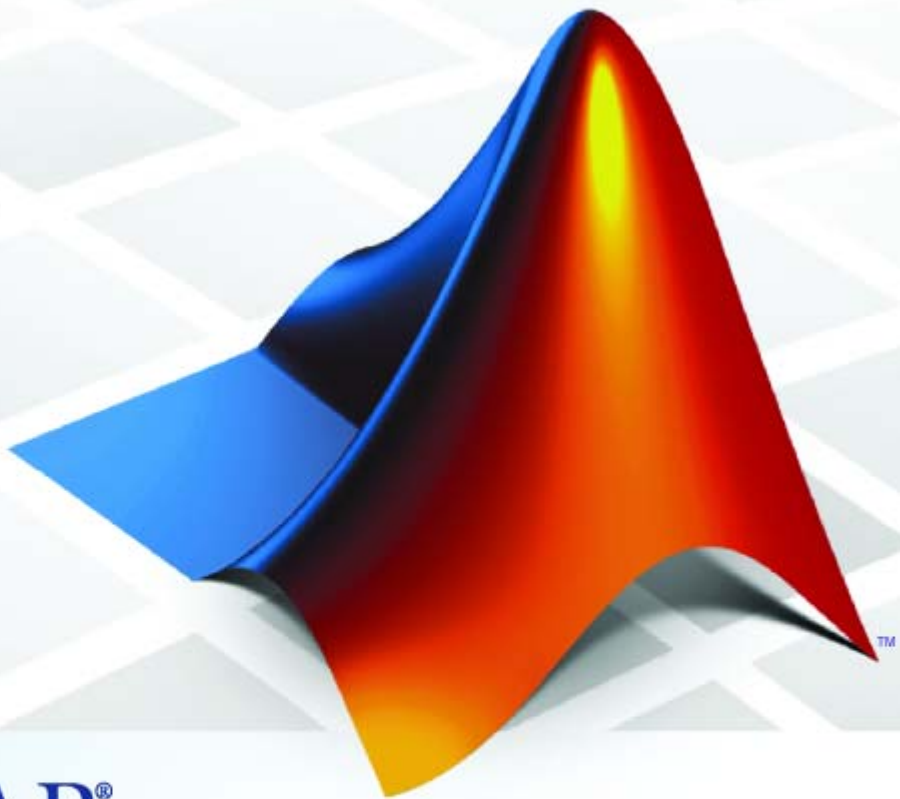


MATLAB® Compiler™ 4

User's Guide



MATLAB®

How to Contact The MathWorks



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



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



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

MATLAB® Compiler™ User's Guide

© COPYRIGHT 1995–2009 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

September 1995	First printing	
March 1997	Second printing	
January 1998	Third printing	Revised for Version 1.2
January 1999	Fourth printing	Revised for Version 2.0 (Release 11)
September 2000	Fifth printing	Revised for Version 2.1 (Release 12)
October 2001	Online only	Revised for Version 2.3
July 2002	Sixth printing	Revised for Version 3.0 (Release 13)
June 2004	Online only	Revised for Version 4.0 (Release 14)
August 2004	Online only	Revised for Version 4.0.1 (Release 14+)
October 2004	Online only	Revised for Version 4.1 (Release 14SP1)
November 2004	Online only	Revised for Version 4.1.1 (Release 14SP1+)
March 2005	Online only	Revised for Version 4.2 (Release 14SP2)
September 2005	Online only	Revised for Version 4.3 (Release 14SP3)
March 2006	Online only	Revised for Version 4.4 (Release 2006a)
September 2006	Online only	Revised for Version 4.5 (Release 2006b)
March 2007	Online only	Revised for Version 4.6 (Release 2007a)
September 2007	Seventh printing	Revised for Version 4.7 (Release 2007b)
March 2008	Online only	Revised for Version 4.8 (Release 2008a)
October 2008	Online only	Revised for Version 4.9 (Release 2008b)
March 2009	Online only	Revised for Version 4.10 (Release 2009a)
September 2009	Online only	Revised for Version 4.11 (Release 2009b)

Getting Started

1

Product Overview	1-2
What Does This Product Do?	1-2
When To Use MATLAB® Compiler	1-2
When Not To Use MATLAB® Compiler	1-3
How Do I Use This Product?	1-4
How Does This Product Work?	1-4
What Is The MATLAB Compiler Runtime (MCR)?	1-5
Before You Use MATLAB® Compiler	1-6
Understand Your Role in the Application Deployment Process	1-6
Verify Your Knowledge Base	1-8
Install Required Products	1-9
Select Your C or C++ Compiler with mbuild -setup	1-9
Deploying with the Magic Square Example	1-10
About This Example	1-10
Magic Square Example: MATLAB Programmer Tasks ...	1-11
Magic Square Example: C/C++ Programmer Tasks	1-16
For More Information	1-20

Installation and Configuration

2

Requirements	2-2
System Requirements	2-2
Supported Third-Party Compilers	2-2
Installation	2-4

Installing MATLAB® Compiler	2-4
Installing an ANSI C or C++ Compiler	2-5
Configuration	2-7
About the mbuild Utility	2-7
Configuring an ANSI C or C++ Compiler	2-7
Supported Compiler Restrictions	2-11
Options Files	2-12
Locating the Options File	2-12
Changing the Options File	2-13

Compilation Process

3

Overview of MATLAB® Compiler Technology	3-2
MATLAB® Compiler Runtime	3-2
Component Technology File	3-2
Build Process	3-3
Input and Output Files	3-6
Standalone Executable	3-6
C Shared Library	3-7
C++ Shared Library	3-9
Macintosh 64 (Maci64)	3-11
Working with MATLAB Data Files Using Load and Save	3-12
Example: Using Load/Save Functions To Process MATLAB Data for Deployed Applications	3-12

Overview	4-2
Deploying to Programmers	4-3
Steps by the Programmer to Deploy to Programmers	4-3
What Software Does a Programmer Need?	4-4
Ensuring Memory for Deployed Applications	4-8
Deploying to End Users	4-9
Steps by the Programmer to Deploy to End Users	4-9
What Software Does the End User Need?	4-13
Using Relative Paths with Project Files	4-16
Porting Generated Code to a Different Platform	4-16
Extracting a CTF Archive Without Executing the Component	4-17
Dependency Analysis Function (depfun) and User Interaction with the Compilation Path	4-17
Ensuring Memory for Deployed Applications	4-20
Working with the MCR	4-21
Understanding the MCR	4-21
Installing the MCR and MATLAB on the Same Machine ..	4-22
Installing Multiple MCRs on One Machine	4-24
Retrieving MCR Attributes	4-24
Improving Data Access Using the MCR User Data Interface	4-26
Displaying MCR Initialization Start-Up and Completion Messages For Users	4-33
Deploying a Standalone Application on a Network Drive	4-35
MATLAB® Compiler Deployment Messages	4-36
Using MATLAB® Compiler Generated DLLs in Windows Services	4-37

