-Windows operating system, console applications are often referred to as standalone applications.

CTF archive (Component Technology File) — The Component Technology File (CTF) archive is embedded by default in each generated binary by MATLAB Compiler. It houses the deployable package. All MATLAB-based content in the CTF archive uses the Advanced Encryption Standard (AES) cryptosystem. See "Additional Details" on page 3-11 in the MATLAB Compiler documentation.

D

Data Marshaling — Data conversion, usually from one type to another. Unless a MATLAB deployment customer is using type-safe interfaces, data marshaling—as from mathematical data types to MathWorks data types such as represented by the MWArray API—must be performed manually, often at great cost.

Deploy — The act of integrating a component into a larger-scale computing environment, usually to an enterprise application, and often to end users.

DLL — Dynamic link library. Microsoft's implementation of the shared library concept for Windows. Using DLLs is much preferred over the previous technology of static (or non-dynamic) libraries, which had to be manually linked and updated.

E

Empties — Arrays of zero (0) dimensions.

Executable — An executable bundle of code, made up of binary bits (zeros and ones) and sometimes called a *binary*.

F

Fields — For this definition in the context of MATLAB Data Structures, see *Structs*.

Fields and Properties — In the context of .NET, *Fields* are specialized classes used to hold data. *Properties* allow users to access class variables as if they were accessing member fields directly, while actually implementing that access through a class method.

H

Helper files — Files that support the main file or the file that calls all supporting functions. Add resources that depend upon the function that calls the supporting function to the **Shared Resources and Helper Files** section of the Deployment Tool GUI. Other examples of supporting files or resources include:

- Functions called using `eval` (or variants of `eval`)

- Functions not on the MATLAB path

- Code you want to remain private

- Code from other programs that you want to compile and link into the main file

I

Integration — Combining a deployed component's functionality with functionality that currently exists in an enterprise application. For example, a customer creates a mathematical model to forecast trends in certain commodities markets. In order to use this model in a larger-scale financial application (one written with the Microsoft .NET Framework, for instance) the deployed financial model must be integrated with existing C# applications, run in the .NET enterprise environment. Integration is usually performed by an IT developer, rather than a MATLAB Programmer, in larger environments.

Instance — For the definition of this term in context of MATLAB Production Server™ software, see *MATLAB Production Server Server Instance*.

J

JAR — Java archive. In computing software, a JAR file (or Java ARchive) aggregates many files into one. Software developers generally use JARs to distribute Java applications or libraries, in the form of classes and associated metadata and resources (text, images, etc.). Computer users can create or extract JAR files using the `jar` command that comes with a Java Development Kit (JDK).

Java-MATLAB Interface — Known as the *JMI Interface*, this is the Java interface built into MATLAB software.

JDK — The *Java Development Kit* is a free Sun Microsystems product which provides the environment required for programming in Java. The JDK™ is available for various platforms, but most notably Sun™ Solaris™ and Microsoft Windows. To build components with MATLAB Builder JA, download the JDK that corresponds to the latest version of Java supported by MATLAB.

JMI Interface — see *Java-MATLAB Interface*.

JRE — *Java Run-Time Environment* is the part of the Java Development Kit (JDK) required to run Java programs. It comprises the Java Virtual Machine, the Java platform core classes, and supporting files. It does not include the compiler, debugger, or other tools present in the JDK. The JRE™

is the smallest set of executables and files that constitute the standard Java platform.

M

Magic Square — A square array of integers arranged so that their sum is the same when added vertically, horizontally, or diagonally.

MATLAB Production Server Client — In the MATLAB Production Server software, clients are applications written in a language supported by MATLAB Production Server that call deployed functions hosted on a server.

MATLAB Production Server Configuration — An instance of the MATLAB Production Server containing at least one server and one client. Each configuration of the software usually contains a unique set of values in the server configuration file, `main_config`.

MATLAB Production Server Server Instance — A logical server configuration created using the `mps-new` command in MATLAB Production Server software.

MATLAB Production Server Software — Product for server/client deployment of MATLAB programs within your production systems, enabling you to incorporate numerical analytics in enterprise applications. When you use this software, Web, database, and enterprise applications connect to MATLAB programs running on MATLAB Production Server via a lightweight client library, isolating the MATLAB programs from your production system. MATLAB Production Server software consists of one or more servers and clients.

Marshaling — See *Data Marshaling*.

mbuild — MATLAB Compiler command that invokes a script which compiles and links C and C++ source files into standalone applications or shared libraries. For more information, see the `mbuild` function reference page.

mcc — The MATLAB command that invokes MATLAB Compiler. It is the command-line equivalent of using the Deployment Tool GUI. See the `mcc` reference page for the complete list of options available. Each builder product has customized `mcc` options. See the respective builder documentation for details.

MCR — The *MATLAB Compiler Runtime* is an execution engine made up of the same shared libraries. MATLAB uses these libraries to enable the execution of MATLAB files on systems without an installed version of MATLAB. To deploy a component, you package the MCR along with it. Before you use the MCR on a system without MATLAB, run the *MCR Installer*.

MCR Installer — An installation program run to install the MATLAB Compiler Runtime on a development machine that does not have an installed version of MATLAB. Find out more about the MCR Installer by reading "Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)" on page 1-23.

MCR Singleton — See *Shared MCR Instance.*

MCR Workers — A MATLAB Compiler Runtime session. Using MATLAB Production Server software, you have the option of specifying more than one MCR session, using the `--num-workers` options in the server configurations file, `main_config`.

Method Attribute — In the context of .NET, a mechanism used to specify declarative information to a .NET class. For example, in the context of client programming with MATLAB Production Server software, you specify method attributes to define MATLAB structures for input and output processing.

mxArray interface — The MATLAB data type containing all MATLAB representations of standard mathematical data types.

MWArray interface — A proxy to `mxArray`. An application program interface (API) for exchanging data between your application and MATLAB. Using `MWArray`, you marshal data from traditional mathematical types to a form that can be processed and understood by MATLAB data type `mxArray`. There are different implementations of the `MWArray` proxy for each application programming language.

P

Package — The act of bundling the deployed component, along with the MCR and other files, for rollout to users of the MATLAB deployment products. After running the packaging function of the Deployment Tool, the package file resides in the `distrib` subfolder. On Windows®, the package is a

self-extracting executable. On platforms other than Windows, it is a `.zip` file. Use of this term is unrelated to *Java Package*.

PID File — See *Process Identification File (PID File)*.

Pool — A pool of threads, in the context of server management using MATLAB Production Server software. Servers created with the software do not allocate a unique thread to each client connection. Rather, when data is available on a connection, the required processing is scheduled on a *pool*, or group, of available threads. The server configuration file option `--num-threads` sets the size of that pool (the number of available request-processing threads) in the master server process.

Process Identification File (PID File) — A file that documents informational and error messages relating to a running server instance of MATLAB Production Server software.

Program — A bundle of code that is executed to achieve a purpose. Programs usually are written to automate repetitive operations through computer processing. Enterprise system applications usually consist of hundreds or even thousands of smaller programs.

Properties — For this definition in the context of .NET, see *Fields and Properties*.

Proxy — A software design pattern typically using a class, which functions as an interface to something else. For example, `MWArray` is a proxy for programmers who need to access the underlying type `mxArray`.

S

Server Instance — See MATLAB Production Server Server Instance.

Shared Library — Groups of files that reside in one space on disk or memory for fast loading into Windows applications. Dynamic-link libraries (DLLs) are Microsoft's implementation of the shared library concept in for Microsoft Windows.

Shared MCR Instance — When using MATLAB Builder NE or MATLAB Builder JA, you can create a shared MCR instance, also known as a *singleton*.

For builder NE, this only applies to COM components. When you invoke MATLAB Compiler with the -S option through the builders (using either mcc or the Deployment Tool), a single MCR instance is created for each COM or Java component in an application. You reuse this instance by sharing it among all subsequent class instances within the component. Such sharing results in more efficient memory usage and eliminates the MCR startup cost in each subsequent class instantiation. All class instances share a single MATLAB workspace and share global variables in the MATLAB files used to build the component. MATLAB Builder NE and MATLAB Builder EX are designed to create singletons by default for .NET assemblies and COM components, respectively. For more information, see "Sharing an MCR Instance in COM or Java Applications".

Standalone application — Programs that are not part of a bundle of linked libraries (as in shared libraries). Standalones are not dependent on operating system services and can be accessed outside of a shared network environment. Standalones are typically .exes (EXE files) in the Windows run-time environment.

State — A specific data value in a program or program variable. MATLAB functions often carry state, in the form of variable values or even the MATLAB Workspace itself. When deploying functions that carry state, you must often take additional steps to ensure state retention when deploying applications that use such functions. When running MATLAB Production Server software, there are additional considerations for preserving state. See "Write MATLAB Code for Deployment" in the *MATLAB Production Server User's Guide* for additional information.

Structs — MATLAB Structures. Structs are MATLAB arrays with elements that you access using textual field designators. Fields are data containers that store data of a specific MATLAB type.

System Compiler — A key part of Interactive Development Environments (IDEs) such as Microsoft Visual Studio. Before using MATLAB Compiler, select a system compiler using the MATLAB command mbuild -setup.

T

Threads — In the context of MATLAB Production Server software, this term has essentially the same meaning as in any server/client software product.

See *pool* for additional information on managing the number of processing threads available to a server instance.

Type-safe interface — An API that minimizes explicit type conversions by hiding the `MWArray` type from the calling application. Using "Generate and Implement Type-Safe Interfaces", for example, .NET Developers work directly with familiar native data types. You can avoid performing tedious `MWArray` data marshaling by using type-safe interfaces.

W

WAR — Web Application ARchive. In computing, a WAR file is a JAR file used to distribute a collection of `JavaServer` pages, servlets, Java classes, XML files, tag libraries, and static Web pages (HTML and related files) that together constitute a Web application.

WCF — Windows Communication Foundation. The Windows Communication Foundation™ (or WCF) is an application programming interface in the .NET Framework for building connected, service-oriented, Web-centric applications. WCF is designed in accordance with service oriented architecture principles to support distributed computing where services are consumed. Clients consume multiple services that can be consumed by multiple clients. Services are loosely coupled to each other.

Webfigure — A MathWorks representation of a MATLAB figure, rendered on the Web. Using the WebFigures feature, you display MATLAB figures on a Web site for graphical manipulation by end users. This enables them to use their graphical applications from anywhere on the Web, without the need to download MATLAB or other tools that can consume costly resources.

Windows standalone application — Windows standalones differ from regular standalones in that Windows standalones suppress their MS-DOS window output. The equivalent method to specify a Windows standalone target on the `mcc` command line is "-e Suppress MS-DOS Command Window" on page 12-28 If you are using a non-Windows operating system, console applications are referred to as standalone applications.

12

Functions — Alphabetical List

%#function
ctfroot
deployprint
deploytool
figToImStream
getmcruserdata
<library>Initialize[WithHandlers]
isdeployed
ismcc
mbuild
mcc
mclGetLastErrorMessage
mclGetLogFileName
mclInitializeApplication
mclIsJVMEnabled
mclIsMCRInitialized
mclIsNoDisplaySet
mclRunMain
mclTerminateApplication
mclWaitForFiguresToDie
mcrinstaller
mcrversion
setmcruserdata
<library>Terminate

%#function

Purpose   Pragma to help MATLAB Compiler locate functions called through
`feval`, `eval`, or Handle Graphics callback

Syntax   %#function *function1* [*function2 ... functionN*]

%#function *object_constructor*

Description   The %#function pragma informs MATLAB Compiler that the specified
function(s) will be called through an `feval`, `eval`, or Handle Graphics
callback.

Use the %#function pragma in standalone applications to inform
MATLAB Compiler that the specified function(s) should be included
in the compilation, whether or not MATLAB Compiler's dependency
analysis detects the function(s). It is also possible to include objects by
specifying the object constructor.

Without this pragma, the product's dependency analysis will not be able
to locate and compile all MATLAB files used in your application. This
pragma adds the top-level function as well as all the local functions in
the file to the compilation.

Examples   ### Example 1

```
function foo
  %#function bar

      feval('bar');

  end %function foo
```

By implementing this example, MATLAB Compiler is notified that
function `bar` will be included in the compilation and is called through
`feval`.

Example 2

```
function foo
  %#function bar foobar
```

```
    feval('bar');
    feval('foobar');

end %function foo
```

In this example, multiple functions (bar and foobar) are included in the compilation and are called through feval.

ctfroot

Purpose Location of files related to deployed application (CTF archive)

Syntax *ctfroot*

Description root = *ctfroot* returns a string that is the name of the folder where the CTF file for the deployed application is expanded.

This function differs from matlabroot, which returns the path to where core MATLAB functions and libraries are located. matlabroot returns the root directory of the MCR when run against an installed MCR.

To determine the location of various toolbox folders in deployed mode, use the toolboxdir function.

Examples appRoot = ctfroot; will return the location of your deployed application files in this form: *application_name*_mcr.

Use this function to access any file that the user would have included in their project (excluding the ones in the packaging folder).

How To • "Component Technology File (CTF Archive)" on page 3-8

Purpose   Use to print (as substitute for MATLAB print function) when working
with deployed Windows applications

Syntax   `deployprint`

Description   In cases where the `print` command would normally be issued when
running MATLAB software, use `deployprint` when working with
deployed applications.

`deployprint` is available on all platforms, however it is only required
on Windows.

`deployprint` supports all of the input arguments supported by `print`
except for the following.

Argument	Description
-d	Used to specify the type of the output (for example. .JPG, .BMP, etc.). deployprint only produces .BMP files. **Note** To print to a file, use the `print` function.
-noui	Used to suppress printing of user interface controls. Similar to use in MATLAB `print` function.
-setup	The -setup option is not supported.
-s *windowtitle*	MATLAB Compiler does not support Simulink®.

`deployprint` supports a subset of the figure properties supported by
`print`. The following are supported:

- `PaperPosition`
- `PaperSize`
- `PaperUnits`

deployprint

- Orientation

- PrintHeader

> **Note** deployprint requires write access to the file system in order to write temporary files.

Examples   The following is a simple example of how to print a figure in your application, regardless of whether the application has been deployed or not:

```
figure;
plot(1:10);
if isdeployed
 deployprint;
else
 print(gcf);
end
```

See Also   isdeployed

Purpose	Open Deployment Tool, GUI for MATLAB Compiler
Syntax	`deploytool`
Description	The `deploytool` command opens the Deployment Tool window, which is the graphical user interface (GUI) for MATLAB Compiler. To get started using the Deployment Tool to create standalone applications and libraries, see . You can start `deploytool` in this manner on all platforms supported by MATLAB Compiler.

Desired Results	Command
Start Deployment Tool GUI with the **New/Open** dialog box active	`deploytool` (default) or `deploytool -n`
Start Deployment Tool GUI and load *project_name*	`deploytool` *project_name*`.prj`
Start Deployment Tool command line interface and build project_name after initializing	`deploytool -build` *project_name*`.prj`
Start Deployment Tool command line interface and package project_name after initializing	`deploytool -package` *project_name*`.prj`
Start Deployment Tool and package an existing project from the Command Line Interface. Specifying the *package_name* is optional. By default, a project is packaged into a `.zip` file. On Windows, if the *package_name* ends with `.exe`, the project is packaged into a self-extracting `.exe`.	`deploytool -package` *project_name*`.prj` *package_name*
Display MATLAB Help for the `deploytool` command	`deploytool -?`

-win32 Run in 32-Bit Mode

Use this option to build a 32-bit application on a 64-bit system *only* when the following are both true:

- You use the same MATLAB installation root (`matlabroot`) for both 32-bit and 64-bit versions of MATLAB.

- You are running from a Windows command line (not a MATLAB command line).

Examples
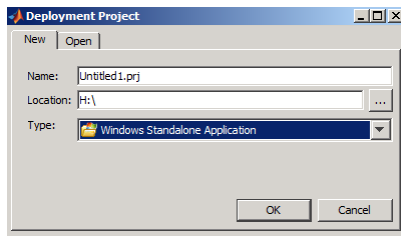
The Deployment Project Dialog Box (Shown when Deployment Tool is first launched)
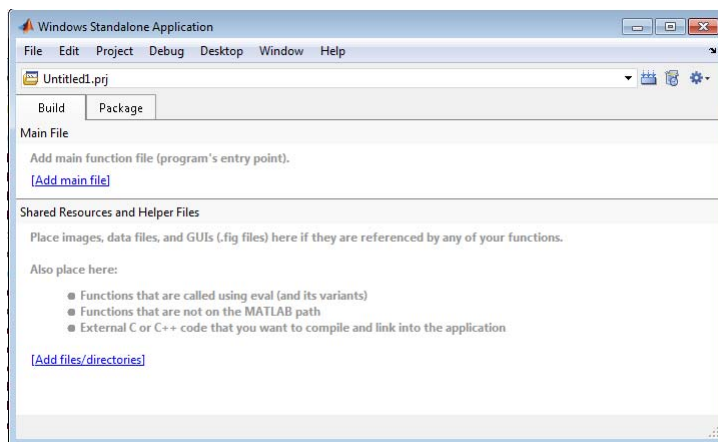
The Deployment Tool GUI (Shown after Creating a Project or Opening a Project)

Purpose	Stream out figure as byte array encoded in format specified, creating signed byte array in `.png` format
Syntax	*output type* = figToImStream ('fighandle', *figure_handle*, 'imageFormat', *image_format*, 'outputType', *output_type*)
Description	The *output type* = figToImStream ('fighandle', *figure_handle*, 'imageFormat', *image_format*, 'outputType', *output_type*) command also accepts user-defined variables for any of the input arguments, passed as a comma-separated list
	The size and position of the printed output depends on the figure's `PaperPosition[mode]` properties.
Options	`figToImStream('figHandle', Figure_Handle, ...)` allows you to specify the figure output to be used. The Default is the current image
	`figToImStream('imageFormat', [png|jpg|bmp|gif])` allows you to specify the converted image format. Default value is `png`.
	`figToImStream('outputType', [int8!uint8])` allows you to specify an output byte data type. `uint8` (unsigned byte) is used primarily for .NET primitive byte. Default value is `uint8`.
Examples	Convert the current figure to a signed `png` byte array:

```
surf(peaks)
bytes = figToImStream
```

Convert a specific figure to an unsigned `bmp` byte array:

```
f = figure;
surf(peaks);
bytes = figToImStream(  'figHandle', f, ...
                        'imageFormat', 'bmp', ...
                        'outputType', 'uint8' );
```

getmcruserdata

Purpose	Retrieve MATLAB array value associated with given string key
Syntax	*function_value* = getmcruserdata(*key*)
Description	The *function_value* = getmcruserdata(*key*) command is part of the MCR User Data interface API. It returns an empty matrix if no such key exists. For information about this function, as well as complete examples of usage, see "Using the MCR User Data Interface" on page 5-34.
	The MATLAB functions getmcruserdata and setmcruserdata can be dragged and dropped (as you would any other MATLAB file), directly to the deploytool GUI.
Examples	*function_value* = getmcruserdata('ParallelProfile');
See Also	setmcruserdata

<library>Initialize[WithHandlers]

Purpose   Initialize MCR instance associated with *library*

Syntax   bool *library*Initialize(void)
                     bool *library*InitializeWithHandlers(
                                  mclOutputHandlerFcn error_handler,
                                  mclOutputHandlerFcn print_handler)

Description   Each generated library has its own MCR instance. These two functions,
                    *library*Initialize and *library*InitializeWithHandlers initialize
                    the MCR instance associated with *library*. Users must call one of these
                    functions after calling mclInitializeApplication and before calling
                    any of the compiled functions exported by the library. Each returns
                    a boolean indicating whether or not initialization was successful. If
                    they return false, calling any further compiled functions will result
                    in unpredictable behavior. *library*InitializeWithHandlers allows
                    users to specify how to handle error messages and printed text. The
                    functions passed to *library*InitializeWithHandlers will be installed
                    in the MCR instance and called whenever error text or regular text
                    is to be output.

Examples
```
if (!libmatrixInitialize())
{
    fprintf(stderr,
             An error occurred while initializing: \n %s  ,
            mclGetLastErrorMessage());
    return -2;
}
```

See Also   <library>Terminate

How To   • "Library Initialization and Termination Functions" on page 8-26

isdeployed

Purpose　　　Determine whether code is running in deployed or MATLAB mode

Syntax　　　　`x = isdeployed`

Description　　`x = isdeployed` returns true (1) when the function is running in deployed mode and false (0) if it is running in a MATLAB session.

If you include this function in an application and compile the application with MATLAB Compiler, the function will return true when the application is run in deployed mode. If you run the application containing this function in a MATLAB session, the function will return false.

Purpose	Test if code is running during compilation process (using mcc)
Syntax	x = ismcc

Description   x = ismcc returns true when the function is being executed by mcc dependency checker and false otherwise.

When this function is executed by the compilation process started by mcc, it will return true. This function will return false when executed within MATLAB as well as in deployed mode. To test for deployed mode execution, use isdeployed. This function should be used to guard code in matlabrc, or hgrc (or any function called within them, for example startup.m in the example on this page), from being executed by MATLAB Compiler (mcc) or any of the Builder products.

In a typical example, a user has ADDPATH calls in their MATLAB code. These can be guarded from executing using ismcc during the compilation process and isdeployed for the deployed application or component as shown in the example on this page.

Examples
```
`% startup.m
   if ~(ismcc || isdeployed)
      addpath(fullfile(matlabroot,'work'));
   end
```

See Also   isdeployed | mcc

mbuild

Purpose   Compile and link source files into standalone application or shared library

Syntax
```
mbuild [option1 ... optionN] sourcefile1 [... sourcefileN]
    [objectfile1 ... objectfileN] [libraryfile1 ... libraryfileN]
    [exportfile1 ... exportfileN]
```

Note Supported types of source files are `.c`, `.cpp`, `.idl`, `.rc`. To specify IDL source files to be compiled with the Microsoft Interface Definition Language (MIDL) Compiler, add `<filename>.idl` to the `mbuild` command line. To specify a DEF file, add `<filename>.def` to the command line. To specify an RC file, add `<filename>.rc` to the command line. Source files that are not one of the supported types are passed to the linker.

Description   `mbuild` is a script that supports various options that allow you to customize the building and linking of your code. This table lists the set of `mbuild` options. If no platform is listed, the option is available on both UNIX and Windows.

Option	Description
@<rspfile>	(Windows only) Include the contents of the text file `<rspfile>` as command line arguments to `mbuild`.
-<arch>	Build an output file for architecture -`<arch>`. To determine the value for -`<arch>`, type `computer ('arch')` at the MATLAB Command Prompt on the target machine. Note: Valid values for -`<arch>` depend on the architecture of the build platform.
-c	Compile only. Creates an object file only.

Option	Description
-D<name>	Define a symbol name to the C preprocessor. Equivalent to a #define <name> directive in the source.
-D<name>=<value>	Define a symbol name and value to the C preprocessor. Equivalent to a #define <name> <value> directive in the source.
-f <optionsfile>	Specify location and name of options file to use. Overrides the mbuild default options file search mechanism.
-g	Create an executable containing additional symbolic information for use in debugging. This option disables the mbuild default behavior of optimizing built object code (see the -O option).
-h[elp]	Print help for mbuild.
-I<pathname>	Add <pathname> to the list of folders to search for #include files.
-inline	Inline matrix accessor functions (mx*). The executable generated may not be compatible with future versions of MATLAB.
-install_name	Fully-qualified path name of product installation on Mac.

mbuild

Option	Description
-l\<name\>	Link with object library. On Windows systems, \<name\> expands to \<name\>.lib or lib\<name\>.lib and on UNIX systems, to lib\<name\>.so or lib\<name\>.dylib. Do not add a space after this switch. **Note** When linking with a library, it is essential that you first specify the path (with -I\<pathname\>, for example).
-L\<folder\>	Add \<folder\> to the list of folders to search for libraries specified with the -l option. On UNIX systems, you must also set the run-time library path, as explained in . Do not add a space after this switch.
-lang \<language\>	Specify compiler language. \<language\> can be c or cpp. By default, mbuild determines which compiler (C or C++) to use by inspection of the source file's extension. This option overrides that default.
-n	No execute mode. Print out any commands that mbuild would otherwise have executed, but do not actually execute any of them.
-O	Optimize the object code. Optimization is enabled by default and by including this option on the command line. If the -g option appears without the -O option, optimization is disabled.
-outdir \<dirname\>	Place all output files in folder \<dirname\>.

Option	Description
-output <resultname>	Create an executable named <resultname>. An appropriate executable extension is automatically appended. Overrides the mbuild default executable naming mechanism.
-regsvr	(Windows only) Use the regsvr32 program to register the resulting shared library at the end of compilation. MATLAB Compiler uses this option whenever it produces a COM or .NET wrapper file.
-setup	Interactively specify the compiler options file to use as the default for future invocations of mbuild by placing it in the user profile folder (returned by the prefdir command). When this option is specified, no other command line input is accepted.
-U<name>	Remove any initial definition of the C preprocessor symbol <name>. (Inverse of the -D option.)
-v	Verbose mode. Print the values for important internal variables after the options file is processed and all command line arguments are considered. Prints each compile step and final link step fully evaluated.
<name>=<value>	Supplement or override an options file variable for variable <name>. This option is processed after the options file is processed and all command line arguments are considered. You may need to use the shell's quoting syntax to protect characters such as spaces that have a meaning in the shell syntax. On Windows double quotes are used (e.g., COMPFLAGS="opt1 opt2"), and on UNIX single quotes are used (e.g., CFLAGS='opt1 opt2').

Option	Description
	It is common to use this option to supplement a variable already defined. To do this, refer to the variable by prepending a $ (e.g., `COMPFLAGS="$COMPFLAGS opt2"` on Windows or `CFLAGS='$CFLAGS opt2'` on UNIX).

Caution

In recent releases,mbuild's functionality has been all but eclipsed by `mcc` and `deploytool`. Avoid using `mbuild` in your long-term development plans, as it is likely to be obsoleted in coming releases.

Some of these options (`-f`, `-g`, and `-v`) are available on the `mcc` command line and are passed along to `mbuild`. Others can be passed along using the `-M` option to `mcc`. For details on the `-M` option, see the `mcc` reference page.

`mbuild` can also create shared libraries from C source code. If a file with the extension `.exports` is passed to MBUILD, a shared library is built. The `.exports` file must be a text file, with each line containing either an exported symbol name, or starting with a `#` or `*` in the first column (in which case it is treated as a comment line). If multiple `.exports` files are specified, all symbol names in all specified `.exports` files are exported.

Note On Windows platforms, at either the MATLAB prompt or the DOS prompt, use double quotes (") when specifying command-line overrides with `mbuild`. For example:

```
mbuild -v COMPFLAGS="$COMPFLAGS -Wall"
            LINKFLAGS="$LINKFLAGS /VERBOSE" yprime.c
```

At the MATLAB command line on UNIX platforms, (") when specifying command-line overrides with `mbuild`. Use the backslash (\) escape character before the dollar sign ($). For example:

```
mbuild -v CFLAGS="\$CFLAGS -Wall"
            LDFLAGS="\$LDFLAGS-w" yprime.c
```

At the shell command line on UNIX platforms, use single quotes (').
For example:

```
 mbuild -v CFLAGS='$CFLAGS -Wall'
            LDFLAGS='$LDFLAGS -w' yprime.c
```

Examples   To set up or change the default C/C++ compiler for use with MATLAB Compiler, use

```
mbuild -setup
```

To compile and link an external C program `foo.c` against `libfoo`, use

```
mbuild foo.c -L. -lfoo (on UNIX)
mbuild foo.c libfoo.lib (on Windows)
```

This assumes both `foo.c` and the library generated above are in the current working folder.

mcc

Purpose	Invoke MATLAB Compiler
Syntax	`mcc [-options] mfile1  [mfile2 ... mfileN]` `                                 [C/C++file1 ... C/C++fileN]`
Description	`mcc` is the MATLAB command that invokes MATLAB Compiler. You can issue the `mcc` command either from the MATLAB command prompt (MATLAB mode) or the DOS or UNIX command line (standalone mode). `mcc` prepares MATLAB file(s) for deployment outside of the MATLAB environment, generates wrapper files in C or C++, optionally builds standalone binary files, and writes any resulting files into the current folder, by default. If more than one MATLAB file is specified on the command line, MATLAB Compiler generates a C or C++ function for each MATLAB file. If C or object files are specified, they are passed to `mbuild` along with any generated C files. `mcc` assumes all input variables are strings, unless otherwise specified.

Note **Using `mcc` in Function Mode** — It is possible to use the `mcc` command in Function Mode by enclosing each `mcc` argument with single-quotes (').

Options

-a Add to Archive

Add a file to the CTF archive. Use

```
-a filename
```

to specify a file to be directly added to the CTF archive. Multiple -a options are permitted. MATLAB Compiler looks for these files on the MATLAB path, so specifying the full path name is optional. These files are not passed to mbuild, so you can include files such as data files.

If only a folder name is included with the -a option, the entire contents of that folder are added recursively to the CTF archive. For example:

```
mcc -m hello.m -a ./testdir
```

In this example, testdir is a folder in the current working folder. All files in testdir, as well as all files in subfolders of testdir, are added to the CTF archive, and the folder subtree in testdir is preserved in the CTF archive.

If a wildcard pattern is included in the file name, only the files in the folder that match the pattern are added to the CTF archive and subfolders of the given path are not processed recursively. For example:

```
mcc -m hello.m -a ./testdir/*
```

In this example, all files in ./testdir are added to the CTF archive and subfolders under ./testdir are not processed recursively.

```
mcc -m hello.m -a ./testdir/*.m
```

In this example, all files with the extension .m under ./testdir are added to the CTF archive and subfolders of ./testdir are not processed recursively.

All files added to the CTF archive using -a (including those that match a wildcard pattern or appear under a folder specified using -a) that do not appear on the MATLAB path at the time of compilation will cause a path entry to be added to the deployed application's run-time path

so that they will appear on the path when the deployed application or component is executed.

When files are included, the absolute path for the DLL and header files is changed. The files are placed in the `.\`*exe*`_mcr\` folder when the CTF file is expanded. The file is not placed in the local folder. This folder gets created from the CTF file the first time the EXE file is executed. The `isdeployed` function is provided to help you accommodate this difference in deployed mode.

The `-a` switch also creates a `.auth` file for authorization purposes. It ensures that the executable looks for the DLL- and H-files in the *exe*`_mcr\`*exe* folder.

Caution

If you use the `-a` flag to include a file that is not on the MATLAB path, the folder containing the file will be added to MATLAB's dependency analysis path. As a result, other files from that folder may be included in the compiled application.

Note Currently, `*` is the only supported wildcard.

Note If the `-a` flag is used to include custom Java classes, standalone applications will work without any need to change the `classpath` as long as the Java class is not a member of a package. The same applies for JAR files. However, if the class being added is a member of a package, the MATLAB code will need to make an appropriate call to `javaaddpath` that will update the `classpath` with the parent folder of the package.

-b Generate Excel Compatible Formula Function

Generate a Visual Basic file (`.bas`) containing the Microsoft Excel Formula Function interface to the COM object generated by MATLAB Compiler. When imported into the workbook Visual Basic code, this code allows the MATLAB function to be seen as a cell formula function. This option requires MATLAB Builder EX.

-B Specify Bundle File

Replace the file on the mcc command line with the contents of the specified file. Use

```
-B filename[:<a1>,<a2>,...,<an>]
```

The bundle file filename should contain only mcc command line options and corresponding arguments and/or other file names. The file may contain other -B options. A bundle file can include replacement parameters for Compiler options that accept names and version numbers. See "Using Bundle Files to Build MATLAB Code" on page 6-9 for a list of the bundle files included with MATLAB Compiler.

-c Generate C Code Only

When used with a macro option, generate C wrapper code but do not invoke `mbuild`, i.e., do not produce a standalone application. This option is equivalent to the defunct `-T codegen` placed at the end of the `mcc` command line.

-C Do Not Embed CTF Archive by Default

Overrides automatically embedding the CTF archive in C/C++ and
main/Winmain shared libraries and standalone binaries by default. See
"MCR Component Cache and CTF Archive Embedding" on page 6-14
for more information.

-d Specified Directory for Output

Place output in a specified folder. Use

```
-d directory
```

to direct the output files from the compilation to the folder specified by the -d option.

-e Suppress MS-DOS Command Window

Suppress appearance of the MS-DOS command window when generating a standalone application. Use -*e* in place of the -*m* option. This option is available for Windows only. Use with -R option to generate error logging as such:

```
 mcc -e -R -logfile -R 'filename' -v function_name
```

or:

```
mcc -e -R '-logfile,logfilename' -v function_name
```

For example, to build a Windows standalone from function foo.m that suppresses the MS-DOS command window, and specifying error logging to a text file, enter this command:

```
mcc -e -R '-logfile,errorlog.txt' -v foo.m
```

You can suppress the MS-DOS command window when using deploytool by creating a Windows Standalone Application. For information about creating a Windows Standalone Application, open the Deployment Tool **Help** by clicking the Actions icon (⚙▾).

This macro is equivalent to the defunct:

```
-W WinMain -T link:exe
```

Note This feature requires the application to successfully compile with a Microsoft Compiler (such as that offered with the free Microsoft Visual Studio Express).

-f Specified Options File

Override the default options file with the specified options file. Use

```
-f filename
```

to specify `filename` as the options file when calling `mbuild`. This option allows you to use different ANSI compilers for different invocations of MATLAB Compiler. This option is a direct pass-through to the `mbuild` script.

Note MathWorks recommends that you use `mbuild -setup`.

-g Generate Debugging Information

Include debugging symbol information for the C/C++ code generated by MATLAB Compiler. It also causes `mbuild` to pass appropriate debugging flags to the system C/C++ compiler. The debug option enables you to backtrace up to the point where you can identify if the failure occurred in the initialization of MCR, the function call, or the termination routine. This option does not allow you to debug your MATLAB files with a C/C++ debugger.

-G Debug Only

Same as -g.

-I Add Directory to Include Path

Add a new folder path to the list of included folders. Each -I option adds a folder to the beginning of the list of paths to search. For example,

```
-I <directory1> -I <directory2>
```

would set up the search path so that directory1 is searched first for MATLAB files, followed by directory2. This option is important for standalone compilation where the MATLAB path is not available.

-K Preserve Partial Output Files

Directs mcc to not delete output files if the compilation ends prematurely, due to error.

mcc's default behavior is to dispose of any partial output if the command fails to execute successfully.

-l Generate a Function Library

Macro to create a function library. This option generates a library wrapper function for each MATLAB file on the command line and calls your C compiler to build a shared library, which exports these functions. You must supply the name of the library (*foo* in the following example).

Using

```
mcc -l foo.m
```

is equivalent to using:

```
mcc -W lib:foo -T link:lib foo.m
```

-m Generate a Standalone Application

Macro to produce a standalone application. This macro is equivalent to the defunct:

```
-W main -T link:exe
```

Use the `-e` option instead of the `-m` option to generate a standalone application while suppressing the appearance of the MS-DOS Command Window.

Note Using the `-e` option requires the application to successfully compile with a Microsoft Compiler (such as that offered with the free Microsoft Visual Studio Express).

-M Direct Pass Through

Define compile-time options. Use

```
-M string
```

to pass `string` directly to the `mbuild` script. This provides a useful mechanism for defining compile-time options, e.g., `-M "-Dmacro=value"`.

Note Multiple `-M` options do not accumulate; only the rightmost `-M` option is used.

-N Clear Path

Passing -N effectively clears the path of all folders except the following core folders (this list is subject to change over time):

- *matlabroot*\toolbox\matlab

- *matlabroot*\toolbox\local

- *matlabroot*\toolbox\compiler\deploy

It also retains all subfolders of the above list that appear on the MATLAB path at compile time. Including -N on the command line allows you to replace folders from the original path, while retaining the relative ordering of the included folders. All subfolders of the included folders that appear on the original path are also included. In addition, the -N option retains all folders that the user has included on the path that are not under *matlabroot*\toolbox.

-o Specify Output Name

Specify the name of the final executable (standalone applications only). Use

```
-o outputfile
```

to name the final executable output of MATLAB Compiler. A suitable, possibly platform-dependent, extension is added to the specified name (e.g., `.exe` for Windows standalone applications).

-p Add Directory to Path

Used in conjunction with required option -N to add specific folders (and subfolders) under *matlabroot*\toolbox to the compilation MATLAB path in an order sensitive way. Use the syntax:

```
-N -p directory
```

where directory is the folder to be included. If directory is not an absolute path, it is assumed to be under the current working folder. The rules for how these folders are included are

- If a folder is included with -p that is on the original MATLAB path, the folder and all its subfolders that appear on the original path are added to the compilation path in an order-sensitive context.

- If a folder is included with -p that is not on the original MATLAB path, that folder is not included in the compilation. (You can use -I to add it.)

If a path is added with the -I option while this feature is active (-N has been passed) and it is already on the MATLAB path, it is added in the order-sensitive context as if it were included with -p. Otherwise, the folder is added to the head of the path, as it normally would be with -I.

-R Run-Time

Provide MCR run-time options. Use the syntax

-R *option*

to provide one of these run-time options.

Option	Description
-logfile *filename*	Specify a log file name.
-nodisplay	Suppress the MATLAB nodisplay run-time warning.
-nojvm	Do not use the Java Virtual Machine (JVM).
-startmsg	Customizable user message displayed at MCR initialization time. See "Displaying MCR Initialization Start-Up and Completion Messages For Users" on page 5-36.
-completemsg	Customizable user message displayed when MCR initialization is complete. See "Displaying MCR Initialization Start-Up and Completion Messages For Users" on page 5-36.

See "Best Practices" on page 5-38 for information about using mcc -R with initialization messages.

Note The -R option is available only for standalone applications. To override MCR options in the other MATLAB Compiler targets, use the mclInitializeApplication and mclTerminateApplication functions. For more information on these functions, see "Calling a Shared Library" on page 8-12.

Caution

When running on Mac, if `-nodisplay` is used as one of the
options included in `mclInitializeApplication`, then the call to
`mclInitializeApplication` must occur before calling `mclRunMain`.

-S Create Singleton MCR

Create a singleton MCR when compiling a COM object. Each instance of the component uses the same MCR. Requires MATLAB Builder NE.

-T Specify Target Stage

Specify the output target phase and type.

Use the syntax -T *target* to define the output type. Valid target values are as follows:

Target	Description
codegen	Generates a C/C++ wrapper file. The default is codegen.
compile:exe	Same as codegen plus compiles C/C++ files to object form suitable for linking into a standalone application.
compile:lib	Same as codegen plus compiles C/C++ files to object form suitable for linking into a shared library/DLL.
link:exe	Same as compile:exe plus links object files into a standalone application.
link:lib	Same as compile:lib plus links object files into a shared library/DLL.

-u Register COM Component for Current User

Registers COM component for the current user only on the development machine. The argument applies only for generic COM component and Microsoft Excel add-in targets only.

-v Verbose

Display the compilation steps, including:

- MATLAB Compiler version number

- The source file names as they are processed

- The names of the generated output files as they are created

- The invocation of mbuild

The -v option passes the -v option to mbuild and displays information about mbuild.

-w Warning Messages

Displays warning messages. Use the syntax

```
-w option [:<msg>]
```

to control the display of warnings. This table lists the valid syntaxes.

Syntax	Description
-w list	Generates a table that maps `<string>` to warning message for use with `enable`, `disable`, and `error`. Appendix C, "Error and Warning Messages", lists the same information.
-w enable	Enables complete warnings.
-w disable[:*<string>*]	Disables specific warning associated with *<string>*. Appendix C, "Error and Warning Messages", lists the valid *<string>* values. Leave off the optional *<string>* to apply the `disable` action to all warnings.
-w enable[:*<string>*]	Enables specific warning associated with *<string>*. Appendix C, "Error and Warning Messages", lists the valid *<string>* values. Leave off the optional *<string>* to apply the `enable` action to all warnings.
-w error[:*<string>*]	Treats specific warning associated with *<string>* as error. Leave off the optional *<string>* to apply the `error` action to all warnings.

Syntax	Description
`-w off[:<string>]` `[<filename>]`	Turns warnings off for specific error messages defined by `<string>`. You can also narrow scope by specifying warnings be turned off when generated by specific `<filename>`s.
`-w on[:<string>]` `[<filename>]`	Turns warnings on for specific error messages defined by `<string>`. You can also narrow scope by specifying warnings be turned on when generated by specific `<filename>`s.

It is also possible to turn warnings `on` or `off` in your MATLAB code.

For example, to turn warnings off for deployed applications (specified using isdeployed) in your `startup.m`, you write:

```
if isdeployed
    warning off
end
```

To turn warnings on for deployed applications, you write:

```
if isdeployed
    warning on
end
```

-W Wrapper Function

Controls the generation of function wrappers. Use the syntax

`-W type`

to control the generation of function wrappers for a collection of MATLAB files generated by MATLAB Compiler. You provide a list of functions and MATLAB Compiler generates the wrapper functions and any appropriate global variable definitions. This table shows the valid options.

Type	Description
main	Produces a POSIX shell `main()` function.
lib:<string>	Creates a C interface and produces an initialization and termination function for use when compiling this compiler generated code into a larger application. This option also produces a header file containing prototypes for all public functions in all MATLAB files specified. <string> becomes the base (file) name for the generated C/C++ and header file. Creates a `.exports` file that contains all nonstatic function names.
cpplib:<string>	Creates a C++ interface and produces an initialization and termination function for use when compiling this compiler generated code into a larger application. This option also produces a header file containing prototypes for all public functions in all MATLAB files specified. <string> becomes the base (file) name for the generated C/C++ and header file. Creates a `.exports` file that contains all nonstatic function names.
none	Does not produce a wrapper file. The default is `none`.