Reserving Memory for Deployed Applications with MATLAB Memory Shielding	4-38
What Is MATLAB Memory Shielding and When Should You Use It?	4-38
Requirements for Using MATLAB Memory Shielding	4-39
Invoking MATLAB Memory Shielding for Your Deployed Application	4-39

Compiler Commands

5

Command Overview	5-2
Compiler Options	5-2
Combining Options	5-2
Conflicting Options on the Command Line	5-3
Using File Extensions	5-3
Using Macros to Simplify Compilation	5-5
Macro Options	5-5
Working With Macro Options	5-5
Using Path Names	5-8
Using Bundle Files	5-9
Using Wrapper Files	5-11
What Are Wrapper Files?	5-11
Main File Wrapper	5-11
C Library Wrapper	5-12
C++ Library Wrapper	5-13
Interfacing M-Code to C/C++ Code	5-14
Overview	5-14
Code Proper Return Types From C and C++ Methods	5-14
C Example	5-14
C++ Example	5-16

Overriding Default CTF Archive Embedding Using the MCR Component Cache	5-20
Using Pragmas	5-22
Using feval	5-22
Example: Using %#function	5-22
Using mxArray	5-24
Script Files	5-25
Converting Script M-Files to Function M-Files	5-25
Including Script Files in Deployed Applications	5-26
Compiler Tips	5-27
Calling Built-In Functions from C or C++	5-27
Calling a Function from the Command Line	5-28
Using MAT-Files in Deployed Applications	5-28
Compiling a GUI That Contains an ActiveX Control	5-28
Debugging MATLAB® Compiler Generated Executables ..	5-29
Deploying Applications That Call the Java Native Libraries	5-29
Locating .fig Files in Deployed Applications	5-29
Blocking Execution of a Console Application That Creates Figures and Terminating Figures by Force	5-30
Passing Arguments to and from a Standalone Application	5-31
Using Graphical Applications in Shared Library Targets ..	5-33
Using the VER Function in a Compiled MATLAB Application	5-33

Standalone Applications

6

Introduction	6-2
C Standalone Application Target	6-3
Compiling the Application	6-3
Testing the Application	6-3

Deploying the Application	6-4
Running the Application	6-6
Coding with M-Files Only	6-8
M-File Advantages	6-8
Example	6-8
Mixing M-Files and C or C++	6-10
Examples Overview	6-10
Simple Example	6-10
Advanced C Example	6-15

Libraries

7

Introduction	7-2
Addressing mxArray Arrays Above the 2 GB Limit	7-3
C Shared Library Target	7-4
C Shared Library Wrapper	7-4
C Shared Library Example	7-4
Calling a Shared Library	7-11
C++ Shared Library Target	7-18
C++ Shared Library Wrapper	7-18
C++ Shared Library Example	7-18
MATLAB® Compiler Generated Interface Functions ..	7-24
Functions in the Shared Library	7-24
Type of Application	7-24
Structure of Programs That Call Shared Libraries	7-26
Library Initialization and Termination Functions	7-27
Print and Error Handling Functions	7-28
Functions Generated from M-Files	7-30
Retrieving MCR State Information While Using Shared Libraries	7-33

Using C/C++ Shared Libraries on a Mac OS X System ..	7-34
About Memory Management and Cleanup	7-38
Overview	7-38
Passing mxArray's to Shared Libraries	7-38

Troubleshooting

8

Introduction	8-2
Common Issues	8-3
Failure Points and Possible Solutions	8-4
How to Use this Section	8-4
Does the Failure Occur During Compilation?	8-4
Does the Failure Occur When Testing Your Application? ..	8-8
Does the Failure Occur When Deploying the Application to End Users?	8-11
mbuild	8-14
MATLAB® Compiler	8-16
Deployed Applications	8-20

Limitations and Restrictions

9

Limitations About What May Be Compiled	9-2
Compiling MATLAB and Toolboxes	9-2
Fixing Callback Problems: Missing Functions	9-3
Finding Missing Functions in an M-File	9-5
Suppressing Warnings on the UNIX System	9-5

Cannot Use Graphics with the -nojvm Option	9-6
Cannot Create the Output File	9-6
No M-File Help for Compiled Functions	9-6
No MCR Versioning on Mac OS X	9-6
Older Neural Networks Not Deployable with MATLAB® Compiler	9-7
Restrictions on Calling PRINTDLG with Multiple Arguments in Compiled Mode	9-7
Compiling a Function with WHICH Does Not Search Current Working Directory	9-8
Unsupported Functions	9-9

Reference Information

10

Directories Required for Development and Testing ...	10-2
Overview	10-2
Path for Java Development on All Platforms	10-2
Path Modifications Required for Accessibility	10-2
Windows Settings for Development and Testing	10-3
UNIX Settings for Development and Testing	10-3
Directories Required for Run-Time Deployment	10-5
General Path Guidelines	10-5
Path for Java Applications on All Platforms	10-5
Windows Path for Run-Time Deployment	10-5
UNIX Paths for Run-Time Deployment	10-6
MATLAB® Compiler Licensing	10-8
Using MATLAB® Compiler Licenses for Development	10-8
Using MCR Installer Command Line Options	10-9
Overview	10-9
Displaying MCR Installer Location and Related Information	10-9
Accessing MCR Installer Command Line Options on Windows Systems	10-10

Accessing MCR Installer Command Line Options on UNIX and Linux Systems	10-12
---	-------

Function Reference

11

Pragmas	11-2
Command-Line Tools	11-2
API Functions	11-3

Functions — Alphabetical List

12

MATLAB® Compiler Quick Reference

A

Common Uses of MATLAB® Compiler	A-2
Create a Standalone Application	A-2
Create a Library	A-2
<code>mcc</code>	A-4

Error and Warning Messages

B

About Error and Warning Messages	B-2
--	-----

Compile-Time Errors	B-3
Warning Messages	B-6
depfun Errors	B-9
About depfun Errors	B-9
MCR/Dispatcher Errors	B-9
XML Parser Errors	B-9
depfun-Produced Errors	B-10

C++ Utility Library Reference

C	Primitive Types	C-2
	Utility Classes	C-3
	mwString Class	C-4
	About mwString	C-4
	Constructors	C-4
	Methods	C-4
	Operators	C-4
	mwException Class	C-20
	About mwException	C-20
	Constructors	C-20
	Methods	C-20
	Operators	C-20
	mwException Class Functions	C-21
	mwArray Class	C-29
	About mwArray	C-29
	Constructors	C-29
	Methods	C-30
	Operators	C-31
	Static Methods	C-32

mwArray Class Functions **C-33**

Index

Getting Started

- “Product Overview” on page 1-2
- “Before You Use MATLAB® Compiler” on page 1-6
- “Deploying with the Magic Square Example” on page 1-10
- “For More Information” on page 1-20

Product Overview

In this section...
“What Does This Product Do?” on page 1-2
“When To Use MATLAB® Compiler” on page 1-2
“When Not To Use MATLAB® Compiler” on page 1-3
“How Do I Use This Product?” on page 1-4
“How Does This Product Work?” on page 1-4
“What Is The MATLAB Compiler Runtime (MCR)?” on page 1-5

What Does This Product Do?