-win32 Run in 32-Bit Mode

Use this option to build a 32-bit application on a 64-bit system *only* when the following are both true:

- You use the same MATLAB installation root (*matlabroot*) for both 32-bit and 64-bit versions of MATLAB.

- You are running from a Windows command line (not a MATLAB command line).

-Y License File

Use

```
-Y license.lic
```

to override the default license file with the specified argument.

-? Help Message

Display MATLAB Compiler help at the command prompt.

mclGetLastErrorMessage

Purpose   Last error message from unsuccessful function call

Syntax   `const char* mclGetLastErrorMessage()`

Description   This function returns a function error message (usually in the form of `false` or `-1`).

Example
```
char *args[] = { "-nodisplay" };
if(!mclInitializeApplication(args, 1))
{
    fprintf(stderr,
             An error occurred while initializing: \n %s  ,
            mclGetLastErrorMessage());
    return -1;
}
```

See Also   `mclInitializeApplication | mclTerminateApplication | <library>Initialize[WithHandlers] | <library>Terminate`

Purpose	Retrieve name of log file used by MCR
Syntax	const char* mclGetLogFileName()
Description	Use mclGetLogFileName() to retrieve the name of the log file used by the MCR. Returns a character string representing log file name used by MCR. For more information, see "MCR Startup Options" on page 5-31 in the User's Guide.
Examples	printf("Logfile name : %s\n",mclGetLogFileName());

mclInitializeApplication

Purpose   Set up application state shared by all (future) MCR instances created in current process

Syntax
```
bool
  mclInitializeApplication(const char **options, int count)
```

Description   MATLAB Compiler-generated standalone executables contain auto-generated code to call this function; users of shared libraries must call this function manually. Call only once per process. The function takes an array of strings (possibly of zero length) and a count containing the size of the string array. The string array may contain the following MATLAB command line switches, which have the same meaning as they do when used in MATLAB. :

- -appendlogfile
- -Automation
- -beginfile
- -debug
- -defer
- -display
- -Embedding
- -endfile
- -fork
- -java
- -jdb
- -logfile
- -minimize
- -MLAutomation
- -noaccel
- -nodisplay

- `-noFigureWindows`

- `-nojit`

- `-nojvm`

- `-noshelldde`

- `-nosplash`

- `-r`

- `-Regserver`

- `-shelldde`

- `-student`

- `-Unregserver`

- `-useJavaFigures`

- `-mwvisual`

- `-xrm`

Caution

`mclInitializeApplication` must be called once only per process. Calling `mclInitializeApplication` more than once may cause your application to exhibit unpredictable or undesirable behavior.

Caution

When running on Mac, if `-nodisplay` is used as one of the options included in `mclInitializeApplication`, then the call to `mclInitializeApplication` must occur before calling `mclRunMain`.

Examples   To start all MCRs in a given process with the `-nodisplay` option, for example, use the following code:

mclInitializeApplication

```
const char *args[] = { "-nodisplay" };
if (! mclInitializeApplication(args, 1))
{
  fprintf(stderr,
          An error occurred while initializing: \n %s  ,
          mclGetLastErrorMessage());
  return -1;
}
```

See Also   mclTerminateApplication

How To   • "Initializing and Terminating Your Application with
                mclInitializeApplication and mclTerminateApplication" on page 8-13

mclIsJVMEnabled

Purpose   Determine if MCR was launched with instance of Java Virtual Machine (JVM)

Syntax   `bool mclIsJVMEnabled()`

Description   Use `mclIsJVMEnabled()` to determine if the MCR was launched with an instance of a Java Virtual Machine (JVM). Returns `true` if MCR is launched with a JVM instance, else returns `false`. For more information, see "MCR Startup Options" on page 5-31 in the User's Guide.

Examples   `printf("JVM initialized : %d\n", mclIsJVMEnabled());`

mclIsMCRInitialized

Purpose   Determine if MCR has been properly initialized

Syntax   `bool mclIsMCRInitialized()`

Description   Use `mclIsMCRInitialized()` to determine whether or not the MCR has been properly initialized. Returns `true` if MCR is already initialized; else returns `false`. For more information, see "MCR Startup Options" on page 5-31 in the User's Guide.

Examples   `printf("MCR initialized : %d\n", mclIsMCRInitialized());`

Purpose   Determine if -nodisplay mode is enabled

Syntax   bool mclIsNoDisplaySet()

Description Use mclIsNoDisplaySet() to determine if -nodisplay mode is enabled.
                 Returns true if -nodisplay is enabled, else returns false. For more
                 information, see "MCR Startup Options" on page 5-31 in the User's
                 Guide.

> **Note** Always returns false on Windows systems since the -nodisplay
> option is not supported on Windows systems.

Examples   printf("nodisplay set : %d\n",mclIsNoDisplaySet());

mclRunMain

Purpose	Mechanism for creating identical wrapper code across all compiler platform environments

Syntax	`typedef int (*mclMainFcnType)(int, const char **);`
	
	`int mclRunMain(mclMainFcnType run_main,`
	`                int argc,`
	`                const char **argv)`

run_main

Name of function to execute after MCR set-up code.

argc

Number of arguments being passed to run_main function. Usually, argc is received by application at its main function.

argv

Pointer to an array of character pointers. Usually, argv is received by application at its main function.

Description	As you need to provide wrapper code when creating an application which uses a C or C++ shared library created by MATLAB Compiler, mclRunMain enables you with a mechanism for creating identical wrapper code across all MATLAB Compiler platform environments. mclRunMain is especially helpful in Macintosh OS X environments where a run loop must be created for correct MCR operation. When an OS X run loop is started, if mclInitializeApplication specifies the -nojvm or -nodisplay option, creating a run loop is a straight-forward process. Otherwise, you must create a Cocoa framework. The Cocoa frameworks consist of libraries, APIs, and Runtimes that form the development layer for all of Mac OS X. Generally, the function pointed to by run_main returns with a pointer (return value) to the code that invoked it. When using Cocoa on the

Macintosh, however, when the function pointed to by run_main returns, the MCR calls exit before the return value can be received by the application, due to the inability of the underlying code to get control when Cocoa is shut down.

Caution

You should not use mclRunMain if your application brings up its own full graphical environment.

Note In non-Macintosh environments, mclRunMain acts as a wrapper and doesn't perform any significant processing.

Examples　　Call using this basic structure:

```
int returncode = O;
mclInitializeApplication(NULL,O);
returncode = mclRunMain((mclmainFcn)
                    my_main_function,O,NULL);
```

See Also　　mclInitializeApplication

mclTerminateApplication

Purpose   Close down all MCR-internal application state

Syntax   `bool mclTerminateApplication(void)`

Description   Call this function once at the end of your program to close down all
                  MCR-internal application state. Call only once per process. After
                  you have called this function, you cannot call any further MATLAB
                  Compiler-generated functions or any functions in any MATLAB library.

Caution

`mclTerminateApplication` must be called once only per process.
Calling `mclTerminateApplication` more than once may cause your
application to exhibit unpredictable or undesirable behavior.

Caution

`mclTerminateApplication` will close any visible or invisible figures
before exiting. If you have visible figures that you would like to wait
for, use `mclWaitForFiguresToDie`.

Examples   At the start of your program, call `mclInitializeApplication` to ensure
                  your library was properly initialized:

```
mclInitializeApplication(NULL,O);
if (!libmatrixInitialize()){
    fprintf(stderr,
             An error occurred while initializing: \n %s  ,
            mclGetLastErrorMessage());
    return -1;
}
```

At your program's exit point, call `mclTerminateApplication` to
properly shut the application down:

```
mxDestroyArray(in1); in1=0;
mxDestroyArray(in2); in2 = 0;
mclTerminateApplication();
return 0;
```

See Also   mclInitializeApplication

How To   • "Initializing and Terminating Your Application with
                    mclInitializeApplication and mclTerminateApplication" on page 8-13

mclWaitForFiguresToDie

Purpose Enable deployed applications to process Handle Graphics events, enabling figure windows to remain displayed

Syntax `void mclWaitForFiguresToDie(HMCRINSTANCE instReserved)`

Description Calling `void mclWaitForFiguresToDie` enables the deployed application to process Handle Graphics events.

NULL is the only parameter accepted for the MCR instance (`HMCRINSTANCE instReserved`).

This function can only be called after *library*Initialize has been called and before *library*Terminate has been called.

`mclWaitForFiguresToDie` blocks all open figures. This function runs until no visible figures remain. At that point, it displays a warning if there are invisible figures present. This function returns only when the last figure window is manually closed — therefore, this function should be called after the library launches at least one figure window. This function may be called multiple times.

If this function is not called, any figure windows initially displayed by the application briefly appear, and then the application exits.

Note `mclWaitForFiguresToDie` will block the calling program only for MATLAB figures. It will not block any Java GUIs, ActiveX controls, and other non-MATLAB GUIs unless they are embedded in a MATLAB figure window.

Examples
```
int run_main(int argc, const char** argv)
{
  int some_variable = O;
  if (argc > 1)
    test_to_run = atoi(argv[1]);

    /* Initialize application */
```

```
  if( !mclInitializeApplication(NULL,O) )
   {
 fprintf(stderr,
     An error occurred while
       initializing: \n %s  ,
    mclGetLastErrorMessage());
 return -1;
}

if (test_to_run == 1 || test_to_run == 0)
{
 /* Initialize ax1ks library */
 if (!libax1ksInitialize())
 {
   fprintf(stderr,
      An error occurred while
      initializing: \n %s  ,
     mclGetLastErrorMessage());
   return -1;
 }
}

if (test_to_run == 2 || test_to_run == 0)
{
 /* Initialize simple library */
 if (!libsimpleInitialize())
 {
   fprintf(stderr,
      An error occurred while
      initializing: \n %s  ,
     mclGetLastErrorMessage());
   return -1;
 }
}

/* your code here
```

mclWaitForFiguresToDie

```
                      /* your code here
                      /* your code here
                      /* your code here
                      /*
                      /* Block on open figures */
                       mclWaitForFiguresToDie(NULL);
                      /* Terminate libraries */
                       if (test_to_run == 1 || test_to_run == 0)
                        libax1ksTerminate();
                       if (test_to_run == 2 || test_to_run == 0)
                        libsimpleTerminate();
                      /* Terminate application */
                       mclTerminateApplication();
                          return(0);
                    }
```

How To   • "Terminating Figures by Force In a Console Application" on page 6-25

Purpose   Display version and location information for MCR installer
corresponding to current platform

Syntax   [*INSTALLER_PATH*, *MAJOR*, *MINOR*, *PLATFORM*,
                    LIST] = mcrinstaller;

Description Displays information about available MCR installers using the
format: [*INSTALLER_PATH*, *MAJOR*, *MINOR*, *PLATFORM*, *LIST*] =
mcrinstaller; where:

- *INSTALLER_PATH* is the full path to the installer for the current
  platform.

- *MAJOR* is the major version number of the installer.

- *MINOR* is the minor version number of the installer.

- *PLATFORM* is the name of the current platform (returned by
  COMPUTER(*arch*)).

- *LIST* is a cell array of strings containing the full paths to MCR
  installers for other platforms. This list is non-empty only in a
  multi-platform MATLAB installation.

Note You must distribute the MATLAB Compiler Runtime library
to your end users to enable them to run applications developed with
MATLAB Compiler. Prebuilt MCR installers for all licensed platforms
ship with MATLAB Compiler.

See "Working with the MCR" on page 5-17 for more information about
the MCR installer.

Examples   **Find MCR Installer Locations**

Display locations of MCR Installers for platform. This example shows
output for a win64 system.

mcrinstaller

```
mcrinstaller

The WIN64 MCR Installer, version 7.16, is:
    X:\jobx\clusterc\current\matlab\toolbox\compiler\
                            deploy\win64\MCRInstaller.exe

MCR installers for other platforms are located in:
    X:\jobx\clusterc\current\matlab\toolbox\compiler\
                        deploy\win64
    win64 is the value of COMPUTER(win64) on
            the target machine.

For more information, read your local MCR Installer help.
Or see the online documentation at MathWorks' web site. (Page
                                        may load slowly.)

ans =

X:\jobx\clusterc\current\matlab\toolbox\compiler\
                            deploy\win64\MCRInstaller.exe
```

Purpose   Determine version of installed MATLAB Compiler Runtime (MCR)

Syntax   [major, minor] = mcrversion;

Description   The MCR version number consists of two digits, separated by a decimal point. This function returns each digit as a separate output variable: [major, minor] = mcrversion; Major and minor are returned as integers.

If the version number ever increases to three or more digits, call mcrversion with more outputs, as follows:

[major, minor, point] = mcrversion;

At this time, all outputs past "minor" are returned as zeros.

Typing only mcrversion will return the major version number only.

Examples
```
mcrversion
ans =
     7
```

setmcruserdata

Purpose	Associate MATLAB data value with string key
Syntax	*function* setmcruserdata(*key*, *value*)
Description	The *function* setmcruserdata(*key*, *value*) command is part of the MCR User Data interface API. For information about this function, as well as complete examples of usage, see "Using the MCR User Data Interface" on page 5-34.
	The MATLAB functions getmcruserdata and setmcruserdata can be dragged and dropped (as you would any other MATLAB file), directly to the deploytool GUI.

Examples

```
setmcruserdata('ParallelProfile','config.mat')


mxArray *value = mxCreateString("\usr\userdir\config.mat");
if (!SetMCRUserData("ParallelProfile",
      "\usr\userdir\config.mat") )
{
    fprintf(stderr,
            Could not set MCR user data: \n %s  ,
            mclGetLastErrorMessage());
    return -1;}
```

See Also   getmcruserdata

Purpose   Free all resources allocated by MCR instance associated with *library*

Syntax   void *library*Terminate(void)

Description   This function should be called after you finish calling the functions in this MATLAB Compiler-generated library, but before mclTerminateApplication is called.

Examples   Call libmatrixInitialize to initialize libmatrix library properly near the start of your program:

```
/* Call the library intialization routine and ensure the
*  library was initialized properly. */
if (!libmatrixInitialize())
{
    fprintf(stderr,
             An error occurred while initializing: \n %s  ,
            mclGetLastErrorMessage());
    return -2;
}
else
    ...
```

Near the end of your program (but before calling mclTerminateApplication) free resources allocated by the MCR instance associated with library libmatrix:

```
/* Call the library termination routine */
libmatrixTerminate();
/* Free the memory created */
mxDestroyArray(in1); in1=0;
mxDestroyArray(in2); in2 = 0;
}
```

See Also   <library>Initialize[WithHandlers]

How To   • "Library Initialization and Termination Functions" on page 8-26

<library>Terminate

MATLAB Compiler Quick Reference

Common Uses of MATLAB Compiler

In this section...
"Create a Standalone Application" on page A-2
"Create a Library" on page A-2

Create a Standalone Application

Example 1

To create a standalone application from `mymfile.m`, use

```
mcc -m mymfile
```

Example 2

To create a standalone application from `mymfile.m`, look for `mymfile.m` in the folder `/files/source`, and put the resulting C files and in `/files/target`, use

```
mcc -m -I /files/source -d /files/target mymfile
```

Example 3

To create a standalone application `mymfile1` from `mymfile1.m` and `mymfile2.m` using a single `mcc` call, use

```
mcc -m mymfile1 mymfile2
```

Create a Library

Example 1

To create a C shared library from `foo.m`, use

```
mcc -l foo.m
```

Example 2

To create a C shared library called library_one from foo1.m and foo2.m, use

```
mcc -W lib:library_one -T link:lib foo1 foo2
```

Note You can add the -g option to any of these for debugging purposes.

mcc Command Arguments Listed Alphabetically

Bold entries in the Comment column indicate default values.

Option	Description	Comment
-a *filename*	Add `filename` to the CTF archive.	None
-b	Generate Excel compatible formula function.	Requires MATLAB Builder EX
-B filename[:arg[,arg]]	Replace `-B filename` on the `mcc` command line with the contents of `filename`.	The file should contain only `mcc` command-line options. These are MathWorks included options files: • `-B csharedlib:foo` — C shared library • `-B cpplib:foo` — C++ library
-c	Generate C wrapper code.	Equivalent to `-T codegen`
-C	Directs `mcc` to not embed the CTF archive in C/C++ and main/Winmain shared libraries and standalone binaries by default.	See "MCR Component Cache and CTF Archive Embedding" on page 6-14 for more information.
-d directory	Place output in specified folder.	None

Option	Description	Comment
-e	Suppresses appearance of the MS-DOS Command Window when generating a standalone application.	Use -e in place of the -m option. Available for Windows only. Use with -R option to generate error logging. Equivalent to -W WinMain -T link:exe You can suppress the MS-DOS command window when using deploytool by creating a Windows Standalone Application. For information about creating a Windows Standalone Application, open the Deployment Tool **Help** by clicking the Actions icon (⚙▾).
-f filename	Use the specified options file, filename, when calling mbuild.	mbuild -setup is recommended.
-g	Generate debugging information.	None
-G	Same as -g	None
-I directory	Add folder to search path for MATLAB files.	MATLAB path is automatically included when running from MATLAB, but not when running from a DOS/UNIX shell.
-K	Directs mcc to not delete output files if the compilation ends prematurely, due to error.	mcc's default behavior is to dispose of any partial output if the command fails to execute successfully.
-l	Macro to create a function library.	Equivalent to -W lib -T link:lib
-m	Macro to generate a standalone application.	Equivalent to -W main -T link:exe
-M string	Pass string to mbuild.	Use to define compile-time options.

Option	Description	Comment
-N	Clear the path of all but a minimal, required set of folders.	None
-o outputfile	Specify name of final output file.	Adds appropriate extension
-p directory	Add directory to compilation path in an order-sensitive context.	Requires -N option

-R *option*	Specify run-time options for MCR.	*option* =
		    -nojvm
		    -nodisplay
		    -logfile *filename*
		    -startmsg
		    -completemsg *filename*

-S	Create Singleton MCR.	For COM components only. Requires MATLAB Builder NE or MATLAB Builder EX.
-u	Registers COM component for current user only on development machine	Valid only for generic COM components and Microsoft Excel add-ins (requiring MATLAB Builder EX)
-T	Specify the output target phase and type.	Default is codegen.
-v	Verbose; display compilation steps.	None

-w *option*	Display warning messages.	*option* = list
		level
		level:*string*
		where *level* = disable
		             enable
		             error
		             error
		[off:*string* | on:*string*]

Option	Description	Comment

-W *type*	Control the generation of function wrappers.	*type* = main
		    cpplib:\<string\>
		    lib:\<string\>
		        none
		    com:compname,clname,version

-Y licensefile	Use licensefile when checking out a MATLAB Compiler license.	None
-?	Display help message.	None

mcc Command Line Arguments Grouped by Task

Bold entries in the Comment column indicate default values.

COM Components

Option	Description	Comment
-u	Registers COM component for current user only on development machine	Valid only for generic COM components and Microsoft Excel add-ins (requiring MATLAB Builder EX)

CTF Archive

Option	Description	Comment
-a *filename*	Add filename to the CTF archive.	None
-C	Directs mcc to not embed the CTF archive in C/C++ and main/Winmain shared libraries and standalone binaries by default.	See "MCR Component Cache and CTF Archive Embedding" on page 6-14 for more information.

Debugging

Option	Description	Comment
-g	Generate debugging information.	None
-G	Same as -g	None
-K	Directs mcc to not delete output files if the compilation ends prematurely, due to error.	mcc's default behavior is to dispose of any partial output if the command fails to execute successfully.
-v	Verbose; display compilation steps.	None
-W *type*	Control the generation of function wrappers.	*type* = main cpplib:<string> lib:<string> **none** com:compname, clname,version
-?	Display help message.	None

Dependency Function (depfun) Processing

Option	Description	Comment
-a *filename*	Add filename to the CTF archive.	None

Licenses

Option	Description	Comment
-Y licensefile	Use licensefile when checking out a MATLAB Compiler license.	None

MATLAB Builder EX

Option	Description	Comment
-b	Generate Excel compatible formula function.	Requires MATLAB Builder EX
-u	Registers COM component for current user only on development machine	Valid only for generic COM components and Microsoft Excel add-ins (requiring MATLAB Builder EX)

MATLAB Path

Option	Description	Comment
-I directory	Add folder to search path for MATLAB files.	MATLAB path is automatically included when running from MATLAB, but not when running from a DOS/UNIX shell.
-N	Clear the path of all but a minimal, required set of folders.	None
-p directory	Add directory to compilation path in an order-sensitive context.	Requires -N option

mbuild

Option	Description	Comment
`-f filename`	Use the specified options file, `filename`, when calling `mbuild`.	`mbuild -setup` is recommended.
`-M string`	Pass string to `mbuild`.	Use to define compile-time options.

MATLAB Compiler Runtime (MCR)

Option	Description	Comment

-R *option*	Specify run-time options for MCR.	*option* =
		        -nojvm
		        -nodisplay
		        -logfile *filename*
		        -startmsg
		        -completemsg *filename*

-S	Create Singleton MCR.	Requires MATLAB Builder NE

Override Default Inputs

Option	Description	Comment
`-B filename[:arg[,arg]]`	Replace `-B filename` on the `mcc` command line with the contents of `filename` (bundle).	The file should contain only `mcc` command-line options. These are MathWorks included options files: • `-B csharedlib:foo` — C shared library • `-B cpplib:foo` — C++ library

Override Default Outputs

Option	Description	Comment
-d directory	Place output in specified folder.	None
-o outputfile	Specify name of final output file.	Adds appropriate extension
-e	Suppresses appearance of the MS-DOS Command Window when generating a standalone application.	Use -e in place of the -m option. Available for Windows only. Use with -R option to generate error logging. Equivalent to -W WinMain -T link:exe
		You can suppress the MS-DOS command window when using deploytool by creating a Windows Standalone Application. For information about creating a Windows Standalone Application, open the Deployment Tool **Help** by clicking the Actions icon ( ).

Wrappers and Libraries

Option	Description	Comment
-c	Generate C wrapper code.	Equivalent to -T codegen
-l	Macro to create a function library.	Equivalent to -W lib -T link:lib
-m	Macro to generate a standalone application.	Equivalent to -W main -T link:exe

-W *type*	Control the generation of function wrappers.	*type* = main
		    cpplib:<string>
		    lib:<string>
		        none
		    com:compname,
		clname,version

Using MATLAB Compiler on Mac or Linux

Overview

If you use MATLAB Compiler on Linux or Macintosh systems, use this appendix as a quick reference to common tasks.

Installing MATLAB Compiler on Mac or Linux

In this section...
"Installing MATLAB® Compiler™" on page B-3
"Selecting Your gcc Compiler" on page B-3
"Custom Configuring Your Options File" on page B-3
"Install Apple Xcode from DVD on Maci64" on page B-3

Installing MATLAB Compiler

See "Supported ANSI C and C++ UNIX Compilers" on page 2-3 for general installation instructions and information about supported compilers.

Selecting Your gcc Compiler

Run `mbuild -setup` to select your `gcc` compiler . See the "UNIX" on page 2-9 configuration instructions for more information about running `mbuild` .

Custom Configuring Your Options File

To modify the current linker settings, or disable a particular set of warnings, locate your options file for your "UNIX Operating System" on page 2-11, and view instructions for "Changing the Options File" on page 2-11.

Install Apple Xcode from DVD on Maci64

When installing on 64-bit Macintosh systems, install the Apple Xcode from the installation DVD.

Writing Applications for Mac or Linux

In this section...
"Objective-C/C++ Applications for Apple's Cocoa API" on page B-4
"Where's the Example Code?" on page B-4
"Preparing Your Apple Xcode Development Environment" on page B-4
"Build and Run the Sierpinski Application" on page B-5
"Running the Sierpinski Application" on page B-7

Objective-C/C++ Applications for Apple's Cocoa API

Apple Xcode, implemented in the Objective-C language, is used to develop applications using the Cocoa framework, the native object-oriented API for the Mac OS X operating system.

This article details how to deploy a graphical MATLAB application with Objective C and Cocoa, and then deploy it using MATLAB Compiler.

Where's the Example Code?

You can find example Apple Xcode, header, and project files in *matlabroot*/extern/examples/compiler/xcode.

Preparing Your Apple Xcode Development Environment

To run this example, you should have prior experience with the Apple Xcode development environment and the Cocoa framework.

The example in this article is ready to build and run. However, before you build and run your own applications, you must do the following (as has been done in our example code):

1 Build the shared library with MATLAB Compiler using either the Deployment Tool or mcc.

2 Compile application code against the component's header file and link the application against the component library and `libmwmclmcrrt`. See "Set MCR Paths on Mac or Linux with Scripts" on page B-12 and "Solving Problems Related to Setting MCR Paths on Mac or Linux" on page B-12 for information about and MCR paths and `libmwmclmcrrt`.

3 In your Apple Xcode project:

- Specify `mcc` in the project target (Build Component Library in the example code).

- Specify target settings in `HEADER_SEARCH_PATHS`.

  - Specify directories containing the component header.

  - Specify the path *matlabroot*/extern/include.

  - Define `MWINSTALL_ROOT`, which establishes the install route using a relative path.

- Set `LIBRARY_SEARCH_PATHS` to any directories containing the component's shared library, as well as to the path *matlabroot*/runtime/maci64.

Build and Run the Sierpinski Application

In this example, you deploy the graphical Sierpinski function (`sierpinski.m`, located at *matlabroot*/extern/examples/compiler).

```
function [x, y] = sierpinski(iterations, draw)
% SIERPINSKI Calculate (optionally draw) the points
%  in Sierpinski's triangle

% Copyright 2004 The MathWorks, Inc.

    % Three points defining a nice wide triangle
    points = [0.5 0.9 ; 0.1 0.1 ; 0.9 0.1];

    % Select an initial point
    current = rand(1, 2);

    % Create a figure window
    if (draw == true)
        f = figure;
        hold on;
```

```
    end

    % Pre-allocate space for the results, to improve performance
    x = zeros(1,iterations);
    y = zeros(1,iterations);

    % Iterate
    for i = 1:iterations

        % Select point at random
        index = floor(rand * 3) + 1;

        % Calculate midpoint between current point and random point
        current(1) = (current(1) + points(index, 1)) / 2;
        current(2) = (current(2) + points(index, 2)) / 2;

        % Plot that point
        if draw, line(current(1),current(2));, end
x(i) = current(1);
        y(i) = current(2);

    end

    if (draw)
        drawnow;
    end
```

1 Using the Mac Finder, locate the Apple Xcode project
(*matlabroot*/extern/examples/compiler/xcode). Copy files to
a working directory to run this example, if needed.

2 Open sierpinski.xcodeproj. The development environment starts.

3 In the **Groups and Files** pane, select **Targets**.

4 Click **Build and Run**. The make file runs that launches MATLAB
Compiler (mcc).

Running the Sierpinski Application

Run the **Sierpinski** application from the build output directory. The following GUI appears:
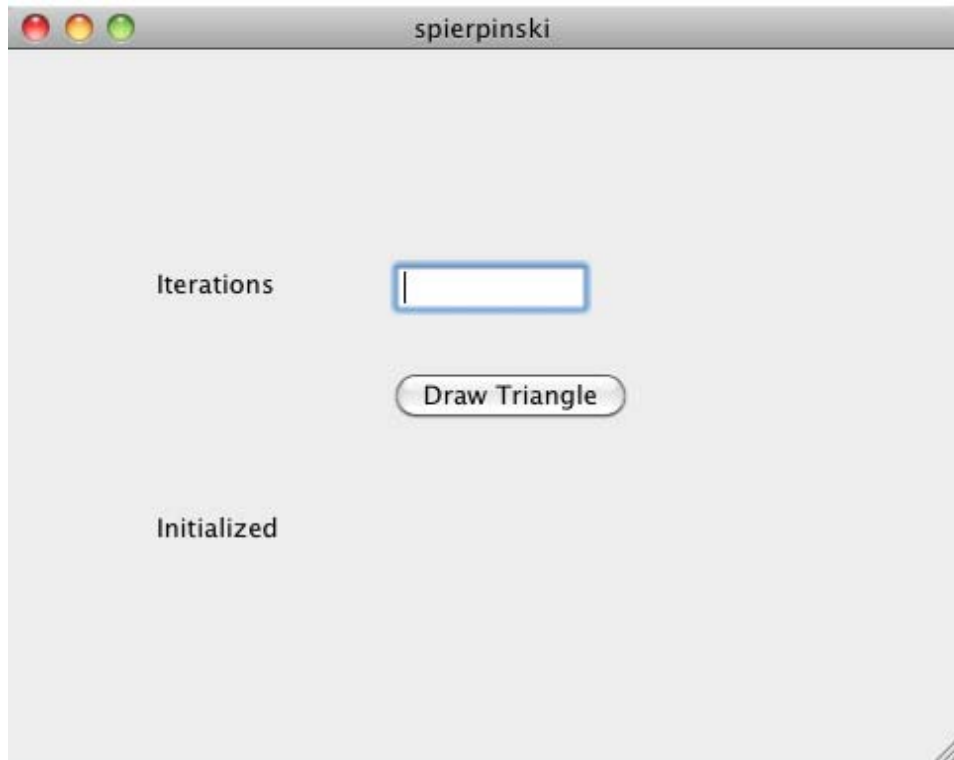
MATLAB Sierpinski Function Implemented in the Mac Cocoa Environment

   1 In the **Iterations** field, enter an integer such as 10000:
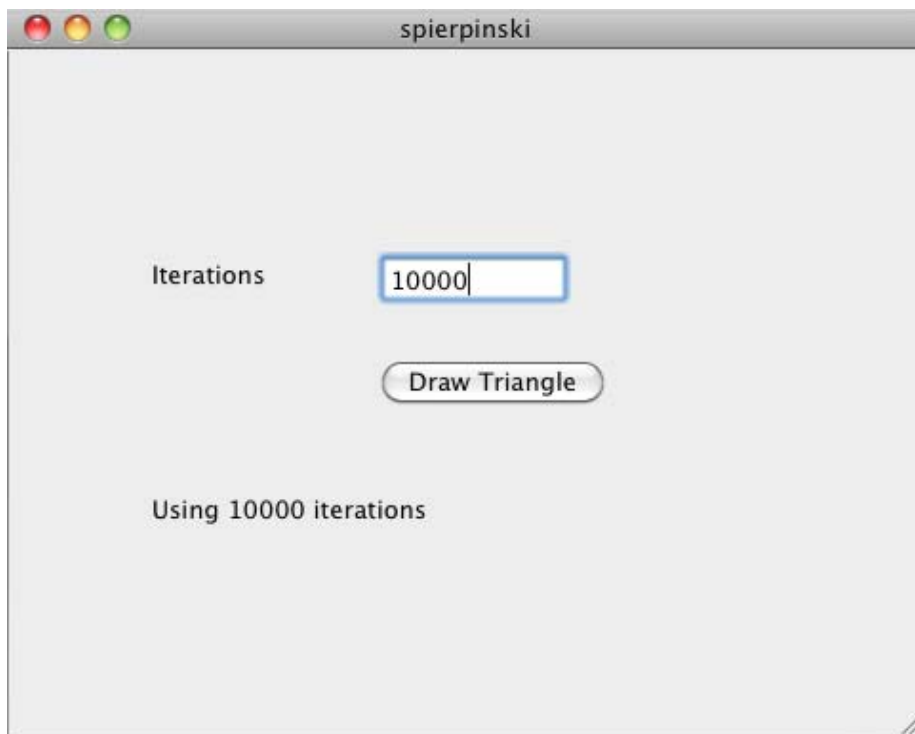
2 Click **Draw Triangle**. The following figure appears:
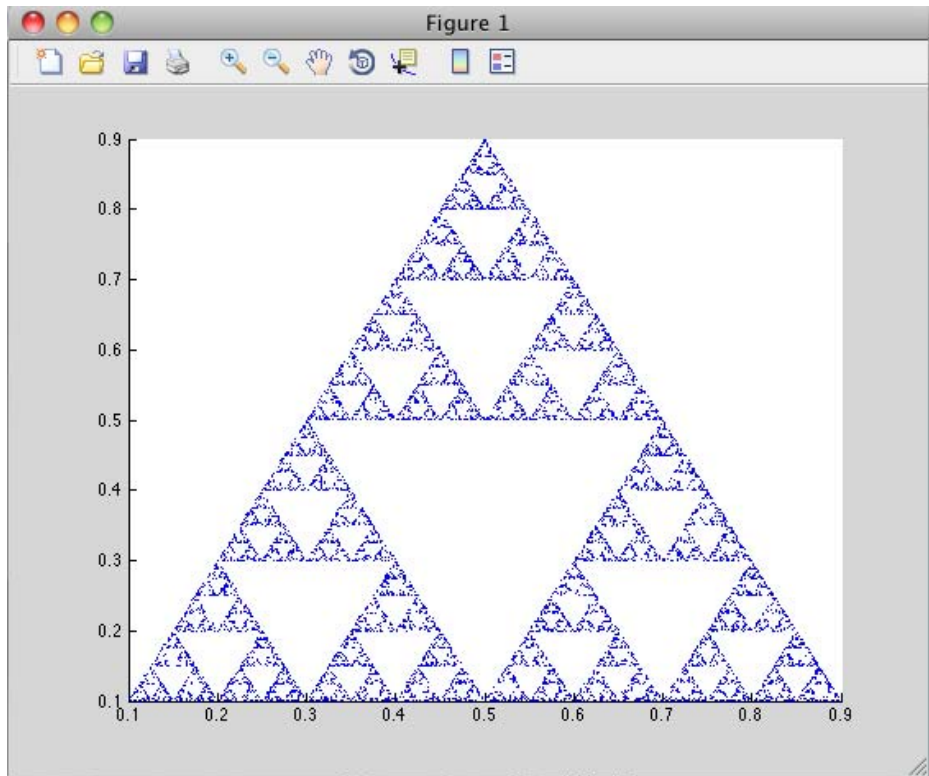
Building Your Application on Mac or Linux

In this section...
"Compiling Your Application with the Deployment Tool" on page B-10
"Compiling Your Application with the Command Line" on page B-10

Compiling Your Application with the Deployment Tool

When running a graphical interface (such as XWindows) from your Mac or Linux desktop, use as an end-to-end template for building a standalone or shared library with the Deployment Tool (`deploytool`).

See "Using the Deployment Tool from the Command Line" on page 4-6 for information on invoking `deploytool` from the command line.

Compiling Your Application with the Command Line

For compiling your application at the command line, there are separate Macintosh and non-Macintosh instructions for Mac or Linux platforms.

On Non-Maci64 Platforms

Use the section"Input and Output Files" on page 4-8 for lists of files produced and supplied to `mcc` when building a "Standalone Executable" on page 4-8, "C Shared Library" on page 4-9, or "C++ Shared Library" on page 4-11.

On Maci64

Use the section "Input and Output Files" on page 4-8 for lists of files produced and supplied to `mcc` when building a "Macintosh 64 (Maci64)" on page 4-13 application.

Testing Your Application on Mac or Linux

When you test your application, understand that deployed applications in the Windows environment automatically modify the system `PATH` variable.

On , however, you perform this step manually, based on what type of operating system you use. Refer to for details.

Running Your Application on Mac or Linux

In this section...
"Installing the MCR on Mac or Linux" on page B-12
"Set MCR Paths on Mac or Linux with Scripts" on page B-12
"Running Applications on Linux Systems with No Display Console" on page B-14

Installing the MCR on Mac or Linux

See "Working with the MCR" on page 5-17 for complete information on installing the MCR.

See "The MCR Installer" on page 5-18 for details about how to run deployed components against specific MCR installations.

Performing a Silent Installation of the MCR on Mac or Linux

See the MATLAB Installation Guide for information on silent install and other command-line options for installing the MCR.

Set MCR Paths on Mac or Linux with Scripts

When you build applications, associated shell scripts (run_*application*.sh) are automatically generated in the same folder as your binary. By running these scripts, you can conveniently set the path to your MCR location.

These paths can be found in the article .

Solving Problems Related to Setting MCR Paths on Mac or Linux

Use the following to solve common problems and issues:

I tried running SETENV on Mac and the command failed

If the setenv command fails with a message similar to setenv: command not found or setenv: not found, you may not be using a C Shell command interpreter (such as csh or tcsh).

Instead of running the commands above, which are in the format of setenv *my_variable my_value*, use the command format *my_variable=my_value* ; export *my_variable*.

For example, to set DYLD_LIBRARY_PATH, run the following command:

```
export DYLD_LIBRARY_PATH = mcr_root/v711/runtime/maci64:mcr_root/
...
```

My Mac application fails with "Library not loaded" or "Image not found" even though my EVs are set

If you set your environment variables, you may still receive the following message when you run your application:

```
imac-joe-user:~ joeuser$ /Users/joeuser/Documents/MATLAB/Dip/Dip ; exit;
dyld: Library not loaded: @loader_path/libmwmclmcrrt.7.11.dylib
Referenced from: /Users/joeuser/Documents/MATLAB/Dip/Dip
Reason: image not found
Trace/BPT trap
logout
```

You may have set your environment variables initially, but they were not set up as persistent variables. Do the following:

1 In your home directory, open a file such as .bashrc or .profile file in your log-in shell.

2 In either of these types of log-in shell files, add commands to set your environment variables so that they persist. For example, to set DYLD_LIBRARY_PATH or XAPPLRESDIR in this manner, you enter the following in your file:

```
# Setting PATH for MCR

DYLD_LIBRARY_PATH=/Users/joeuser/Desktop/mcr/v711/runtime/maci64:
/Users/joeuser/Desktop/mcr/v711/sys/os/maci64:/Users/joeuser/Desktop/
mcr//v711/bin/maci64:/System/Library/Frameworks/JavaVM.framework/
JavaVM:/System/Library/Frameworks/JavaVM.framework/Libraries
export DYLD_LIBRARY_PATH
```

B-13

?

> **Note** The DYLD_LIBRARY_PATH= statement is one statement that must be entered as a single line. The statement is shown on different lines, in this example, for readability only.

Running Applications on Linux Systems with No Display Console

Your compiled MATLAB applications should execute normally on a computer with no display, providing there is some graphical environment installed.

To test whether a compiled application will run on a system with no console, attempt to install and run the MATLAB Compiler Runtime.

Run Your 64-Bit Mac Application

Overview

64-bit Macintosh graphical applications, launched through the Mac OS X finder utility, require additional configuration if MATLAB software or the MCR were not installed in default locations.

Installing the Macintosh Application Launcher Preference Pane

Install the Macintosh Application Launcher preference pane, which gives you the ability to specify your installation area.

1 In the Mac OS X Finder, navigate to *install_area*/toolbox/compiler/maci64.

2 Double-click on **MW_App_Launch.prefPane**.

Note to **Administrators:** The Macintosh Application Launcher manages only *user* preference settings. If you copy the preferences defined in the launcher to the Macintosh System Preferences area, the preferences are still manipulated in the User Preferences area.

Configuring the Installation Area

Once the preference pane is installed, you configure the installation area.