MATLAB® Compiler™ compiles a MATLAB® application into a standalone application or software component. The act of compiling this code is sometimes 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 your code into a different language. If you are building a standalone application, MATLAB Compiler produces an executable for your end users. If you integrate into C or C++, MATLAB Compiler provides an interface to use your code as a shared library. If you integrate into other development languages, MATLAB builder products (available separately) let you package your MATLAB applications as software components. You are able to use 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 royalty-free, standalone executables and software components
- Incorporate MATLAB-based algorithms into applications developed using other languages and technologies
- Encrypt and protect MATLAB code

When Not To Use MATLAB Compiler

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

To...	Use...
<ul style="list-style-type: none"> • Generate readable, efficient, and embeddable C code from M-code • Generate MEX functions from M-code for rapid prototyping and verification of generated C code within MATLAB • Integrate M-code into Simulink® • Speed up fixed-point M-code • Generate hardware description language (HDL) from M-code 	<i>Embedded MATLAB™ User's Guide</i>
<ul style="list-style-type: none"> • Integrate custom C code into MATLAB with MEX files • Call MATLAB from C and Fortran programs 	<i>MATLAB External Interfaces</i>
Deploy Java components into enterprise computing environments and to MATLAB users	<i>MATLAB® Builder™ JA User's Guide</i>

To...	Use...
Deploy .NET and COM components into enterprise computing environments and to MATLAB users	<i>MATLAB Builder NE User's Guide</i>
Deploy Excel add-ins to enterprise computing environments and to MATLAB users	<i>MATLAB Builder EX User's Guide</i>

How Do I Use This Product?

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

How Does This Product Work?

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

MATLAB Compiler Generated Applications and Libraries

When you package and distribute applications and libraries that MATLAB Compiler generates, you include the MATLAB Compiler Runtime (MCR). You can include a set of supporting files generated by MATLAB Compiler. You also set the system paths on the target machine so your application finds the MCR and supporting files.

The MCR is an engine for execution of compiled MATLAB code. As such, when a compiled application runs, there is sometimes a delay as the MCR is initialized. There can be a delay the first time you run the application, although typically less than you experience running on the MATLAB desktop.

An application or library generated by MATLAB Compiler has two parts: a platform-specific binary file and an archive file containing the encrypted MATLAB application and data. An application binary file consists of a main function. By contrast, a library binary consists of multiple functions for exporting.

Wrapper Files

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

What Is The MATLAB Compiler Runtime (MCR)?

The MATLAB Compiler Runtime is a standalone set of shared libraries that enable the execution of M-files, even on computers without an installed version of MATLAB.

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. Install it on target machines by running the self-extracting package executable. For more information on the MCR and the MCR Installer, see [What Is the MCR and How Do I Get It?](#)

Before You Use MATLAB Compiler

In this section...
“Understand Your Role in the Application Deployment Process” on page 1-6
“Verify Your Knowledge Base” on page 1-8
“Install Required Products” on page 1-9
“Select Your C or C++ Compiler with mbuild -setup” on page 1-9

Understand 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 application or shared library.

For example, you analyze user requirements and satisfy them by writing a program in M-code. You can also implement the infrastructure to deploy an application to users in computing environments different from your own. In smaller organizations, you find one person responsible for performing tasks associated with multiple roles. 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 “Deploying with the Magic Square Example” on page 1-10 in this chapter.

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

Application Deployment Roles, Tasks, and References

Role	Tasks	References
MATLAB Programmer	<ul style="list-style-type: none">• Understand end-user business requirements and the mathematical models they support• Write M-code.• Build an executable component with MATLAB tools (usually with support from an IT programmer or Front-End Developer).• Package the component for distribution to customers.• Pass the packaged component to the IT programmer or Front-End Developer for rollout and further integration into the end-user environment.	“Magic Square Example: MATLAB Programmer Tasks” on page 1-11

Application Deployment Roles, Tasks, and References (Continued)

Role	Tasks	References
C or C++ Programmer	<ul style="list-style-type: none"> • Set the target computer path so the system can support the compiled code at run time. • Install the MCR on target computers by running the MCR Installer. • Ensure that the final standalone application or library executes reliably in the customer environment. 	“Magic Square Example: C/C++ Programmer Tasks” on page 1-16
External user	Executes the solution created by MATLAB and Front-End Developers.	Run the deployed application (outside the scope of this document).

Verify Your Knowledge Base

Using the MATLAB Compiler product requires the following knowledge:

- If your job function is MATLAB programmer:
 - A basic knowledge of MATLAB, and how to work with cell arrays and structures
- If your job function is C or C++ developer:
 - Exposure to the C or C++ programming languages
 - Procedural or object-oriented programming concepts

Install Required Products

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 Chapter 2, “Installation and Configuration”.

Select Your C or C++ Compiler with `mbuild -setup`

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

```
mbuild -setup
```

For more information about `mbuild -setup`, see Chapter 2, “Installation and Configuration”.

If you need information about writing M-files, see MATLAB Programming, which is part of MATLAB documentation.

Deploying with the Magic Square Example

In this section...
“About This Example” on page 1-10
“Magic Square Example: MATLAB Programmer Tasks” on page 1-11
“Magic Square Example: C/C++ Programmer Tasks” on page 1-16

About This Example

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.

The Magic Square example in this section shows you how to:

- Use MATLAB Compiler to create and package a simple standalone application that compiles an M-file, `magicsquare.m`
- Access the examples provided with MATLAB Compiler.

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


Tip To set `examplesDir` to the path `matlabroot\extern\examples\compiler`, [click here](#).

Command-Line Alternative

The examples use the `deploytool` GUI, a graphical front-end interface to MATLAB Compiler software. You can perform these tasks using the command-line interface to MATLAB Compiler software. See “Using the Command Line (`mcc`) to Create Standalone Applications and Shared Libraries” on page 1-16 and the `mcc` reference page for complete reference information.

Magic Square Example: MATLAB Programmer Tasks

The MATLAB programmer performs the tasks described in the following table.