1 Launch the preference pane by clicking on the apple logo in the upper left corner of the desktop.

2 Click on **System Preferences**. The MW_App_Launch preference pane appears in the **Other** area.
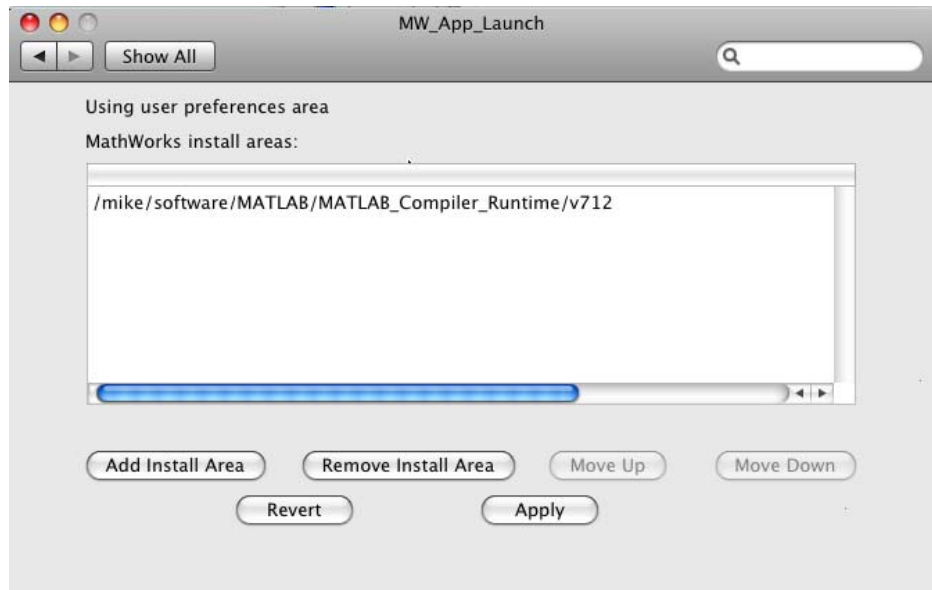
The Macintosh Application Launcher

3 Click **Add Install Area** to define an installation area on your system.

4 Define the default installation path by browsing to it.

5 Click **Open**.
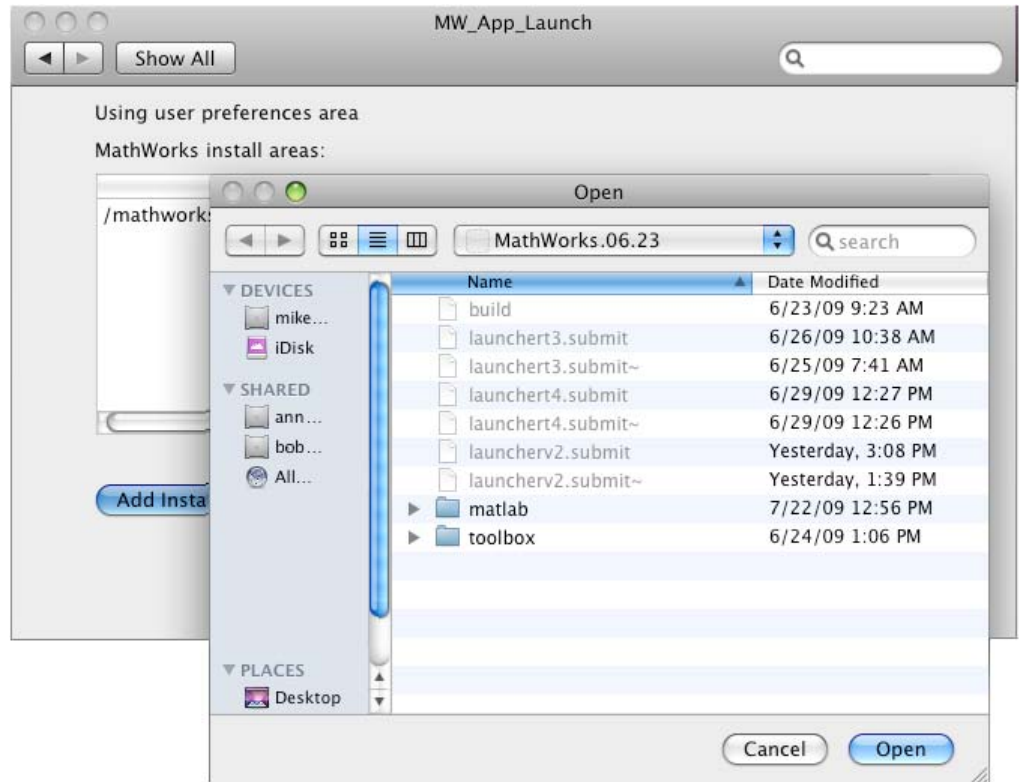
Modifying Your Installation Area

Occasionally, you remove an installation area, define additional areas or change the order of installation area precedence.

You can use the following options in MathWorks Application Launcher to modify your installation area:

- **Add Install Area** — Defines the path on your system where your applications install by default.

- **Remove Install Area** — Removes a previously defined installation area.

- **Move Up** — After selecting an installation area, click this button to move the defined path up the list. Binaries defined in installation areas at the top of the list have precedence over all succeeding entries.

- **Move Down** — After selecting an installation area, click this button to move the defined path down the list. Binaries defined in installation areas at the top of the list have precedence over all succeeding entries.

- **Apply** — Saves changes and exits MathWorks Application Launcher.

- **Revert** — Exits MathWorks Application Launcher without saving any changes.

Launching the Application

When you create a 64-bit Macintosh application, a Macintosh bundle is created. If the application does not require standard input and output, launch the application by clicking on the bundle in the Mac OS X Finder utility.

The location of the bundle is determined by whether you use `mcc` or `deploytool` to build the application:

- If you use `deploytool`, the application bundle is placed in the compiled application's `distrib` directory. Use `deploytool` to package your application. See for more details. Place the resulting archive file anywhere on the desktop.

- If you use `mcc`, the application bundle is placed in the current working directory or in the output directory as specified by the `mcc` "-o Specify Output Name" on page 12-38 switch.

Error and Warning Messages

About Error and Warning Messages

This appendix lists and describes error messages and warnings generated by MATLAB Compiler. Compile-time messages are generated during the compile or link phase. It is useful to note that most of these compile-time error messages should not occur if the MATLAB software can successfully execute the corresponding MATLAB file.

Use this reference to:

- Confirm that an error has been reported

- Determine possible causes for an error

- Determine possible ways to correct an error

When using MATLAB Compiler, if you receive an internal error message, record the specific message and report it to Technical Support at `http://www.mathworks.com/contact_TS.html`.

Compile-Time Errors

Error: An error occurred while shelling out to mex/mbuild (error code = errorno). Unable to build (specify the -v option for more information). MATLAB Compiler reports this error if `mbuild` or `mex` generates an error.

Error: An error occurred writing to file "filename": reason. The file can not be written. The reason is provided by the operating system. For example, you may not have sufficient disk space available to write the file.

Error: Cannot write file "filename" because MCC has already created a file with that name, or a file with that name was specified as a command line argument. MATLAB Compiler has been instructed to generate two files with the same name. For example:

```
mcc -W lib:liba liba -t  % Incorrect
```

Error: Could not check out a Compiler license. No additional MATLAB Compiler licenses are available for your workgroup.

Error: Initializing preferences required to run the application. The `.ctf` file and the corresponding target (standalone application or shared library) created using MATLAB Compiler do not match. Ensure that the `.ctf` file and the target file are created as output from the same `mcc` command. Verify the time stamp of these files to ensure they were created at the same time. Never combine the `.ctf` file and the target application created during execution of different `mcc` commands.

Error: File: "filename" not found. A specified file can not be found on the path. Verify that the file exists and that the path includes the file's location. You can use the `-I` option to add a folder to the search path.

Error: File: "filename" is a script MATLAB file and cannot be compiled with the current Compiler. MATLAB Compiler cannot compile script MATLAB files. To learn how to convert script MATLAB files to function MATLAB files, see "Converting Script MATLAB Files to Function MATLAB Files" on page 6-20.

Error: File: filename Line: # Column: # A variable cannot be made storageclass1 after being used as a storageclass2. You cannot change a variable's storage class (global/local/persistent). Even though MATLAB allows this type of change in scope, MATLAB Compiler does not.

Error: Found illegal whitespace character in command line option: "string". The strings on the left and right side of the space should be separate arguments to MCC. For example:

```
mcc('-m', '-v', 'hello')% Correct
mcc('-m -v', 'hello')   % Incorrect
```

Error: Improper usage of option -optionname. Type "mcc -?" for usage information. You have incorrectly used a MATLAB Compiler option. For more information about MATLAB Compiler options, see "mcc Command Arguments Listed Alphabetically" on page A-4, or type mcc -? at the command prompt.

Error: libraryname library not found. MATLAB has been installed incorrectly.

Error: No source files were specified (-? for help). You must provide MATLAB Compiler with the name of the source file(s) to compile.

Error: "optionname" is not a valid -option option argument. You must use an argument that corresponds to the option. For example:

```
mcc -W main ... % Correct
mcc -W mex  ... % Incorrect
```

Error: Out of memory. Typically, this message occurs because MATLAB Compiler requests a larger segment of memory from the operating system than is currently available. Adding additional memory to your system can alleviate this problem.

Error: Previous warning treated as error. When you use the -w error option, this error appears immediately after a warning message.

Error: The argument after the -option option must contain a colon.
The format for this argument requires a colon. For more information, see
"mcc Command Arguments Listed Alphabetically" on page A-4, or type mcc
-? at the command prompt.

**Error: The environment variable MATLAB must be set to the MATLAB
root directory.** On UNIX, the MATLAB and LM_LICENSE_FILE variables must
be set. The mcc shell script does this automatically when it is called the first
time.

**Error: The license manager failed to initialize (error code is
errornumber).** You do not have a valid MATLAB Compiler license or no
additional MATLAB Compiler licenses are available.

**Error: The option -option is invalid in modename mode (specify -?
for help).** The specified option is not available.

Error: The specified file "filename" cannot be read. There is a problem
with your specified file. For example, the file is not readable because there
is no read permission.

**Error: The -optionname option requires an argument (e.g.
"proper_example_usage").** You have incorrectly used a MATLAB
Compiler option. For more information about MATLAB Compiler options, see
"mcc Command Arguments Listed Alphabetically" on page A-4, or type mcc
-? at the command prompt.

Error: -x is no longer supported. MATLAB Compiler no longer generates
MEX-files because there is no longer any performance advantage to doing so.
The MATLAB JIT Accelerator makes compilation for speed obsolete.

Error: Unable to open file "filename":<string>. There is a problem with
your specified file. For example, there is no write permission to the output
folder, or the disk is full.

**Error: Unable to set license linger interval (error code is
errornumber).** A license manager failure has occurred. Contact Technical
Support with the full text of the error message.

Error: Unknown warning enable/disable string: warningstring. `-w`
`enable:`, `-w disable:`, and `-w error:` require you to use one of the warning
string identifiers listed in "Warning Messages" on page C-7.

Error: Unrecognized option: -option. The option is not a valid option.
See "mcc Command Arguments Listed Alphabetically" on page A-4, for a
complete list of valid options for MATLAB Compiler, or type `mcc -?` at the
command prompt.

Warning Messages

This section lists the warning messages that MATLAB Compiler can generate. Using the -w option for mcc, you can control which messages are displayed. Each warning message contains a description and the warning message identifier string (in parentheses) that you can enable or disable with the -w option. For example, to produce an error message if you are using a trial MATLAB Compiler license to create your standalone application, you can use:

```
mcc -w error:trial_license -mvg hello
```

To enable all warnings except those generated by the save command, use:

```
mcc -w enable -w disable:trial_license ...
```

To display a list of all the warning message identifier strings, use:

```
mcc -w list -m mfilename
```

For additional information about the -w option, see "mcc Command Arguments Listed Alphabetically" on page A-4.

Warning: File: filename Line: # Column: # The #function pragma expects a list of function names. *(pragma_function_missing_names)* This pragma informs MATLAB Compiler that the specified function(s) provided in the list of function names will be called through an feval call. This will automatically compile the selected functions.

Warning: MATLAB file "filename" was specified on the command line with full path of "pathname", but was found on the search path in directory "directoryname" first. *(specified_file_mismatch)* MATLAB Compiler detected an inconsistency between the location of the MATLAB file as given on the command line and in the search path. MATLAB Compiler uses the location in the search path. This warning occurs when you specify a full path name on the mcc command line and a file with the same base name (file name) is found earlier on the search path. This warning is issued in the following example if the file afile.m exists in both dir1 and dir2:

```
mcc -m -I /dir1 /dir2/afile.m
```

Warning: The file filename was repeated on MATLAB Compiler command line. *(repeated_file)* This warning occurs when the same file name appears more than once on the compiler command line. For example:

```
mcc -m sample.m sample.m % Will generate the warning
```

Warning: The name of a shared library should begin with the letters "lib". "libraryname" doesn't. *(missing_lib_sentinel)* This warning is generated if the name of the specified library does not begin with the letters "lib". This warning is specific to UNIX and does not occur on the Windows operating system. For example:

```
mcc -t -W lib:liba -T link:lib a0 a1 % No warning
mcc -t -W lib:a -T link:lib a0 a1    % Will generate a warning
```

Warning: All warnings are disabled. *(all_warnings)* This warning displays all warnings generated by MATLAB Compiler. This warning is disabled.

Warning: A line has num1 characters, violating the maximum page width (num2). *(max_page_width_violation)* This warning is generated if there are lines that exceed the maximum page width, num2. This warning is disabled.

Warning: The option -optionname is ignored in modename mode (specify -? for help). *(switch_ignored)* This warning is generated if an option is specified on the mcc command line that is not meaningful in the specified mode. This warning is enabled.

Warning: Unrecognized Compiler pragma "pragmaname". *(unrecognized_pragma)* This warning is generated if you use an unrecognized pragma. This warning is enabled.

Warning: "functionname1" is a MEX- or P-file being referenced from "functionname2". *(mex_or_p_file)* This warning is generated if functionname2 calls functionname1, which is a MEX- or P-file. This warning is enabled.

Note A link error is produced if a call to this function is made from standalone code.

Trial Compiler license. The generated application will expire 30 days from today, on date. *(trial_license)* This warning displays the date that the deployed application will expire. This warning is enabled.

depfun Errors

About depfun Errors

MATLAB Compiler uses a dependency analysis (`depfun`) to determine the list of necessary files to include in the CTF package. If this analysis encounters a problem, `depfun` displays an error.

These error messages take the form

```
depfun Error: <message>
```

There are three causes of these messages:

- MCR/Dispatcher errors
- XML parser errors
- `depfun`-produced errors

MCR/Dispatcher Errors

These errors originate directly from the MCR/Dispatcher. If one of these error occurs, report it to Technical Support at MathWorks at http://www.mathworks.com/contact_TS.html.

XML Parser Errors

These errors appear as

```
depfun Error: XML error: <message>
```

Where <message> is a message returned by the XML parser. If this error occurs, report it to Technical Support at MathWorks at http://www.mathworks.com/contact_TS.html.

depfun-Produced Errors

These errors originate directly from depfun.

depfun Error: Internal error. This error occurs if an internal error is encountered that is unexpected, for example, a memory allocation error or a system error of some kind. This error is never user generated. If this error occurs, report it to Technical Support at MathWorks at http://www.mathworks.com/contact_TS.html.

depfun Error: Unexpected error thrown. This error is similar to the previous one. If this error occurs, report it to Technical Support at MathWorks at http://www.mathworks.com/contact_TS.html.

depfun Error: Invalid file name: <filename>. An invalid file name was passed to depfun.

depfun Error: Invalid directory: <dirname>. An invalid folder was passed to depfun.

C++ Utility Library Reference

Data Conversion Restrictions for the C++ MWArray API

Currently, returning a Java object to your application, from a compiled MATLAB function, is unsupported.

Primitive Types

The `mwArray` API supports all primitive types that can be stored in a MATLAB array. This table lists all the types.

Type	Description	mxClassID
mxChar	Character type	mxCHAR_CLASS
mxLogical	Logical or Boolean type	mxLOGICAL_CLASS
mxDouble	Double-precision floating-point type	mxDOUBLE_CLASS
mxSingle	Single-precision floating-point type	mxSINGLE_CLASS
mxInt8	1-byte signed integer	mxINT8_CLASS
mxUint8	1-byte unsigned integer	mxUINT8_CLASS
mxInt16	2-byte singed integer	mxINT16_CLASS
mxUint16	2-byte unsigned integer	mxUINT16_CLASS
mxInt32	4-byte signed integer	mxINT32_CLASS
mxUint32	4-byte unsigned integer	mxUINT32_CLASS
mxInt64	8-byte signed integer	mxINT64_CLASS
mxUint64	8-byte unsigned integer	mxUINT64_CLASS

C++ Utility Classes

The following are C++ utility classes:

- "mwString Class" on page D-5
- "mwException Class" on page D-21
- "mwArray Class" on page D-30

mwString Class

About mwString

The mwString class is a simple string class used by the mwArray API to pass string data as output from certain methods.

Constructors

- mwString()

- mwString(const char* str)

- mwString(const mwString& str)

Methods

- int Length() const

Operators

- operator const char* () const

- mwString& operator=(const mwString& str)

- mwString& operator=(const char* str)

- bool operator== [equal] (const mwString& str) const

- bool operator!= [not equal] (const mwString& str) const

- bool operator< [less than] (const mwString& str) const

- `bool operator<= [less than or equal] (const mwString& str) const`

- `bool operator> [greater than] (const mwString& str) const`

- `bool operator>= [greater than or equal] (const mwString& str) const`

- `friend std::ostream& operator<<(std::ostream& os, const mwString& str)`

Purpose	Construct empty string
C++ Syntax	`#include "mclcppclass.h"` `mwString str;`
Arguments	None
Return Value	None
Description	Use this constructor to create an empty string.

mwString(const char* str)

Purpose	Construct new string and initialize strings data with supplied char buffer
C++ Syntax	`#include "mclcppclass.h"` `mwString str("This is a string");`
Arguments	str NULL-terminated char buffer to initialize the string.
Return Value	None
Description	Use this constructor to create a string from a NULL-terminated char buffer.

mwString(const mwString& str)

Purpose	Copy constructor for mwString

C++ Syntax	```#include "mclcppclass.h"```
	```mwString str("This is a string");```
	```mwString new_str(str); // new_str contains a copy of the```
	``` // characters in str.```

Arguments	str mwString to be copied.
Return Value	None
Description	Use this constructor to create an mwString that is a copy of an existing one. Constructs a new string and initializes its data with the supplied mwString.

int Length() const

Purpose	Return number of characters in string

C++ Syntax	`#include "mclcppclass.h"`
	`mwString str("This is a string");`
	`int len = str.Length();          // len should be 16.`

Arguments	None
Return Value	The number of characters in the string.
Description	Use this method to get the length of an mwString. The value returned does not include the terminating NULL character.

Purpose	Return pointer to internal buffer of string

C++ Syntax	```#include "mclcppclass.h"```
	```mwString str("This is a string");```
	```const char* pstr = (const char*)str;```

Arguments	None
Return Value	A pointer to the internal buffer of the string.
Description	Use this operator to get direct read-only access to the string's data buffer.

mwString& operator=(const mwString& str)

Purpose	mwString assignment operator

C++ Syntax	`#include "mclcppclass.h"`
	`mwString str("This is a string");`
	`mwString new_str = str;   // new_str contains a copy of`
	`                          // the data in str.`

Arguments	str String to make a copy of.
Return Value	A reference to the invoking mwString object.
Description	Use this operator to copy the contents of one string into another.

mwString& operator=(const char* str)

Purpose	mwString assignment operator

C++ Syntax	`#include "mclcppclass.h"`
	`const char* pstr = "This is a string";`
	`mwString str = pstr; // str contains copy of data in pstr.`

Arguments	str char buffer to make copy of.
Return Value	A reference to the invoking mwString object.
Description	Use this operator to copy the contents of a NULL-terminated buffer into an mwString.

bool operator== [equal] (const mwString& str) const

Purpose	Test two `mwString`s for equality

C++ Syntax	`#include "mclcppclass.h"`
	`mwString str("This is a string");`
	`mwString str2("This is another string");`
	`bool ret = (str == str2);// ret should have value of false.`

Arguments	`str` String to compare.
Return Value	The result of the comparison.
Description	Use this operator to test two strings for equality.

bool operator!= [not equal] (const mwString& str) const

Purpose	Test two mwStrings for inequality

C++
Syntax

```
#include "mclcppclass.h"
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str != str2);  // ret should have value of
                           // true.
```

Arguments   str

   String to compare.

Return   The result of the comparison.
Value

Description Use this operator to test two strings for inequality.

bool operator< [less than] (const mwString& str) const

Purpose   Compare input string with this string and return `true` if this string is lexicographically less than input string

C++ Syntax
```
#include "mclcppclass.h"
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str < str2);        // ret should have a value
                                //  of true.
```

Arguments   str
                     String to compare.

Return Value   The result of the comparison.

Description   Use this operator to test two strings for order.

bool operator<= [less than or equal] (const mwString& str) const

Purpose   Compare input string with this string and return `true` if this string is lexicographically less than or equal to input string

C++
Syntax
```
#include "mclcppclass.h"
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str <= str2);        // ret should have value
                                 //  of true.
```

Arguments   str
        String to compare.

Return   The result of the comparison.
Value

Description   Use this operator to test two strings for order.

bool operator> [greater than] (const mwString& str) const

Purpose	Compare input string with this string and return `true` if this string is lexicographically greater than input string

C++ Syntax	`#include "mclcppclass.h"`
	`mwString str("This is a string");`
	`mwString str2("This is another string");`
	`bool ret = (str > str2);        // ret should have value`
	`                                // of false.`

Arguments	str String to compare.
Return Value	The result of the comparison.
Description	Use this operator to test two strings for order.

bool operator>= [greater than or equal] (const mwString& str) const

Purpose   Compare input string with this string and return `true` if this string is lexicographically greater than or equal to input string

C++
Syntax
```
#include "mclcppclass.h"
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str >= str2);//ret should have value of false.
```

Arguments   str
                    String to compare.

Return   The result of the comparison.
Value

Description Use this operator to test two strings for order.

friend std::ostream& operator<<(std::ostream& os, const mwString& str)

Purpose	Copy contents of input string to specified ostream

C++ Syntax	`#include "mclcppclass.h"`
	`#include <ostream>`
	`mwString str("This is a string");`
	`std::cout << str << std::endl; //should print "This is a`
	`                                    //string" to standard out.`

Arguments	os
	    ostream to copy string to.
	
	str
	    String to copy.

Return Value	The input ostream.
Description	Use this operator to print the contents of an mwString to an ostream.

mwException Class

About mwException

The `mwException` class is the basic exception type used by the `mwArray` API and the C++ interface functions. All errors created during calls to the `mwArray` API and to MATLAB Compiler generated C++ interface functions are thrown as `mwExceptions`.

Constructors

- `mwException()`
- `mwException(const char* msg)`
- `mwException(const mwException& e)`
- `mwException(const std::exception& e)`

Methods

- `const char *what() const throw()`
- `void print_stack_trace()`

Operators

- `mwException& operator=(const mwException& e)`
- `mwException& operator=(const std::exception& e)`

mwException()

Purpose	Construct new `mwException` with default error message
C++ Syntax	`#include "mclcppclass.h"` `throw mwException();`
Arguments	None
Return Value	None
Description	Use this constructor to create an `mwException` without specifying an error message.

Purpose	Construct new `mwException` with specified error message
| **C++ Syntax** | ```
#include "mclcppclass.h"
try
{
    throw mwException("This is an error");
}
catch (const mwException& e)
{
     std::cout << e.what() << std::endl // Displays "This
                                        // is an error" to
                                        // standard out.
}
``` |
Arguments	`msg` Error message.
Return Value	None
Description	Use this constructor to create an `mwException` with a specified error message.

mwException(const mwException& e)

Purpose	Copy constructor for mwException class

**C++
Syntax**

```
#include "mclcppclass.h"
try
{
        throw mwException("This is an error");
}
catch (const mwException& e)
{
        throw mwException(e);     // Rethrows same error.
}
```

Arguments

e
    mwException to create copy of.

**Return
Value**

None

Description   Use this constructor to create a copy of an mwException. The copy will
have the same error message as the original.

mwException(const std::exception& e)

Purpose   Create new `mwException` from existing `std::exception`

C++
Syntax
```
#include "mclcppclass.h"
try
{
    ...
}
catch (const std::exception& e)
{
    throw mwException(e);        // Rethrows same error.
}
```

Arguments   e
                     `std::exception` to create copy of.

Return   None
Value

Description   Use this constructor to create a new `mwException` and initialize the
                  error message with the error message from the given `std::exception`.

const char *what() const throw()

Purpose   Return error message contained in this exception

C++
Syntax

```
#include "mclcppclass.h"
try
{
    ...
}
catch (const std::exception& e)
{
    std::cout << e.what() << std::endl; // Displays error
                                        // message to
                                        // standard out.
}
```

Arguments   None

Return
Value   A pointer to a NULL-terminated character buffer containing the error message.

Description   Use this method to retrieve the error message from an mwException.

void print_stack_trace()

Purpose	Prints stack trace to std::cerr
C++ Syntax	void print_stack_trace()
Arguments	None
Return Value	None
Description	Use this method to print a stack trace, providing more debugging information about a C++ exception.

mwException& operator=(const mwException& e)

Purpose	Assignment operator for `mwException` class

C++ Syntax

```
#include "mclcppclass.h"
try
{
    ...
}
catch (const mwException& e)
{
    mwException e2 = e;
    throw e2;
}
```

Arguments

e

    `mwException` to create copy of.

Return Value

A reference to the invoking `mwException`.

Description

Use this operator to create a copy of an `mwException`. The copy will have the same error message as the original.

mwException& operator=(const std::exception& e)

Purpose   Assignment operator for `mwException` class

C++
Syntax
```
#include "mclcppclass.h"
try
{
    ...
}
catch (const std::exception& e)
{
    mwException e2 = e;
    throw e2;
}
```

Arguments   e
                    `std::exception` to initialize copy with.

Return   A reference to the invoking `mwException`.
Value

Description Use this operator to create a copy of an `std::exception`. The copy will
                have the same error message as the original.

mwArray Class

In this section...
"About mwArray" on page D-30
"Constructors" on page D-30
"Methods" on page D-31
"Operators" on page D-32
"Static Methods" on page D-33

About mwArray

Use the `mwArray` class to pass input/output arguments to MATLAB Compiler generated C++ interface functions. This class consists of a thin wrapper around a MATLAB array. As explained in further detail in the MATLAB documentation, all data in MATLAB is represented by matrices (in other words, even a simple data structure should be declared as a 1-by-1 matrix). The `mwArray` class provides the necessary constructors, methods, and operators for array creation and initialization, as well as simple indexing.

Note Arithmetic operators, such as addition and subtraction, are no longer supported as of Release 14.

Constructors

- `mwArray()`

- `mwArray(mxClassID mxID)`

- `mwArray(mwSize num_rows, mwSize num_cols, mxClassID mxID, mxComplexity cmplx = mxREAL)`

- `mwArray(mwSize num_dims, const mwSize* dims, mxClassID mxID, mxComplexity cmplx = mxREAL)`

- `mwArray(const char* str)`

- `mwArray(mwSize num_strings, const char** str)`

- mwArray(mwSize num_rows, mwSize num_cols, int num_fields,
  const char** fieldnames)

- mwArray(mwSize num_dims, const mwSize* dims, int num_fields,
  const char** fieldnames)

- mwArray(const mwArray& arr)

- mwArray(<type> re)

- mwArray(<type> re, <type> im)

Methods

- mwArray Clone() const

- mwArray SharedCopy() const

- mwArray Serialize() const

- mxClassID ClassID() const

- int ElementSize() const

- size_t ElementSize() const

- mwSize NumberOfElements() const

- mwSize NumberOfNonZeros() const

- mwSize MaximumNonZeros() const

- mwSize NumberOfDimensions() const

- int NumberOfFields() const

- mwString GetFieldName(int index)

- mwArray GetDimensions() const

- bool IsEmpty() const

- bool IsSparse() const

- bool IsNumeric() const

- bool IsComplex() const

- bool Equals(const mwArray& arr) const

- int CompareTo(const mwArray& arr) const

D-31

- int HashCode() const
- mwString ToString() const
- mwArray RowIndex() const
- mwArray ColumnIndex() const
- void MakeComplex()
- mwArray Get(mwSize num_indices, ...)
- mwArray Get(const char* name, mwSize num_indices, ...)
- mwArray Get(mwSize num_indices, const mwIndex* index)
- mwArray Get(const char* name, mwSize num_indices, const mwIndex* index)
- mwArray Real()
- mwArray Imag()
- void Set(const mwArray& arr)
- void GetData(<numeric-type>* buffer, mwSize len) const
- void GetLogicalData(mxLogical* buffer, mwSize len) const
- void GetCharData(mxChar* buffer, mwSize len) const
- void SetData(<numeric-type>* buffer, mwSize len)
- void SetLogicalData(mxLogical* buffer, mwSize len)
- void SetCharData(mxChar* buffer, mwSize len)

Operators

- mwArray operator()(mwIndex i1, mwIndex i2, mwIndex i3, ...,)
- mwArray operator()(const char* name, mwIndex i1, mwIndex i2, mwIndex i3, ...,)
- mwArray& operator=(const <type>& x)
- operator <type>() const

Static Methods

- `static mwArray Deserialize(const mwArray& arr)`

- `static double GetNaN()`

- `static double GetEps()`

- `static double GetInf()`

- `static bool IsFinite(double x)`

- `static bool IsInf(double x)`

- `static bool IsNaN(double x)`

Static Factory Methods for Sparse Arrays

- `static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxDouble* rData, num_rows, num_cols, nzmax)`

- `static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxDouble* rdata, nzmax)`

- `static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxDouble* rdata, mxDouble* idata, num_rows, num_cols, nzmax)`

- `static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxDouble* rdata, mxDouble* idata, mwsize, nzmax)`

- `static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxLogical* rdata, num_rows, num_cols, nzmax)`

- `static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxLogical* rData, nzmax)`

- `static mwArray NewSparse (num_rows, num_cols, nzmax, mxClassID mxID, mxComplexity cmplx = mxREAL)`

D-33

mwArray()

Purpose	Construct empty array of type mxDOUBLE_CLASS
C++ Syntax	```#include "mclcppclass.h"``` ```mwArray a;```
Return Value	None
Description	Use this constructor to create an empty array of type mxDOUBLE_CLASS.

mwArray(mxClassID mxID)

Purpose	Construct empty array of specified type
C++ Syntax	`#include "mclcppclass.h"` `mwArray a(mxDOUBLE_CLASS);`
Return Value	None
Description	Use this constructor to create an empty array of the specified type. You can use any valid mxClassID. See the External Interfaces documentation for more information on mxClassID.

mwArray(mwSize num_rows, mwSize num_cols, mxClassID mxID, mxComplexity cmplx = mxREAL)

Purpose	Construct 2-D matrix of specified type and dimensions

C++ Syntax	`#include "mclcppclass.h"`
	`mwArray a(2, 2, mxDOUBLE_CLASS);`
	`mwArray b(3, 3, mxSINGLE_CLASS, mxCOMPLEX);`
	`mwArray c(2, 3, mxCELL_CLASS);`

Arguments

num_rows
>   The number of rows.

num_cols
>   The number of columns.

mxID
>   The data type type of the matrix.

cmplx
>   The complexity of the matrix (numeric types only).

Return Value

None

Description   Use this constructor to create a matrix of the specified type and complexity. For numeric types, the matrix can be either real or complex. You can use any valid mxClassID. Consult the External Interfaces documentation for more information on mxClassID. For numeric types, pass mxCOMPLEX for the last argument to create a complex matrix. All elements are initialized to zero. For cell matrices, all elements are initialized to empty cells.

mwArray(mwSize num_dims, const mwSize* dims, mxClassID mxID, mxComplexity cmplx = mxREAL)

Purpose	Construct n-dimensional array of specified type and dimensions

C++
Syntax

```
#include "mclcppclass.h"
mwSize dims[3] = {2,3,4};
mwArray a(3, dims, mxDOUBLE_CLASS);
mwArray b(3, dims, mxSINGLE_CLASS, mxCOMPLEX);
mwArray c(3, dims, mxCELL_CLASS);
```

Arguments

num_dims
>   Size of the dims array.

dims
>   Dimensions of the array.

mxID
>   The data type type of the matrix.

cmplx
>   The complexity of the matrix (numeric types only).

Return Value

None

Description   Use this constructor to create an n-dimensional array of the specified type and complexity. For numeric types, the array can be either real or complex. You can use any valid mxClassID. Consult the External Interfaces documentation for more information on mxClassID. For numeric types, pass mxCOMPLEX for the last argument to create a complex matrix. All elements are initialized to zero. For cell arrays, all elements are initialized to empty cells.

mwArray(const char* str)

Purpose	Construct character array from supplied string
C++ Syntax	`#include "mclcppclass.h"` `mwArray a("This is a string");`
Arguments	`str` NULL-terminated string
Return Value	None
Description	Use this constructor to create a 1-by-*n* array of type `mxCHAR_CLASS`, with `n = strlen(str)`, and initialize the array's data with the characters in the supplied string.

mwArray(mwSize num_strings, const char** str)

Purpose	Construct character matrix from list of strings

C++ Syntax	`#include "mclcppclass.h"`
	`const char* str[] = {"String1", "String2", "String3"};`
	`mwArray a(3, str);`

Arguments	num_strings
	    Number of strings in the input array
	
	str
	    Array of NULL-terminated strings

Return Value	None
Description	Use this constructor to create a matrix of type mxCHAR_CLASS, and initialize the array's data with the characters in the supplied strings. The created array has dimensions m-by-max, where max is the length of the longest string in str.

mwArray(mwSize num_rows, mwSize num_cols, int num_fields, const char** fieldnames)

Purpose	Construct 2-D MATLAB structure matrix of specified dimensions and field names

C++ Syntax	```#include "mclcppclass.h"```
	```const char* fields[] = {"a", "b", "c"};```
	```mwArray a(2, 2, 3, fields);```

Arguments	num_rows
	    Number of rows in the struct matrix.
	
	num_cols
	    Number of columns in the struct matrix.
	
	num_fields
	    Number of fields in the struct matrix.
	
	fieldnames
	    Array of NULL-terminated strings representing the field names.

Return Value	None
Description	Use this constructor to create a matrix of type mxSTRUCT_CLASS, with the specified field names. All elements are initialized with empty cells.

mwArray(mwSize num_dims, const mwSize* dims, int num_fields, const char** fieldnames)

Purpose	Construct n-dimensional MATLAB structure array of specified dimensions and field names

C++ Syntax	`#include "mclcppclass.h"`
	`const char* fields[] = {"a", "b", "c"};`
	`mwSize dims[3] = {2, 3, 4}`
	`mwArray a(3, dims, 3, fields);`

Arguments

num_dims
    Size of the dims array.

dims
    Dimensions of the struct array.

num_fields
    Number of fields in the struct array.

fieldnames
    Array of NULL-terminated strings representing the field names.

Return Value

None

Description

Use this constructor to create an n-dimensional array of type mxSTRUCT_CLASS, with the specified field names. All elements are initialized with empty cells.

mwArray(const mwArray& arr)

Purpose	Constructs new `mwArray` from existing array

C++ Syntax	`#include "mclcppclass.h"`
	`mwArray a(2, 2, mxDOUBLE_CLASS);`
	`mwArray b(a);`

Arguments	`arr` `mwArray` to copy.
Return Value	None
Description	Use this constructor to create a copy of an existing array. The new array contains a deep copy of the input array.

Purpose   Construct real scalar array of type of the input argument and initialize
                     data with input argument's value

C++
Syntax
```
#include "mclcppclass.h"
double x = 5.0;
mwArray a(x);   // Creates 1X1 double array with value 5.0
```

Arguments   re
                         Scalar value to initialize array with.

Return   None
Value

Description   Use this constructor to create a real scalar array. \<type\> can be any
                    of the following:

- mxDouble

- mxSingle

- mxInt8

- mxUint8

- mxInt16

- mxUint16

- mxInt32

- mxUint32

- mxInt64

- mxUint64

- mxLogical

The scalar array is created with the type of the input argument.

mwArray(<type> re, <type> im)

Purpose	Construct complex scalar array of type of input arguments and initialize real and imaginary parts of data with input argument's values

C++ Syntax	```#include "mclcppclass.h"
	double re = 5.0;
	double im = 10.0;
	mwArray a(re, im); // Creates 1X1 complex array with
	                   // value 5+10i```

Arguments

re

    Scalar value to initialize real part with.

im

    Scalar value to initialize imaginary part with.

Return Value

None

Description

Use this constructor to create a complex scalar array. The first input argument initializes the real part and the second argument initializes the imaginary part. `<type>` can be any of the following: `mxDouble`, `mxSingle`, `mxInt8`, `mxUint8`, `mxInt16`, `mxUint16`, `mxInt32`, `mxUint32`, `mxInt64`, or `mxUint64`.

- `mxDouble`
- `mxSingle`
- `mxInt8`
- `mxUint8`
- `mxInt16`
- `mxUint16`
- `mxInt32`
- `mxUint32`
- `mxInt64`

- `mxUint64`

- `mxLogical`

The scalar array is created with the type of the input arguments.

mwArray Clone() const

Purpose	Return new array representing deep copy of array

C++ Syntax	`#include "mclcppclass.h"`
	`mwArray a(2, 2, mxDOUBLE_CLASS);`
	`mwArray b = a.Clone();`

Arguments	None
Return Value	New `mwArray` representing a deep copy of the original.
Description	Use this method to create a copy of an existing array. The new array contains a deep copy of the input array.

mwArray SharedCopy() const

Purpose	Return new array representing shared copy of array

**C++
Syntax**

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray b = a.SharedCopy();
```

Arguments   None

**Return
Value**

New mwArray representing a reference counted version of the original.

Description   Use this method to create a shared copy of an existing array. The new array and the original array both point to the same data.

mwArray Serialize() const

Purpose
Serialize underlying array into byte array, and return data in new array of type mxUINT8_CLASS

C++ Syntax
```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray s = a.Serialize();
```

Arguments
None

Return Value
New mwArray of type mxUINT8_CLASS containing the serialized data.

Description
Use this method to serialize an array into bytes. A 1-by-n numeric matrix of type mxUINT8_CLASS is returned containing the serialized data. The data can be deserialized back into the original representation by calling mwArray::Deserialize().

Purpose	Return type of array
C++ Syntax	```#include "mclcppclass.h" mwArray a(2, 2, mxDOUBLE_CLASS); mxClassID id = a.ClassID();// Should return mxDOUBLE_CLASS```
Arguments	None
Return Value	The mxClassID of the array.
Description	Use this method to determine the type of the array. Consult the External Interfaces documentation for more information on mxClassID.

int ElementSize() const

Purpose	Return size in bytes of element of array

C++
Syntax
```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
int size = a.ElementSize();// Should return sizeof(double)
```

Arguments   None

Return
Value   The size in bytes of an element of this type of array.

Description   Use this method to determine the size in bytes of an element of array
type.

Purpose	Return size in bytes of element in array

C++ Syntax	```#include "mclcppclass.h"```
	```mwArray a(2, 2, mxDOUBLE_CLASS);```
	```int size = a.ElementSize();// Should return sizeof(double)```

Arguments	None
Return Value	The size in bytes of an element of this type of array.
Description	Use this method to determine the size in bytes of an element of array type.

mwSize NumberOfElements() const

Purpose	Return number of elements in array

C++ Syntax	`#include "mclcppclass.h"`
	`mwArray a(2, 2, mxDOUBLE_CLASS);`
	`int n = a.NumberOfElements();// Should return 4`

Arguments	None
Return Value	Number of elements in array.
Description	Use this method to determine the total size of the array.

mwSize NumberOfNonZeros() const

Purpose	Return number of nonzero elements for sparse array

C++ Syntax	`#include "mclcppclass.h"`
	`mwArray a(2, 2, mxDOUBLE_CLASS);`
	`int n = a.NumberOfNonZeros();// Should return 4`

Arguments	None
Return Value	Actual number of nonzero elements in array.
Description	Use this method to determine the size of the of the array's data. If the underlying array is not sparse, this returns the same value as `NumberOfElements()`.

Note This method does not analyze the actual values of the array elements. Instead, it returns the number of elements that can potentially be nonzero. This is exactly the number of elements for which the sparse matrix has allocated storage.

mwSize MaximumNonZeros() const

Purpose	Return maximum number of nonzero elements for sparse array

C++ Syntax	`#include "mclcppclass.h"`
	`mwArray a(2, 2, mxDOUBLE_CLASS);`
	`int n = a.MaximumNonZeros();// Should return 4`

Arguments	None
Return Value	Number of allocated nonzero elements in array.
Description	Use this method to determine the allocated size of the of the array's data. If the underlying array is not sparse, this returns the same value as `NumberOfElements()`.

Note This method does not analyze the actual values of the array elements.

mwSize NumberOfDimensions() const

Purpose	Return number of dimensions in array

C++ Syntax	`#include "mclcppclass.h"`
	`mwArray a(2, 2, mxDOUBLE_CLASS);`
	`int n = a.NumberOfDimensions();// Should return 2`

Arguments	None
Return Value	Number of dimensions in array.
Description	Use this method to determine the dimensionality of the array.

int NumberOfFields() const

Purpose	Return number of fields in `struct` array

C++ Syntax	`#include "mclcppclass.h"`
	`const char* fields[] = {"a", "b", "c"};`
	`mwArray a(2, 2, 3, fields);`
	`int n = a.NumberOfFields(); // Should return 3`

Arguments	None
Return Value	Number of fields in the array.
Description	Use this method to determine the number of fields in a `struct` array. If the underlying array is not of type `struct`, zero is returned.

mwString GetFieldName(int index)

Purpose   Return string representing name of (zero-based) field in `struct` array

C++   ```
Syntax   #include "mclcppclass.h"
                   const char* fields[] = {"a", "b", "c"};
                   mwArray a(2, 2, 3, fields);
                   mwString tempname = a.GetFieldName(1);
                   const char* name = (const char*)tempname; // Should
                                                             // return "b"
                   ```

Arguments   Index
                       Zero-based field number.

Return   `mwString` containing the field name.
Value

Description   Use this method to determine the name of a given field in a `struct` array. If the underlying array is not of type `struct`, an exception is thrown.

mwArray GetDimensions() const

Purpose	Return array of type `mxINT32_CLASS` representing dimensions of array

C++ Syntax	`#include "mclcppclass.h"`
	`mwArray a(2, 2, mxDOUBLE_CLASS);`
	`mwArray dims = a.GetDimensions();`

Arguments	None
Return Value	`mwArray` type `mxINT32_CLASS` containing the dimensions of the array.
Description	Use this method to determine the size of each dimension in the array. The size of the returned array is 1-by-`NumberOfDimensions()`.

Purpose	Return `true` if underlying array is empty

C++ Syntax	`#include "mclcppclass.h"`
	`mwArray a;`
	`bool b = a.IsEmpty(); // Should return true`

Arguments	None
Return Value	Boolean indicating if the array is empty.
Description	Use this method to determine if an array is empty.

bool IsSparse() const

Purpose	Return `true` if underlying array is sparse

C++ Syntax	```#include "mclcppclass.h"```
	```mwArray a(2, 2, mxDOUBLE_CLASS);```
	```bool b = a.IsSparse(); // Should return false```

Arguments	None
Return Value	Boolean indicating if the array is sparse.
Description	Use this method to determine if an array is sparse.

Purpose	Return `true` if underlying array is numeric

C++ Syntax	`#include "mclcppclass.h"`
	`mwArray a(2, 2, mxDOUBLE_CLASS);`
	`bool b = a.IsNumeric(); // Should return true.`

Arguments	None
Return Value	Boolean indicating if the array is numeric.
Description	Use this method to determine if an array is numeric.

bool IsComplex() const

Purpose	Return `true` if underlying array is complex

C++ Syntax	`#include "mclcppclass.h"`
	`mwArray a(2, 2, mxDOUBLE_CLASS, mxCOMPLEX);`
	`bool b = a.IsComplex(); // Should return true.`

Arguments	None
Return Value	Boolean indicating if the array is complex.
Description	Use this method to determine if an array is complex.

bool Equals(const mwArray& arr) const

Purpose   Test two arrays for equality

C++
Syntax
```
#include "mclcppclass.h"
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray b(1, 1, mxDOUBLE_CLASS);
a = 1.0;
b = 1.0;
bool c = a.Equals(b); // Should return true.
```

Arguments   arr
   Array to compare to array.

Return   Boolean value indicating the equality of the two arrays.
Value

Description   Returns true if the input array is byte-wise equal to this array. This
method makes a byte-wise comparison of the underlying arrays.
Therefore, arrays of the same type should be compared. Arrays of
different types will not in general be equal, even if they are initialized
with the same data.

int CompareTo(const mwArray& arr) const

Purpose	Compare two arrays for order

C++ Syntax	```#include "mclcppclass.h"```
	```mwArray a(1, 1, mxDOUBLE_CLASS);```
	```mwArray b(1, 1, mxDOUBLE_CLASS);```
	```a = 1.0;```
	```b = 1.0;```
	```int n = a.CompareTo(b); // Should return 0```

Arguments	arr Array to compare to this array.
Return Value	Returns a negative integer, zero, or a positive integer if this array is less than, equal to, or greater than the specified array.
Description	Compares this array with the specified array for order. This method makes a byte-wise comparison of the underlying arrays. Therefore, arrays of the same type should be compared. Arrays of different types will, in general, not be ordered equivalently, even if they are initialized with the same data.