 MATLAB Programmer	MATLAB expert No IT experience No access to IT systems	Develops model Uses the builder tools to create a component that is given to the business service developer
---	--	--

Key Tasks for the MATLAB Programmer

Task	Reference
Start the product.	“Starting the Deployment Tool” on page 1-12
Prepare to run the example by copying the MATLAB example files into a work folder.	“Copying the Example Files” on page 1-12
Test the M-code to ensure that it is suitable for deployment.	“Testing the M-File You Want To Deploy” on page 1-13
Create a standalone application or shared library (encapsulating your M-code in a C or C++ class) by running the Build function in <code>deploytool</code> .	“Creating a Deployable Standalone Application or Shared Library” on page 1-14
Run the Packaging Tool to bundle your standalone application or shared library with the additional files you selected.	“Packaging Your Deployment Application (Optional)” on page 1-15
Copy the output from the Packaging Tool (the <code>distrib</code> folder).	“Copy the Package You Created (Optional)” on page 1-15

The Magic Square example shows how to create a standalone application, or shared library (`magicsquare`), containing the `magic` class and other files for application deployment. The class encapsulates a MATLAB function which computes a magic square.

The client standalone application or shared library converts the array returned by the function to a native array and displays it on the screen. When you run the `magicsquare` application from the command line, you pass the dimension for the magic square as a command-line argument

Note The examples for the MATLAB Compiler product reside in `matlabroot\extern\examples\compiler`. This example assumes that the work folder is on drive D:.

Starting the Deployment Tool

You can access the MATLAB Compiler product through the Deployment Tool GUI (`deploytool`). Alternately, you can use the `mcc` function of MATLAB Compiler. `deploytool` is the GUI front end for `mcc`, the command that executes MATLAB Compiler.

This tutorial uses `deploytool`. If you want to use `mcc`, see “Using the Command Line (`mcc`) to Create Standalone Applications and Shared Libraries” on page 1-16 . Also see the `mcc` reference page for complete reference information.

To start the Deployment Tool by perform the following steps:

- 1 Start MATLAB.
- 2 Type `deploytool` at the MATLAB command prompt. The `deploytool` GUI opens.

Copying the Example Files

Prepare to run the example by copying needed files into your work area as follows:

- 1 Navigate to `matlabroot\extern\examples\compiler`. `matlabroot` is the MATLAB root folder (where you installed MATLAB). To find the value of this variable on your system, type `matlabroot` at a MATLAB command prompt.

- 2** Create a work folder named `Work` (`D:\Work`). Create a subfolder in your `Work` folder and name it `MagicExample` (`D:\Work\MagicExample`). Avoid using spaces in your folder names, if possible.
- 3** Copy `magicsquare.m` from `matlabroot\extern\examples\compiler` to `D:\Work\MagicExample`.
- 4** To run your resulting client standalone application or shared library, verify that MATLAB can find it. Use the **File > Set Path** option in MATLAB to add the `D:\Work\MagicExample` folder to the MATLAB search path.
- 5** At the MATLAB command prompt, change your working folder to `D:\Work\MagicExample`.

Testing the M-File You Want To Deploy

In this example, you test an M-file (`magicsquare.m`) containing the predefined MATLAB function `magic`. Testing the file provides a baseline to compare to the results of the function as a deployable standalone application or shared library.

- 1** Using MATLAB, locate and open `magicsquare.m`. This file has the following:

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

% Copyright 2003-2007 The MathWorks, Inc.

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

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

```
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

Creating a Deployable Standalone Application or Shared Library

You create a deployable standalone application or shared library by using the Deployment Tool GUI to build a wrapper. This wrapper encloses the sample M-code discussed in “Testing the M-File You Want To Deploy” on page 1-13. To run the Magic Square example, use the following information:

Project Name	MagicExample
File to compile	magicsquare.m

- 1** Create a deployment project. A *project* is a collection of files you bundle together under a project file name (.prj file) for your convenience in the Deployment Tool. Using a project makes it easy for you to build and run an application many times—effectively testing it—until it is ready for deployment.
 - a** Type the name of your project in the **Name** field.
 - b** Enter the location of the project in the **Location** field. Alternately, navigate to the location.
 - c** Select the target for the deployment project from the **Target** drop-down menu.
 - d** Click **OK**.
- 2** On the **Build** tab, add what you want to compile, and any supporting files, to the project.
 - a** Do the following, depending on the type of application you are building:
 - If you are building a C or C++ application, click **Add files**
 - b** Add any supporting files. For example, you can add the following files, as appropriate for your project:
 - 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

If you want to include additional files, in the Shared Resources and Helper Files area, click **Add files/directories**. Click **Open** to select the file or files.

3 When you complete your changes, click the Build button ()

Packaging Your Deployment Application (Optional)

Packaging is bundling the standalone application or shared libraries with additional files for end users. Perform this step using the **Package** tab of `deploytool`. Alternately, copy the contents of the `distrib` folder and the MCR Installer to a local folder of your choice.

- 1** On the **Package** tab, add the MATLAB Compiler Runtime (the MCR) by clicking **Add MCR**.
- 2** Next, add others files useful for end users. The `readme.txt` file contains important information about others files useful for end users. To package additional files or folders, click **Add file/directories**, select the file or folder you want to package, and click **Open**.
- 3** In the Deployment Tool, click the Packaging button ()
- 4** After packaging, the package resides in the `distrib` subfolder. On Windows®, the package is a self-extracting executable. On platforms other than Windows, it is a `.zip` file. Verify that the contents of the `distrib` folder contains the files you specified.

Copy the Package You Created (Optional)

Copy the package that you created from the `distrib` folder to the local folder of your choice or send them to the C/C++ programmer, if applicable.

Using the Command Line (mcc) to Create Standalone Applications and Shared Libraries

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 M-file <code>mymfunction</code>	<code>mcc -m mymfunction.m</code>
	Creates a standalone application named <code>mymfunction.exe</code> on Windows platforms and <code>mymfunction</code> on platforms that are not Windows.
C shared library from the M-files <code>file1.m</code> , <code>file2.m</code> , and <code>file3.m</code>	<code>mcc -B csharedlib:libfiles file1.m file2.m file3.m</code>
	Creates a shared library named <code>libfiles.dll</code> on Windows, <code>libfiles.so</code> on Linux [®] and Solaris [™] , and <code>libfiles.dylib</code> on Mac OS [®] X.
C++ shared library from the M-files <code>file1.m</code> , <code>file2.m</code> , and <code>file3.m</code>	<code>mcc -B cpplib:libfiles file1.m file2.m file3.m</code>
	Creates a shared library named <code>libfiles.dll</code> on Windows, <code>libfiles.so</code> on Linux and Solaris, and <code>libfiles.dylib</code> on Mac OS X.

Magic Square Example: C/C++ Programmer Tasks

The C or C++ programmer performs these tasks.

Key Tasks for the C or C++ Programmer

Task	Reference
Ensure that you have the needed files from the MATLAB Programmer before proceeding.	“Gathering Files Necessary for Deployment” on page 1-17
Distribute the files.	“Distribute to End Users” on page 1-17

Key Tasks for the C or C++ Programmer (Continued)

Task	Reference
Install the MCR on target computers by running the MCR Installer. Update system paths on UNIX systems.	“Install the MCR on Target Computers Without MATLAB and Update System Paths” on page 1-18
Ensure that the final standalone application or library executes reliably in the end-user environment.	“Build and Test” on page 1-19

Gathering Files Necessary for Deployment

Before beginning, verify that you have access to the following files, packaged by the MATLAB Programmer in “Copy the Package You Created (Optional)” on page 1-15. End users who do not have a copy of MATLAB installed need the following:

- MCR Installer. For locations of all MCR Installers, run the `mcrinstaller` command.
- `readme.txt` file

See “Packaging Your Deployment Application (Optional)” on page 1-15 for more information about these files.

Distribute to End Users

If the MATLAB programmer packages the standalone or library (see “Packaging Your Deployment Application (Optional)” on page 1-15), paste the package in a folder on the target machine, and run it. If you are using a `.zip` file bundled with WinZip, unzip and extract the contents to the target machine.

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

Install the MCR on Target Computers Without MATLAB and Update System Paths

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

What Is the MATLAB Compiler Runtime (MCR) and How Do I Get It?

The *MATLAB Compiler Runtime* (MCR) is an execution engine made up of the same shared libraries MATLAB uses to enable the execution of M-files on systems without an installed version of MATLAB. In order to deploy a component, you *package* the MCR along with it. Before you utilize the MCR on a system without MATLAB, run the *MCR installer*.

You have the option to include the MCR with every package generated by the Deployment Tool. On the **Package** tab of the `deploytool` interface, click **Add MCR**.

To install the MCR, perform the following tasks on the target machines:

- 1** If you added the MCR during packaging, open the package to locate the installer. Otherwise, run the command `mcrinstaller` to display the locations where you can download it.
- 2** If you are running on a platform other than Windows, set the system paths on the target machine. Windows paths are set automatically. On Linux and Mac, you can use the run script to set paths.

Setting the paths enables your application to find the MCR.

After you install the MCR, see these resources for more information on working with the various `MWArray` APIs:

- For information on the C++ API, see Appendix C, “C++ Utility Library Reference” in the MATLAB® Compiler™ User’s Guide on page 1.
- For information on the `MWArray` class library (the .NET API), see the MATLAB Builder NE “Documentation Set” at The MathWorks Web site.
- For information on `MWArray` types (the Java™ API), see the MATLAB Builder JA “Documentation Set” at The MathWorks Web site.

For more about the MCR, see “Working with the MCR” on page 4-21 in the MATLAB® Compiler™ User’s Guide on page 1.

Replacement of MCRInstaller.zip and BUILDMCR Functionality. In past releases, you included `MCRInstaller.zip` in your packaged application (created by running the `buildmcr` command). Now, you run the following files, which trigger self-extracting archives, that replace the functionality previously provided by `MCRInstaller.zip`. These files ship with MATLAB Compiler. To get information on the where you can find the MCR Installer, run the command `mcrinstaller`.

Note Since you no longer create `MCRInstaller.zip`, `buildmcr` is no longer supported.

Build and Test

Build and test the standalone application or shared library as you would any application in your environment. After you create and distribute the initial application, you continue to enhance it.

For More Information

About This	Look Here
Detailed information on standalone applications	Chapter 6, “Standalone Applications”
Creating libraries	Chapter 7, “Libraries”
Using the <code>mcc</code> command	Chapter 5, “Compiler Commands”
Troubleshooting	Chapter 8, “Troubleshooting”

Installation and Configuration

This chapter describes the system requirements for MATLAB Compiler. It also contains installation and configuration information for all supported platforms.

When you install your ANSI® C or C++ compiler, you may be required to provide specific configuration details regarding your system. This chapter contains information for each platform that can help you during this phase of the installation process.

- “Requirements” on page 2-2
- “Installation” on page 2-4
- “Configuration” on page 2-7
- “Supported Compiler Restrictions” on page 2-11
- “Options Files” on page 2-12

Requirements

In this section...
“System Requirements” on page 2-2
“Supported Third-Party Compilers” on page 2-2

System Requirements

To install MATLAB Compiler, you must have the proper version of MATLAB installed on your system. The MATLAB Compiler Platform & Requirements page, which is accessible from our Web site, provides this information. MATLAB Compiler imposes no operating system or memory requirements beyond those that are necessary to run MATLAB. MATLAB Compiler consumes a small amount of disk space.

MATLAB Compiler requires that a supported ANSI C or C++ compiler be installed on your system. Certain output targets require particular compilers.

Note Before you use MATLAB Compiler for the first time, you must run `mbuild -setup` to configure your C/C++ compiler to work with MATLAB Compiler.

In general, MATLAB Compiler supports the current release of a third-party compiler and its previous release. Since new versions of compilers are released on a regular basis, it is important to check our Web site for the latest supported compilers.

Supported Third-Party Compilers

For an up-to-date list of all the compilers supported by MATLAB and MATLAB Compiler, see the MathWorks™ Technical Support Department's Technical Notes at

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

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:

- Lcc C version 2.4.1 (included with MATLAB). This is a C-only compiler; it does *not* work with C++.
- Microsoft® Visual C++® (MSVC).

Note 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 .NET Framework (Version 2.0 and higher).

Supported ANSI C and C++ UNIX Compilers

MATLAB Compiler supports the native system compilers on Solaris. On Linux, Linux x86-64, and Mac OS X, MATLAB Compiler supports gcc and g++.

Installation

In this section...
“Installing MATLAB® Compiler ” on page 2-4
“Installing an ANSI C or C++ Compiler” on page 2-5

Installing MATLAB Compiler

MATLAB Compiler requires a supported ANSI C or C++ compiler installed on your system as well. Refer to the “Installing an ANSI C or C++ Compiler” on page 2-5 for more information.

Windows Operating System

To install MATLAB Compiler on Windows, follow the instructions in the Installation Guide for Windows documentation. If you have a license to install MATLAB Compiler, it will appear as one of the installation choices that you can select as you proceed through the installation process.

If MATLAB Compiler does not appear in your list of choices, contact The MathWorks to obtain an updated License File (`license.dat`) for multiuser network installations, or an updated Personal License Password (PLP) for single-user, standard installations.

You can contact The 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 e-mail at service@mathworks.com.

UNIX Operating System

To install MATLAB Compiler on UNIX® workstations, follow the instructions in the Installation Guide for UNIX documentation. If you have a license to install MATLAB Compiler, it appears as one of the installation choices that you can select as you proceed through the installation process. If MATLAB

Compiler does not appear as one of the installation choices, contact The MathWorks to get an updated license file (`license.dat`).

Note In order to run compiled applications from the DOS command prompt, you must reboot (or log out and log in) after installing MATLAB Compiler.

Installing an ANSI C or C++ Compiler

To install your ANSI C or C++ compiler, follow the vendor's instructions that accompany your C or C++ compiler. Be sure to test the C or C++ compiler to make sure it is installed and configured properly. Typically, the compiler vendor provides some test procedures.

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

When you install your C or C++ compiler, you might encounter configuration questions that require you to provide particular details. These tables provide information on some of the more common issues.

Windows Operating System

Issue	Comment
Installation options	We recommend that you do a full installation of your compiler. If you do a partial installation, you may omit a component that MATLAB Compiler relies on.
Installing debugger files	For the purposes of MATLAB Compiler, it is not necessary to install debugger (DBG) files. However, you may need them for other purposes.
Microsoft Foundation Classes (MFC)	This is not required.

Windows Operating System (Continued)

Issue	Comment
16-bit DLLs	This is not required.
ActiveX®	This is not required.
Running from the command line	Make sure 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, you should do it.
Installing Microsoft Visual C++ Version 6.0	If you need to change the location where this compiler is installed, you must 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 installed on your system.	See your system administrator.
Determine the path to your C or C++ compiler.	See your system administrator.
Installing on either Maci or Maci64	Install X Code from installation DVD

Configuration

In this section...
“About the mbuild Utility” on page 2-7
“Configuring an ANSI C or C++ Compiler” on page 2-7

About the mbuild Utility

The `mbuild` script provides an easy way for you to specify an options file that lets you:

- Set the default compiler and linker settings for each supported compiler.
- Change compilers or compiler settings.
- Build your application.

`mbuild` simplifies the process of setting up a C or C++ compiler. Typically, you only need to use the `mbuild` utility's `setup` option once to specify which third-party compiler you want to use. For more information on the `mbuild` utility, see the `mbuild` reference page.

MATLAB Compiler (`mcc`) automatically invokes `mbuild` under certain conditions. In particular, `mcc -m` or `mcc -l` invokes `mbuild` to perform compilation and linking.

See the reference page for more information about `mbuild`. For examples of `mbuild` usage, see “Compiling the Driver Application” on page 7-21.

Configuring an ANSI C or C++ Compiler

Compiler Options Files

Options files contain flags and settings that control the operation of your installed C and C++ compiler. Options files are compiler-specific, i.e., there is a unique options file for each supported C/C++ compiler, which The MathWorks provides.

When you select a compiler to use with MATLAB Compiler, the corresponding options file is activated on your system. To select a default compiler, use