Purpose   Return hash code for array

C++   `#include "mclcppclass.h"`
Syntax   `mwArray a(1, 1, mxDOUBLE_CLASS);`
                   `int n = a.HashCode();`

Arguments   None

Return   An integer value representing a unique hash code for the array.
Value

Description   This method constructs a unique hash value form the underlying bytes
                   in the array. Therefore, arrays of different types will have different
                   hash codes, even if they are initialized with the same data.

mwString ToString() const

Purpose   Return string representation of underlying array

C++
Syntax
```
#include <stdio.h>
#include "mclcppclass.h"
mwArray a(1, 1, mxDOUBLE_CLASS, mxCOMPLEX);
a.Real() = 1.0;
a.Imag() = 2.0;
printf("%s\n", (const char*)(a.ToString()));// Should print
                                            // "1 + 2i" on
                                            // screen.
```

Arguments   None

Return   An mwString containing the string representation of the array.
Value

Description This method returns a string representation of the underlying array.
The string returned is the same string that is returned by typing a
variable's name at the MATLAB command prompt.

Purpose   Return array containing row indices of each element in array

C++
Syntax
```
#include <stdio.h>
#include "mclcppclass.h"
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray rows = a.RowIndex();
```

Arguments   None

Return   An mwArray containing the row indices.
Value

Description Returns an array of type mxINT32_CLASS representing the row indices
                 (first dimension) of this array. For sparse arrays, the indices are
                 returned for just the non-zero elements and the size of the array
                 returned is 1-by-NumberOfNonZeros(). For nonsparse arrays, the size
                 of the array returned is 1-by-NumberOfElements(), and the row indices
                 of all of the elements are returned.

mwArray ColumnIndex() const

Purpose	Return array containing column indices of each element in array

C++ Syntax	```#include "mclcppclass.h"```
	```mwArray a(1, 1, mxDOUBLE_CLASS);```
	```mwArray rows = a.ColumnIndex();```

Arguments	None
Return Value	An mwArray containing the column indices.
Description	Returns an array of type mxINT32_CLASS representing the column indices (second dimension) of this array. For sparse arrays, the indices are returned for just the non-zero elements and the size of the array returned is 1-by-NumberOfNonZeros(). For nonsparse arrays, the size of the array returned is 1-by-NumberOfElements(), and the column indices of all of the elements are returned.

void MakeComplex()

Purpose	Convert real numeric array to complex

C++ Syntax	```#include "mclcppclass.h"```
	 ```double rdata[4] = {1.0, 2.0, 3.0, 4.0};```
	 ```double idata[4] = {10.0, 20.0, 30.0, 40.0};```
	 ```mwArray a(2, 2, mxDOUBLE_CLASS);```
	 ```a.SetData(rdata, 4);```
	 ```a.MakeComplex();```
	 ```a.Imag().SetData(idata, 4);```

Arguments	None
Return Value	None
Description	Use this method to convert a numeric array that has been previously allocated as real to complex. If the underlying array is of a nonnumeric type, an mwException is thrown.

mwArray Get(mwSize num_indices, ...)

Purpose	Return single element at specified 1-based index
| **C++ Syntax** | ```cpp
#include "mclcppclass.h"
double data[4] = {1.0, 2.0, 3.0, 4.0};
double x;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(data, 4);
x = a.Get(1,1);         // x = 1.0
x = a.Get(2, 1, 2);     // x = 3.0
x = a.Get(2, 2, 2);     // x = 4.0
``` |

Arguments　num_indices
　　　　Number of indices passed in.

　　　　...
　　　　Comma-separated list of input indices. Number of items must equal num_indices but should not exceed 32.

Return Value　An mwArray containing the value at the specified index.

Description　Use this method to fetch a single element at a specified index. The index is passed by first passing the number of indices followed by a comma-separated list of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing), in which case the element at the specified 1-based offset is returned, accessing data in column-wise order, or NumberOfDimensions() (multiple subscript indexing), in which case, the index list is used to access the specified element. The valid range for indices is 1 <= index <= NumberOfElements(), for single subscript indexing. For multiple subscript indexing, the ith index has the valid range: 1 <= index[i] <= GetDimensions().Get(1, i). An mwException is thrown if an invalid number of indices is passed in or if any index is out of bounds.

mwArray Get(const char* name, mwSize num_indices, ...)

Purpose	Return single element at specified field name and 1-based index in struct array

C++ Syntax

```
#include "mclcppclass.h"
const char* fields[] = {"a", "b", "c"};

mwArray a(1, 1, 3, fields);
mwArray b = a.Get("a", 1, 1);            // b=a(1).a;
mwArray b = a.Get("b", 2, 1, 1);         // b=a(1,1).b;
```

Arguments

name
> NULL-terminated string containing the field name to get.

num_indices
> Number of indices passed in.

...
> Comma-separated list of input indices. Number of items must equal num_indices.

Return Value

An mwArray containing the value at the specified field name and index.

Description

Use this method to fetch a single element at a specified field name and index. This method may only be called on an array that is of type mxSTRUCT_CLASS. An mwException is thrown if the underlying array is not a struct array. The field name passed must be a valid field name in the struct array. The index is passed by first passing the number of indices followed by a comma-separated list of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing), in which case the element at the specified 1-based offset is returned, accessing data in column-wise order, or NumberOfDimensions() (multiple subscript indexing), in which case, the index list is used to access the specified element. The valid range for indices is 1 <= index <= NumberOfElements(), for single subscript indexing. For multiple subscript indexing, the ith index has the valid range: 1 <= index[i] <= GetDimensions().Get(1, i). An

mwArray Get(const char* name, mwSize num_indices, ...)

mwException is thrown if an invalid number of indices is passed in or if any index is out of bounds.

mwArray Get(mwSize num_indices, const mwIndex* index)

Purpose	Return single element at specified 1-based index

C++
Syntax
```
#include "mclcppclass.h"
double data[4] = {1.0, 2.0, 3.0, 4.0};
int index[2] = {1, 1};
double x;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(data, 4);
x = a.Get(1, index);        // x = 1.0
x = a.Get(2, index);        // x = 1.0
index[0] = 2;
index[1] = 2;
x = a.Get(2, index);         // x = 4.0
```

Arguments   num_indices
        Size of index array.

    index
        Array of at least size num_indices containing the indices.

Return
Value
An mwArray containing the value at the specified index.

Description   Use this method to fetch a single element at a specified index. The index is passed by first passing the number of indices, followed by an array of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing), in which case the element at the specified 1-based offset is returned, accessing data in column-wise order, or NumberOfDimensions() (multiple subscript indexing), in which case, the index list is used to access the specified element. The valid range for indices is 1 <= index <= NumberOfElements(), for single subscript indexing. For multiple subscript indexing, the ith index has the valid range: 1 <= index[i] <= GetDimensions().Get(1, i). An mwException is thrown if an invalid number of indices is passed in or if any index is out of bounds.

mwArray Get(const char* name, mwSize num_indices, const mwIndex* index)

Purpose	Return single element at specified field name and 1-based index in `struct` array

C++
Syntax

```
#include "mclcppclass.h"
const char* fields[] = {"a", "b", "c"};
int index[2] = {1, 1};
mwArray a(1, 1, 3, fields);
mwArray b = a.Get("a", 1, index);          // b=a(1).a;
mwArray b = a.Get("b", 2, index);          // b=a(1,1).b;
```

Arguments

name
> NULL-terminated string containing the field name to get.

num_indices
> Number of indices passed in.

index
> Array of at least size num_indices containing the indices.

Return Value

An `mwArray` containing the value at the specified field name and index.

Description

Use this method to fetch a single element at a specified field name and index. This method may only be called on an array that is of type mxSTRUCT_CLASS. An mwException is thrown if the underlying array is not a `struct` array. The field name passed must be a valid field name in the `struct` array. The index is passed by first passing the number of indices followed by an array of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing), in which case the element at the specified 1-based offset is returned, accessing data in column-wise order, or NumberOfDimensions() (multiple subscript indexing), in which case, the index list is used to access the specified element. The valid range for indices is 1 <= index <= NumberOfElements(), for single subscript indexing. For multiple subscript indexing, the ith index has the valid range: 1 <= index[i] <= GetDimensions().Get(1, i). An mwException is

mwArray Get(const char* name, mwSize num_indices, const mwIndex* index)

thrown if an invalid number of indices is passed in or if any index is out of bounds.

mwArray Real()

Purpose	Return `mwArray` that references real part of complex array

C++ Syntax	```#include "mclcppclass.h"```
	```double rdata[4] = {1.0, 2.0, 3.0, 4.0};```
	```double idata[4] = {10.0, 20.0, 30.0, 40.0};```
	```mwArray a(2, 2, mxDOUBLE_CLASS, mxCOMPLEX);```
	```a.Real().SetData(rdata, 4);```
	```a.Imag().SetData(idata, 4);```

Arguments	None
Return Value	An `mwArray` referencing the real part of the array.
Description	Use this method to access the real part of a complex array. The returned `mwArray` is considered real and has the same dimensionality and type as the original.

Complex arrays consist of Complex numbers, which are 1 X 2 vectors (pairs). For example, if the number is `3+5i`, then the pair is `(3,5i)`. An array of Complex numbers is therefore two dimensional (`N X 2`), where *N* is the number of complex numbers in the array. `2+4i, 7-3i, 8+6i` would be represented as `(2,4i) (7,3i) (8,6i)`. Complex numbers have two components, real and imaginary.

The MATLAB functions `Real` and `Imag` can be applied to an array of Complex numbers. These functions extract the corresponding part of the Complex number. For example, `REAL(3,5i) == 3` and `IMAG(3+5i) == 5`. `Imag` returns `5` in this case and not `5i`. `Imag` returns the magnitude of the imaginary part of the number as a real number.

Purpose   Return `mwArray` that references imaginary part of complex array

C++
Syntax
```
#include "mclcppclass.h"
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double idata[4] = {10.0, 20.0, 30.0, 40.0};
mwArray a(2, 2, mxDOUBLE_CLASS, mxCOMPLEX);
a.Real().SetData(rdata, 4);
a.Imag().SetData(idata, 4);
```

Arguments   None

Return
Value
An `mwArray` referencing the imaginary part of the array.

Description   Use this method to access the imaginary part of a complex array. The returned `mwArray` is considered real and has the same dimensionality and type as the original.

Complex arrays consist of Complex numbers, which are 1 X 2 vectors (pairs). For example, if the number is `3+5i`, then the pair is (`3`,`5i`). An array of Complex numbers is therefore two dimensional (N X 2), where *N* is the number of complex numbers in the array. `2+4i, 7-3i, 8+6i` would be represented as (`2`,`4i`) (`7`,`3i`) (`8`,`6i`). Complex numbers have two components, real and imaginary.

The MATLAB functions `Real` and `Imag` can be applied to an array of Complex numbers. These functions extract the corresponding part of the Complex number. For example, `REAL(3,5i) == 3` and `IMAG(3+5i) == 5`. `Imag` returns `5` in this case and not `5i`. `Imag` returns the magnitude of the imaginary part of the number as a real number.

void Set(const mwArray& arr)

Purpose	Assign shared copy of input array to currently referenced cell for arrays of type mxCELL_CLASS and mxSTRUCT_CLASS

C++
Syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray b(2, 2, mxINT16_CLASS);
mwArray c(1, 2, mxCELL_CLASS);
c.Get(1,1).Set(a);                              // Sets c(1) = a
c.Get(1,2).Set(b);                              // Sets c(2) = b
```

Arguments
    arr
        mwArray to assign to currently referenced cell.

Return
Value
    None

Description   Use this method to construct cell and struct arrays.

void GetData(\<numeric-type>* buffer, mwSize len) const

Purpose	Copy array's data into supplied numeric buffer

C++ Syntax

```
#include "mclcppclass.h"
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double data_copy[4] ;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(rdata, 4);
a.GetData(data_copy, 4);
```

Arguments

buffer
    Buffer to receive copy.

len
    Maximum length of buffer. A maximum of len elements will be copied.

Return Value

None

Description

Valid types for \<numeric-type> are:

- mxDOUBLE_CLASS

- mxSINGLE_CLASS

- mxINT8_CLASS

- mxUINT8_CLASS

- mxINT16_CLASS

- mxUINT16_CLASS

- mxINT32_CLASS

- mxUINT32_CLASS

- mxINT64_CLASS

- mxUINT64_CLASS

void GetData(<numeric-type>* buffer, mwSize len) const

The data is copied in column-major order. If the underlying array is not of the same type as the input buffer, the data is converted to this type as it is copied. If a conversion cannot be made, an mwException is thrown.

void GetLogicalData(mxLogical* buffer, mwSize len) const

Purpose	Copy array's data into supplied mxLogical buffer

C++
Syntax

```
#include "mclcppclass.h"
mxLogical data[4] = {true, false, true, false};
mxLogical data_copy[4] ;
mwArray a(2, 2, mxLOGICAL_CLASS);
a.SetLogicalData(data, 4);
a.GetLogicalData(data_copy, 4);
```

Arguments
buffer
  Buffer to receive copy.

len
  Maximum length of buffer. A maximum of len elements will be
  copied.

Return
Value

None

Description
The data is copied in column-major order. If the underlying array is not
of type mxLOGICAL_CLASS, the data is converted to this type as it is
copied. If a conversion cannot be made, an mwException is thrown.

void GetCharData(mxChar* buffer, mwSize len) const

Purpose	Copy array's data into supplied `mxChar` buffer

C++ Syntax	`#include "mclcppclass.h"`
	 `mxChar data[6] = {'H', 'e' , `l` , 'l' , 'o' , '\0'};`
	 `mxChar data_copy[6] ;`
	 `mwArray a(1, 6, mxCHAR_CLASS);`
	 `a.SetCharData(data, 6);`
	 `a.GetCharData(data_copy, 6);`

Arguments	`buffer`
	      Buffer to receive copy.
	
	 `len`
	      Maximum length of buffer. A maximum of `len` elements will be copied.

Return Value	None
Description	The data is copied in column-major order. If the underlying array is not of type `mxCHAR_CLASS`, the data is converted to this type as it is copied. If a conversion cannot be made, an `mwException` is thrown.

void SetData(<numeric-type>* buffer, mwSize len)

Purpose	Copy data from supplied numeric buffer into array

C++ Syntax	```#include "mclcppclass.h"```
	```double rdata[4] = {1.0, 2.0, 3.0, 4.0};```
	```double data_copy[4] ;```
	```mwArray a(2, 2, mxDOUBLE_CLASS);```
	```a.SetData(rdata, 4);```
	```a.GetData(data_copy, 4);```

Arguments

buffer
> Buffer containing data to copy.

len
> Maximum length of buffer. A maximum of len elements will be copied.

Return Value

None

Description

Valid types for <numeric-type> are mxDOUBLE_CLASS, mxSINGLE_CLASS, mxINT8_CLASS, mxUINT8_CLASS, mxINT16_CLASS, mxUINT16_CLASS, mxINT32_CLASS, mxUINT32_CLASS, mxINT64_CLASS, and mxUINT64_CLASS. The data is copied in column-major order. If the underlying array is not of the same type as the input buffer, the data is converted to this type as it is copied. If a conversion cannot be made, an mwException is thrown.

void SetLogicalData(mxLogical* buffer, mwSize len)

Purpose	Copy data from supplied `mxLogical` buffer into array

C++ Syntax	```#include "mclcppclass.h"```
	```mxLogical data[4] = {true, false, true, false};```
	```mxLogical data_copy[4] ;```
	```mwArray a(2, 2, mxLOGICAL_CLASS);```
	```a.SetLogicalData(data, 4);```
	```a.GetLogicalData(data_copy, 4);```

C++
Syntax

```
#include "mclcppclass.h"
mxLogical data[4] = {true, false, true, false};
mxLogical data_copy[4] ;
mwArray a(2, 2, mxLOGICAL_CLASS);
a.SetLogicalData(data, 4);
a.GetLogicalData(data_copy, 4);
```

Arguments   buffer
           Buffer containing data to copy.

           len
                 Maximum length of buffer. A maximum of len elements will be
                 copied.

Return   None
Value

Description The data is copied in column-major order. If the underlying array is not
                 of type mxLOGICAL_CLASS, the data is converted to this type as it is
                 copied. If a conversion cannot be made, an mwException is thrown.

void SetCharData(mxChar* buffer, mwSize len)

Purpose	Copy data from supplied mxChar buffer into array

C++ Syntax	`#include "mclcppclass.h"`
	`mxChar data[6] = {'H', 'e' , 'l' , 'l' , 'o' , '\0'};`
	`mxChar data_copy[6] ;`
	`mwArray a(1, 6, mxCHAR_CLASS);`
	`a.SetCharData(data, 6);`
	`a.GetCharData(data_copy, 6);`

Arguments	buffer
	    Buffer containing data to copy.
	
	len
	    Maximum length of buffer. A maximum of len elements will be copied.

Return Value	None
Description	The data is copied in column-major order. If the underlying array is not of type mxCHAR_CLASS, the data is converted to this type as it is copied. If a conversion cannot be made, an mwException is thrown.

mwArray operator()(mwIndex i1, mwIndex i2, mwIndex i3, ...,)

Purpose	Return single element at specified 1-based index

C++
Syntax

```
#include "mclcppclass.h"
double data[4] = {1.0, 2.0, 3.0, 4.0};
double x;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(data, 4);
x = a(1,1);                    // x = 1.0
x = a(1,2);                    // x = 3.0
x = a(2,2);                    // x = 4.0
```

Arguments   i1, i2, i3, ...,
                        Comma-separated list of input indices.

**Return
Value**

An mwArray containing the value at the specified index.

Description   Use this operator to fetch a single element at a specified index.
The index is passed as a comma-separated list of 1-based indices.
This operator is overloaded to support 1 through 32 indices. The
valid number of indices that can be passed in is either 1 (single
subscript indexing), in which case the element at the specified
1-based offset is returned, accessing data in column-wise order, or
NumberOfDimensions() (multiple subscript indexing), in which case,
the index list is used to access the specified element. The valid range
for indices is 1 <= index <= NumberOfElements(), for single subscript
indexing. For multiple subscript indexing, the ith index has the
valid range: 1 <= index[i] <= GetDimensions().Get(1, i). An
mwException is thrown if an invalid number of indices is passed in
or if any index is out of bounds.

mwArray operator()(const char* name, mwIndex i1, mwIndex i2, mwIndex i3, ...,)

Purpose	Return single element at specified field name and 1-based index in `struct` array

**C++
Syntax**

```
#include "mclcppclass.h"
const char* fields[] = {"a", "b", "c"};
int index[2] = {1, 1};
mwArray a(1, 1, 3, fields);
mwArray b = a("a", 1, 1);                    // b=a(1).a;
mwArray b = a("b", 1, 1);                    // b=a(1,1).b;
```

Arguments

name
> NULL-terminated string containing the field name to get.

i1, i2, i3, ...,
> Comma-separated list of input indices.

**Return
Value**

An `mwArray` containing the value at the specified field name and index

Description

Use this method to fetch a single element at a specified field name and index. This method may only be called on an array that is of type `mxSTRUCT_CLASS`. An `mwException` is thrown if the underlying array is not a `struct` array. The field name passed must be a valid field name in the `struct` array. The index is passed by first passing the number of indices, followed by an array of 1-based indices. This operator is overloaded to support 1 through 32 indices. The valid number of indices that can be passed in is either 1 (single subscript indexing), in which case the element at the specified 1-based offset is returned, accessing data in column-wise order, or `NumberOfDimensions()` (multiple subscript indexing), in which case, the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the ith index has the valid range: `1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

mwArray& operator=(const <type>& x)

Purpose	Assign single scalar value to array

C++
Syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
a(1,1) = 1.0;                    // assigns 1.0 to element (1,1)
a(1,2) = 2.0;                    // assigns 2.0 to element (1,2)
a(2,1) = 3.0;                    // assigns 3.0 to element (2,1)
a(2,2) = 4.0;                    // assigns 4.0 to element (2,2)
```

Arguments   x
   Value to assign.