```
mbuild -setup
```

Additional information on the options files is provided in this chapter for those users who may need to modify them to suit their own needs. Many users never have to be concerned with the inner workings of the options files and only need the `setup` option to initially designate a C or C++ compiler. If you need more information on options files, see “Options Files” on page 2-12.

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

Windows. Executing the command on Windows gives

```
Please choose your compiler for building standalone
MATLAB applications:
```

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

```
Select a compiler:
```

```
[1] Lcc-win32 C 2.4.1
[2] Microsoft Visual C++ 6.0
[3] Microsoft Visual C++ .NET 2003
[4] Microsoft Visual C++ 2005 SP1
[5] Microsoft Visual C++ 2008
[6] Microsoft Visual C++ 2008 Express
```

```
[0] None
```

The preconfigured options files that are included with MATLAB for Windows are shown below.

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

Options File	Compiler
lcccomp.bat	Lcc C, Version 2.4.1 (included with MATLAB)
msvc60comp.bat msvc80comp.bat	Microsoft Visual C/C++, Version 6.0 Microsoft Visual C/C++, Version 8.0 Microsoft Visual C/C++, Version 8.0 Express Edition
msvc90comp.bat	Microsoft Visual C/C++, Version 9.0 Microsoft Visual C/C++, Version 9.0 Express Edition

UNIX. Executing the command on UNIX gives

```
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 that is included with MATLAB for UNIX is `mbuildopts.sh`, which uses the system native ANSI compiler for Solaris and `gcc` for Linux and Macintosh®.

Supported Compiler Restrictions

The known restrictions regarding the use of supported compilers on the Windows operating system are:

- The LCC C compiler does not support C++ or versions of Windows other than 32-bit.
- The only compiler that supports the building of COM objects and Excel plug-ins is Microsoft Visual C/C++.
- The only compiler that supports the building of .NET objects is the Microsoft Visual C# Compiler for the .NET Framework (Version 2.0 and higher).

Options Files

In this section...
“Locating the Options File” on page 2-12
“Changing the Options File” on page 2-13

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 (see “User Profile Directory Under Windows” on page 2-12 for more information about this folder)

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

User Profile Directory Under Windows. The Windows `user profile` folder is a folder that contains user-specific information such as desktop appearance, recently used files, and **Start** menu items. The `mbuild` utility stores its options files, `compopts.bat`, which is created during the `-setup` process, 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`. Under Windows with user profiles disabled, your `user profile` folder is `%windir%`. You can determine whether or not 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 no options file is found, `mbuild` displays an error message.

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 so that the new compiler is used every time. To reset your C or C++ compiler for future sessions, use

```
mbuild -setup
```

Windows Operating System

Modifying the Options File. You can use of 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** Use `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 will use the new options file every time with your modified settings.

UNIX Operating System

The `setup` option creates a user-specific, `matlab` folder in your individual home folder and copies the appropriate options file to the folder. (If the folder already exists, a new one is not created.) This `matlab` folder is used for your individual options files only; each user can have his or her own default options

files (other MATLAB products may place options files in this folder). Do not confuse these user-specific `matlab` folders with the system `matlab` folder, where MATLAB is installed.

Modifying the Options File. You can use the `setup` option to change your options file settings on UNIX. For example, if you want to make a change to the current linker settings, or you want to disable a particular set of warnings, you should use the `setup` option.

To modify your options file on the Linux operating system:

- 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.

This sets your default compiler's options file to your specific version.

Compilation Process

This chapter provides an overview of how MATLAB Compiler works. In addition, it lists the various sets of input and output files used by this product.

- “Overview of MATLAB® Compiler Technology ” on page 3-2
- “Input and Output Files” on page 3-6
- “Working with MATLAB Data Files Using Load and Save” on page 3-12

Overview of MATLAB Compiler Technology

In this section...
“MATLAB® Compiler Runtime” on page 3-2
“Component Technology File” on page 3-2
“Build Process” on page 3-3

MATLAB Compiler Runtime

MATLAB Compiler 4 uses the MATLAB Compiler Runtime (MCR), which is a standalone set of shared libraries that enable the execution of M-files. The MCR provides complete support for all features of the MATLAB language.

Note 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.

Component Technology File

MATLAB Compiler also embeds a Component Technology File (CTF) archive in each generated binary to house the deployable package. All M-files are encrypted in the CTF archive using the Advanced Encryption Standard (AES) cryptosystem where symmetric keys are protected by 1024-bit RSA keys.

Each application or shared library produced by MATLAB Compiler has an associated CTF archive. The archive contains all the MATLAB based content (M-files, MEX-files, etc.) associated with the component. If you choose to extract the CTF archive as a separate file (see “Overriding Default CTF Archive Embedding Using the MCR Component Cache” on page 5-20), the files remain encrypted.

Additional Details

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

All the M-files from a given CTF archive are locked together with a unique cryptographic key. M-files with different keys will not execute if placed in the same CTF archive. If you want to generate another application with a different mix of M-files, you must recompile these M-files into a new CTF archive.

The CTF archive and generated binary will be cleaned up following a failed compilation, but only if these files did not exist before compilation was initiated.

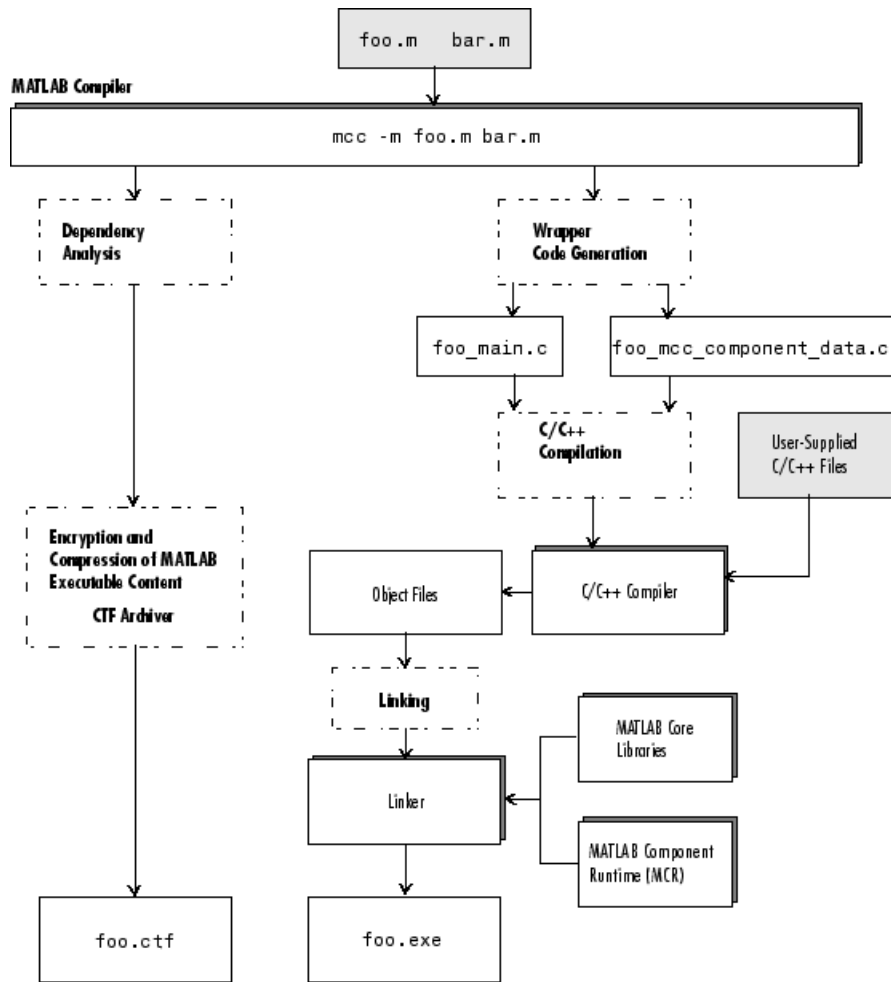
Caution Release Engineers and Software Configuration Managers: CTF archives must not be subjected to build procedures or processes that "strip" shared libraries. If they are, the CTF archive may be stripped from the binary, resulting in run-time errors for the driver applications.

Build Process

The process of creating software components with MATLAB Compiler is completely automatic. For example, to create a standalone MATLAB application, you supply the list of M-files that make up the application. MATLAB Compiler then performs the following operations:

- Dependency analysis
- Code generation
- Archive creation
- Compilation
- Linking

This figure illustrates how MATLAB Compiler takes user code and generates a standalone executable.



Creating a Standalone Executable

Dependency Analysis

The first step determines all the functions on which the supplied M-files, MEX-files, and P-files depend. This list includes all the M-files called by the given files as well as files that they call, and so on. Also included are all built-in functions and MATLAB objects.

Wrapper Code Generation

This step generates all the source code needed to create the target component, including

- The C/C++ interface code to those M-functions supplied on the command line (`foo_main.c`). For libraries and components, this file includes all of the generated interface functions.
- A component data file that contains information needed to execute the M-code at run-time. This data includes path information and encryption keys needed to load the M-code stored in the component's CTF archive.

Archive Creation

The list of MATLAB files (M-files and MEX-files) created during dependency analysis is used to create a CTF archive that contains the files needed by the component to properly execute at run-time. The files are encrypted and compressed into a single file for deployment. Directory information is also included so that the content is properly installed on the target machine.

C/C++ Compilation

This step compiles the generated C/C++ files from wrapper code generation into object code. For targets that support the inclusion of user-supplied C/C++ code on the `mcc` command line, this code is also compiled at this stage.

Linking

The final step links the generated object files with the necessary MATLAB libraries to create the finished component.

The C/C++ compilation and linking steps use the `mbuild` utility that is included with MATLAB Compiler.

Input and Output Files

In this section...
“Standalone Executable” on page 3-6
“C Shared Library” on page 3-7
“C++ Shared Library” on page 3-9
“Macintosh 64 (Maci64)” on page 3-11

Standalone Executable

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

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

File	Description
<code>foo_main.c</code>	The main-wrapper C source file containing the program’s main function. The main function takes the input arguments that are passed on the command line and passes them as strings to the <code>foo</code> function.
<code>foo_mcc_component_data.c</code>	C source file containing data needed by the MCR to run the application. This data includes path information, encryption keys, and other initialization information for the MCR.
<code>foo</code>	The main file of the application. This file reads and executes the content stored in the embedded CTF archive. On Windows, this file is <code>foo.exe</code> .
<code>run_component.sh</code>	<code>mcc</code> generates <code>run_<component>.sh</code> 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, <code>mcc</code> 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 <code>.exe</code> file.