Return
Value

A reference to the invoking mwArray.

Description   Use this operator to set a single scalar value. This operator is
overloaded for all numeric and logical types.

Purpose Fetch single scalar value from array

C++ Syntax
```
#include "mclcppclass.h"
double data[4] = {1.0, 2.0, 3.0, 4.0};
double x;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(data, 4);
x = (double)a(1,1);                          // x = 1.0
x = (double)a(1,2);                          // x = 3.0
x = (double)a(2,1);                          // x = 2.0
x = (double)a(2,2);                          // x = 4.0
```

Arguments None

Return Value A single scalar value from the array.

Description Use this operator to fetch a single scalar value. This operator is overloaded for all numeric and logical types.

static mwArray Deserialize(const mwArray& arr)

Purpose	Deserialize array that was serialized with mwArray::Serialize

C++ Syntax	```#include "mclcppclass.h"```
	```double rdata[4] = {1.0, 2.0, 3.0, 4.0};```
	```mwArray a(1,4,mxDOUBLE_CLASS);```
	```a.SetData(rdata, 4);```
	```mwArray b = a.Serialize();```
	```a = mwArray::Deserialize(b);// a should contain same```
	``` // data as original```

Arguments	arr mwArray that has been obtained by calling mwArray::Serialize.
Return Value	A new mwArray containing the deserialized array.
Description	Use this method to deserialize an array that has been serialized with mwArray::Serialize(). The input array must be of type mxUINT8_CLASS and contain the data from a serialized array. If the input data does not represent a serialized mwArray, the behavior of this method is undefined.

static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxDouble* rData, num_rows, num_cols, nzmax)

Purpose	Creates real sparse matrix of type `double` with specified number of rows and columns.

C++ Signature

```
static mwArray NewSparse(mwSize rowindex_size,
                         const mwIndex* rowindex,
                         mwSize colindex_size,
                         const mwIndex* colindex,
                         mwSize data_size,
                         const mxDouble* rData,
                         mwSize num_rows,
                         mwSize num_cols,
                         mwSize nzmax)
```

Arguments   **Inputs**

rowindex_size
    Size of `rowindex` array.

rowindex
    Array of row indices of non-zero elements.

colindex_size
    Size of `colindex` array.

colindex
    Array of column indices of non-zero elements.

data_size
    size of data array(s).

rData
    Data associated with non-zero row and column indices.

num_rows
    Number of rows in matrix.

num_cols
    Number of columns in matrix.

static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxDouble* rData, num_rows, num_cols, nzmax)

nzmax
> Reserved storage for sparse matrix. If `nzmax` is zero, storage will be set to `max{rowindex_size, colindex_size, data_size}`.

Outputs

mwArray
> `mwArray` containing the sparse array.

Description
The lengths of input row, column index, and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated throughout the construction of the matrix.

If any element of the `rowindex` or `colindex` array is greater than the specified values in `num_rows` or `num_cols` respectively, an exception is thrown.

Example
This example constructs a sparse 4 X 4 tridiagonal matrix:

```
 2 -1  0  0
-1  2 -1  0
 0 -1  2 -1
 0  0 -1  2
```

The following code, when run:

```
double rdata[]      =
        {2.0, -1.0, -1.0, 2.0, -1.0,
         -1.0, 2.0, -1.0, -1.0, 2.0};
mwIndex row_tridiag[] =
        {1,    2,    1,    2,    3,
         2,    3,    4,    3,    4  };
mwIndex col_tridiag[] =
        {1,    1,    2,    2,    2,
          3,   3,    3,    4,    4  };

mwArray mysparse =
        mwArray::NewSparse(10, row_tridiag,
```

static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxDouble* rData, num_rows, num_cols, nzmax)

```
                                        10, col_tridiag,
                                        10, rdata, 4, 4, 10);
std::cout << mysparse << std::endl;
```

will display the following output to the screen:

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

static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxDouble* rdata, nzmax)

Purpose   Creates real sparse matrix of type `double` with number of rows and columns inferred (not explicitly specified) from input data.

C++   ```
static mwArray NewSparse(mwSize rowindex_size,
                         const mwIndex* rowindex,
                         mwSize colindex_size,
                         const mwIndex* colindex,
                         mwSize data_size,
                         const mxDouble* rData,
                         mwSize nzmax)
```
**Signature**

**Arguments**       **Inputs**

rowindex_size
    Size of `rowindex` array.

rowindex
    Array of row indices of non-zero elements.

colindex_size
    Size of `colindex` array.

colindex
    Array of column indices of non-zero elements.

data_size
    Size of data array(s).

rData
    Data associated with non-zero row and column indices.

nzmax
    Reserved storage for sparse matrix. If `nzmax` is zero, storage will be set to max{`rowindex_size`, `colindex_size`, `data_size`}.

**Outputs**

mwArray
    `mwArray` containing the sparse array

# static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxDouble* rdata, nzmax)

**Description**    The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated through out the construction of the matrix.

The number of rows and columns in the created matrix are calculated form the input rowindex and colindex arrays as num_rows = max{rowindex}, num_cols = max{colindex}.

**Example 1**    This example uses the data from the example for creating a real sparse matrix of type double with specified number of rows and columns, but allows the number of rows, number of columns , and allocated storage to be calculated from the input data:

```
mwArray mysparse =
  mwArray::NewSparse(10, row_tridiag,
                     10, col_tridiag,
                     10, rdata,
                     0);

std::cout << mysparse << std::endl;

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

Example 2   In this example, we construct a sparse 4 X 4 identity matrix. The value of 1.0 is copied to each non-zero element defined by row and column index arrays:

static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxDouble* rdata, nzmax)

```
double one = 1.0;
mwIndex row_diag[] = {1, 2, 3, 4};
mwIndex col_diag[] = {1, 2, 3, 4};

mwArray mysparse =
  mwArray::NewSparse(4, row_diag,
                     4, col_diag,
                     1, &one,
                     0);
std::cout << mysparse << std::endl;

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

static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxDouble* rdata, mxDouble* idata, num_rows, num_cols, nzmax)

Purpose   Creates complex sparse matrix of type `double` with specified number of rows and columns.

**C++
Signature**

```
static mwArray NewSparse(mwSize rowindex_size,
                         const mwIndex* rowindex,
                         mwSize colindex_size,
                         const mwIndex* colindex,
                         mwSize data_size,
                         const mxDouble* rData,
                         const mxDouble* iData,
                         mwSize num_rows,
                         mwSize num_cols, mwSize
                         nzmax)
```

Arguments   **Inputs**

rowindex_size
    size of `rowindex` array.

rowindex
    Array of row indices of non-zero elements.

colindex_size
    Size of `colindex` array.

colindex
    Array of column indices of non-zero elements.

data_size
    Size of data array(s).

rData
    Real part of data associated with non-zero row and column indices.

iData
    Imaginary part of data associated with non-zero row and column indices.

num_rows
    Number of rows in matrix.

static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxDouble* rdata, mxDouble* idata, num_rows, num_cols, nzmax)

num_cols

Number of columns in matrix.

nzmax

Reserved storage for sparse matrix. If `nzmax` is zero, storage will be set to `max{rowindex_size, colindex_size, data_size}`.

Outputs

mwArray

`mwArray` containing the sparse array

Description   The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated through out the construction of the matrix.

If any element of the `rowindex` or `colindex` array is greater than the specified values in `num_rows`, `num_cols`, respectively, then an exception is thrown.

Example   This example constructs a complex tridiagonal matrix:

```
double rdata[] =
  {2.0, -1.0, -1.0, 2.0, -1.0, -1.0, 2.0, -1.0, -1.0, 2.0};
double idata[] =
  {20.0, -10.0, -10.0, 20.0, -10.0, -10.0, 20.0, -10.0,
                                            -10.0, 20.0};
mwIndex row_tridiag[] =
  {1,    2,    1,   2,    3,    2,   3,   4,    3,    4};
mwIndex col_tridiag[] =
  {1,    1,    2,   2,    2,    3,   3,   3,    4,    4};

mwArray mysparse = mwArray::NewSparse(10, row_tridiag,
                                      10, col_tridiag,
                                      10, rdata,
                                      idata, 4, 4, 10);
std::cout << mysparse << std::endl;
```

static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxDouble* rdata, mxDouble* idata, num_rows, num_cols, nzmax)

It displays the following output to the screen:

```
(1,1)      2.0000 +20.0000i
(2,1)     -1.0000 -10.0000i
(1,2)     -1.0000 -10.0000i
(2,2)      2.0000 +20.0000i
(3,2)     -1.0000 -10.0000i
(2,3)     -1.0000 -10.0000i
(3,3)      2.0000 +20.0000i
(4,3)     -1.0000 -10.0000i
(3,4)     -1.0000 -10.0000i
(4,4)      2.0000 +20.0000i
```

static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxDouble* rdata, mxDouble* idata, mwsize, nzmax)

Purpose	Creates complex sparse matrix of type `double` with number of rows and columns inferred (not explicitly specified) from input data.

C++ Signature

```
static mwArray NewSparse(mwSize rowindex_size,
                         const mwIndex* rowindex,
                         mwSize colindex_size,
                         const mwIndex* colindex,
                         mwSize data_size,
                         const mxDouble* rData,
                         const mxDouble* iData,
                         mwSize nzmax)
```

Arguments   **Inputs**

rowindex_size
    Size of `rowindex` array.

rowindex
    Array of row indices of non-zero elements.

colindex_size
    Size of `colindex` array.

colindex
    Array of column indices of non-zero elements.

data_size
    Size of data array(s).

rData
    Data associated with non-zero row and column indices.

IData
    Imaginary part of data associated with non-zero row and column indices.

nzmax
    Reserved storage for sparse matrix. If nzmax is zero, storage will be set to `max{rowindex_size, colindex_size, data_size}`.

static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxDouble* rdata, mxDouble* idata, mwsize, nzmax)

Outputs

mwArray
  mwArray containing the sparse array.

Description   The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated through out the construction of the matrix.

The number of rows and columns in the created matrix are calculated form the input rowindex and colindex arrays as num_rows = max{rowindex}, num_cols = max{colindex}.

Example   This example constructs a complex matrix by inferring dimensions and storage allocation from the input data. The matrix used is taken from the example that creates a complex sparse matrix of type double with a specified number of rows and columns.

```
mwArray mysparse =
   mwArray::NewSparse(10, row_tridiag,
                      10, col_tridiag,
                      10, rdata, idata,
                      0);
std::cout << mysparse << std::endl;

(1,1)      2.0000 +20.0000i
(2,1)     -1.0000 -10.0000i
(1,2)     -1.0000 -10.0000i
(2,2)      2.0000 +20.0000i
(3,2)     -1.0000 -10.0000i
(2,3)     -1.0000 -10.0000i
(3,3)      2.0000 +20.0000i
(4,3)     -1.0000 -10.0000i
(3,4)     -1.0000 -10.0000i
(4,4)      2.0000 +20.0000i
```

static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxLogical* rdata, num_rows, num_cols, nzmax)

Purpose   Creates logical sparse matrix with specified number of rows and columns.

C++
Signature
```
static mwArray NewSparse(mwSize rowindex_size,
                         const mwIndex* rowindex,
                         mwSize colindex_size,
                         const mwIndex* colindex,
                         mwSize data_size,
                         const mxLogical* rData,
                         mwSize num_rows,
                         mwSize num_cols,
                         mwSize nzmax)
```

Arguments   **Inputs**

rowindex_size
     Size of rowindex array.

rowindex
     Array of row indices of non-zero elements.

colindex_size
     Size of colindex array.

colindex
     Array of column indices of non-zero elements.

data_size
     Size of data array(s).

rData
     Data associated with non-zero row and column indices.

num_rows
     Number of rows in matrix.

num_cols
     Number of columns in matrix.

static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxLogical* rdata, num_rows, num_cols, nzmax)

nzmax
> Reserved storage for sparse matrix. If nzmax is zero, storage will be set to max{rowindex_size, colindex_size, data_size}.

Outputs

mwArray
> mwArray containing the sparse array.

Description The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated throughout the construction of the matrix.

If any element of the rowindex or colindex array is greater than the specified values in num_rows, num_cols, respectively, then an exception is thrown.

Example This example creates a sparse logical 4 X 4 tridiagonal matrix, assigning true to each non-zero value:

```
mxLogical one = true;
mwIndex row_tridiag[] =
      {1,    2,    1,    2,    3,
       2,    3,    4,    3,    4};
mwIndex col_tridiag[] =
      {1,    1,    2,    2,    2,
       3,    3,    3,    4,    4};

mwArray mysparse =
      mwArray::NewSparse(10, row_tridiag,
                         10, col_tridiag,
                         1, &one,
                         4, 4, 10);
std::cout << mysparse << std::endl;

(1,1)        1
(2,1)        1
(1,2)        1
```

D-103

static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxLogical* rdata, num_rows, num_cols, nzmax)

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

static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxLogical* rData, nzmax)

Purpose   Creates logical sparse matrix with number of rows and columns inferred (not explicitly specified) from input data.

C++ Signature

```
static mwArray NewSparse(mwSize rowindex_size,
                         const mwIndex* rowindex,
                         mwSize colindex_size,
                         const mwIndex* colindex,
                         mwSize data_size,
                         const mxLogical* rData,
                         mwSize nzmax)
```

Arguments   **Inputs**

rowindex_size
    Size of rowindex array.

rowindex
    Array of row indices of non-zero elements.

colindex_size
    Size of colindex array.

colindex
    Array of column indices of non-zero elements.

data_size
    Size of data array(s).

rData
    Data associated with non-zero row and column indices.

nzmax
    Reserved storage for sparse matrix. If nzmax is zero, storage will be set to max{rowindex_size, colindex_size, data_size}.

Outputs

mwArray
    mwArray containing the sparse array.

static mwArray NewSparse(rowindex_size, mwIndex* rowindex, colindex_size, mwIndex* colindex, data_size, mxLogical* rData, nzmax)

Description

The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated through out the construction of the matrix.

The number of rows and columns in the created matrix are calculated form the input rowindex and colindex arrays as num_rows = max {rowindex}, num_cols = max {colindex}.

Example

This example uses the data from the first example, but allows the number of rows, number of columns, and allocated storage to be calculated from the input data:

```
mwArray mysparse =
    mwArray::NewSparse(10, row_tridiag,
                        10, col_tridiag,
                        1, &one,
                        0);
std::cout << mysparse << std::endl;

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

static mwArray NewSparse (num_rows, num_cols, nzmax, mxClassID mxID, mxComplexity cmplx = mxREAL)

Purpose	Creates an empty sparse matrix

C++ Signature

```
static mwArray NewSparse (mwSize num_rows,
                          mwSize num_cols,
                          mwSize nzmax,
                          mxClassID mxID,
                          mxComplexity cmplx = mxREAL)
```

Arguments

Inputs

num_rows
    Number of rows in matrix.

num_cols
    Number of columns in matrix.

nzmax
    Reserved storage for sparse matrix.

mxID
    Type of data to store in matrix. Currently, sparse matrices of type double precision and logical are supported. Pass mxDOUBLE_CLASS to create a double precision sparse matrix. Pass mxLOGICAL_CLASS to create a logical sparse matrix.

cmplx
    Optional. Complexity of matrix. Pass mxCOMPLEX to create a complex sparse matrix and mxREAL to create a real sparse matrix. This argument may be omitted, in which case the default complexity is real.

Outputs

mwArray
    mwArray containing the sparse array.

Description   This method returns an empty sparse matrix. All elements in an empty sparse matrix are initially zero, and the amount of allocated storage for non-zero elements is specified by nzmax.

static mwArray NewSparse (num_rows, num_cols, nzmax, mxClassID mxID, mxComplexity cmplx = mxREAL)

Example   This example constructs a real 3 X 3 empty sparse matrix of type double with reserved storage for 4 non-zero elements:

```
mwArray mysparse = mwArray::NewSparse
                   (3, 3, 4, mxDOUBLE_CLASS);
std::cout << mysparse << std::endl;

All zero sparse: 3-by-3
```

Purpose	Get value of NaN (Not-a-Number)
C++ Syntax	`#include "mclcppclass.h"` `double x = mwArray::GetNaN();`
Arguments	None
Return Value	The value of NaN (Not-a-Number) on your system.
Description	Call mwArray::GetNaN to return the value of NaN for your system. NaN is the IEEE arithmetic representation for Not-a-Number. Certain mathematical operations return NaN as a result, for example:

- 0.0/0.0

- Inf-Inf

The value of NaN is built in to the system; you cannot modify it.

static double GetEps()

Purpose	Get value of eps
C++ Syntax	`#include "mclcppclass.h"` `double x = mwArray::GetEps();`
Arguments	None
Return Value	The value of the MATLAB eps variable.
Description	Call `mwArray::GetEps` to return the value of the MATLAB eps variable. This variable is the distance from 1.0 to the next largest floating-point number. Consequently, it is a measure of floating-point accuracy. The MATLAB pinv and rank functions use eps as a default tolerance.

Purpose   Get value of Inf (infinity)

C++   `#include "mclcppclass.h"`
Syntax   `double x = mwArray::GetInf();`

Arguments   None

Return   The value of Inf (infinity) on your system.
Value

Description Call `mwArray::GetInf` to return the value of the MATLAB internal Inf variable. Inf is a permanent variable representing IEEE arithmetic positive infinity. The value of Inf is built into the system; you cannot modify it.

Operations that return Inf include

- Division by 0. For example, 5/0 returns Inf.

- Operations resulting in overflow. For example, `exp(10000)` returns Inf because the result is too large to be represented on your machine.

static bool IsFinite(double x)

Purpose	Test if value is finite and return true if value is finite
C++ Syntax	`#include "mclcppclass.h"` `bool x = mwArray::IsFinite(1.0);        // Returns true`
Arguments	Value to test for finiteness.
Return Value	Result of test.
Description	Call mwArray::IsFinite to determine whether or not a value is finite. A number is finite if it is greater than -Inf and less than Inf.

Purpose   Test if value is infinite and return `true` if value is infinite

C++   `#include "mclcppclass.h"`
Syntax   `bool x = mwArray::IsInf(1.0);`   `// Returns false`

Arguments   Value to test for infinity.

Return   Result of test.
Value

Description   Call `mwArray::IsInf` to determine whether or not a value is equal to
                     infinity or minus infinity. MATLAB stores the value of infinity in a
                     permanent variable named `Inf`, which represents IEEE arithmetic
                     positive infinity. The value of the variable, `Inf`, is built into the system;
                     you cannot modify it.

                     Operations that return infinity include

                     • Division by 0. For example, 5/0 returns infinity.

                     • Operations resulting in overflow. For example, `exp(10000)` returns
                       infinity because the result is too large to be represented on your
                       machine. If the value equals `NaN` (Not-a-Number), then `mxIsInf`
                       returns `false`. In other words, `NaN` is not equal to infinity.

static bool IsNaN(double x)

Purpose	Test if value is NaN (Not-a-Number) and return true if value is NaN
C++ Syntax	```#include "mclcppclass.h"``` ```bool x = mwArray::IsNaN(1.0); // Returns false```
Arguments	Value to test for NaN.
Return Value	Result of test.
Description	Call mwArray::IsNaN to determine whether or not the value is NaN. NaN is the IEEE arithmetic representation for Not-a-Number. NaN is obtained as a result of mathematically undefined operations such as

- 0.0/0.0

- Inf-Inf

The system understands a family of bit patterns as representing NaN. In other words, NaN is not a single value, rather it is a family of numbers that the MATLAB software (and other IEEE-compliant applications) use to represent an error condition or missing data.

Index