C Shared Library

In this example, MATLAB Compiler takes the M-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 M-functions (<code>foo.m</code> and <code>bar.m</code>) 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 <code>libfoo</code> .
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 <code>mbuild</code> to link the library.
libfoo	<p>The shared library binary file. On Windows, this file is <code>libfoo.dll</code>. On Solaris, this file is <code>libfoo.so</code>.</p> <hr/> <p>Note UNIX extensions vary depending on the platform. See the External Interfaces documentation for additional information.</p> <hr/>

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 M-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
<code>libfoo.cpp</code>	The library wrapper C++ source file containing the exported functions of the library representing the C++ interface to the two M-functions (<code>foo.m</code> and <code>bar.m</code>) as well as library initialization code.
<code>libfoo.h</code>	The library wrapper header file. This file is included by applications that call the exported functions of <code>libfoo</code> .
<code>libfoo_mcc_component_data.c</code>	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.
<code>libfoo.exports</code>	The exports file used by <code>mbuild</code> to link the library.
<code>libfoo</code>	The shared library binary file. On Windows, this file is <code>libfoo.dll</code> . On Solaris, this file is <code>libfoo.so</code> . 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.

Working with MATLAB Data Files Using Load and Save

If your deployed application uses MATLAB data files (MAT files) to store MATLAB variables, graphics, and other data, it is helpful to code LOAD and SAVE functions if you need to manipulate this 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 full path name or relative to `ctfroot`.
- All MAT files are unchanged after `mcc` runs. There is no encryption on these user included data files included in the CTF archive.

For more information about CTF archives, see “Component Technology File” on page 3-2.

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

Example: Using Load/Save Functions To Process MATLAB Data for Deployed Applications

In the following example, three MATLAB data files are specified:

- `user_data.mat`
- `userdata/extra_data.mat`
- `../externdata/extern_data.mat`

Compile `ex_loadsave.m` with the following `mcc` command:

```
mcc -mvC 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 -mC ex_loadsave.m -a 'user_data.mat' -a
%     './userdata/extra_data.mat'
%     -a '../externdata/extern_data.mat'
% All the folders under the current main m-file directory will
% be included as
% relative path to ctroot; 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 m-file path,
% the absolute path will be
% included in ctf and extracted under ctroot. For example:
%   Data file
%     "c:\$matlabroot\examples\externdata\extern_data.mat"
%   will be added into ctf and extracted to
%   "$ctroot\$matlabroot\examples\externdata\extern_data.mat".
%
% All mat/data files are unchanged after mcc runs. There is
% no exryption 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 ctroot 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');
```


Deployment Process

This chapter tells you how to deploy compiled M-code to programmers and to end users.

- “Overview” on page 4-2
- “Deploying to Programmers” on page 4-3
- “Deploying to End Users” on page 4-9
- “Working with the MCR” on page 4-21
- “Deploying a Standalone Application on a Network Drive” on page 4-35
- “MATLAB® Compiler Deployment Messages” on page 4-36
- “Using MATLAB® Compiler Generated DLLs in Windows Services” on page 4-37
- “Reserving Memory for Deployed Applications with MATLAB Memory Shielding” on page 4-38

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 programmers who want to use the library or component to develop an application, or end users who want to run a standalone application.

- “Deploying to Programmers” on page 4-3
- “Deploying to End Users” on page 4-9

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

Deploying to Programmers

In this section...
“Steps by the Programmer to Deploy to Programmers” on page 4-3
“What Software Does a Programmer Need?” on page 4-4
“Ensuring Memory for Deployed Applications” on page 4-8

Steps by the Programmer to Deploy to Programmers

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 M-code. It is frequently helpful to install the MCR on development machines, for testing purposes. See “What Software Does a Programmer Need?” on page 4-4

Note You can use the Deployment Tool to create a package for programmers. 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 Chapter 1, “Getting Started” 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 programmers can just run the self-extracting executable created by the Deployment Tool. UNIX programmers must unzip and install manually.
 - b All programmers must set path environment variables properly. See “Directories Required for Development and Testing” on page 10-2.
- 3 Distribute the package and instructions.

What Software Does a Programmer Need?

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

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

Note MCRInstaller.exe has obsoleted the need for the function `buildmcr` or the creation of `MCRInstaller.zip`. See “Replacement of MCRInstaller.zip and BUILDMCR Functionality” on page 1-19 for more details including complete file paths to all install programs.

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
MCRInstaller.exe (Windows)	MCRInstaller is a self-extracting executable that installs the necessary components to develop your application. This file is included with MATLAB Compiler.
MCRInstaller.bin (UNIX)	MCRInstaller 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.

Software Module	Description
MCRInstaller.dmg (Mac)	MCRInstaller.dmg is a self-extracting executable that installs the necessary components to develop your application on Mac machines. This file is included with MATLAB Compiler.
<i>application_name.exe</i> (Windows)	Application created by MATLAB Compiler. Maci64 must include the bundle directory hierarchy.
<i>application_name</i> (UNIX)	
<i>application_name.app</i> (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
MCRInstaller.bin (UNIX)	MATLAB Compiler Runtime library archive; platform-dependent file that must correspond to the end user’s platform
MCRInstaller.dmg (Mac)	MCRInstaller.dmg is a self-extracting executable that installs the necessary components to develop your application on Mac machines. This file is included with MATLAB Compiler.
MCRInstaller.exe (Windows)	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user’s platform
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; file is 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
<i>componentName.xml</i>	Documentation files
<i>componentName.pdb</i> (if Debug option is selected)	Program Database File, which contains debugging information
<i>componentName.dll</i>	Component assembly file
MCRInstaller.exe	MCR Installer (if not already installed on the target machine)

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.

Software Module	Description
<i>componentname_</i> <i>version.dll</i>	Component that contains compiled M-code.
MCRInstaller.exe	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform. MCRInstaller.exe installs MATLAB Compiler Runtime (MCR), which users of your component need to install on the target machine once per release.

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 M-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
<i>componentname_projectversion.dll</i>	Compiled component.

Software Module	Description
MCRInstaller.exe	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform. MCRInstaller.exe installs the MATLAB Compiler Runtime (MCR), which users of your component need to install on the target machine once per release.
*.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 4-38 for more information.

Deploying to End Users

In this section...

“Steps by the Programmer to Deploy to End Users” on page 4-9

“What Software Does the End User Need?” on page 4-13

“Using Relative Paths with Project Files” on page 4-16

“Porting Generated Code to a Different Platform” on page 4-16

“Extracting a CTF Archive Without Executing the Component” on page 4-17

“Dependency Analysis Function (depfun) and User Interaction with the Compilation Path” on page 4-17

“Ensuring Memory for Deployed Applications” on page 4-20

Steps by the Programmer to Deploy to End Users

Note MCRInstaller.exe has obsoleted the need for the function buildmcr or the creation of MCRInstaller.zip. See “Replacement of MCRInstaller.zip and BUILDMCR Functionality” on page 1-19 for more details including complete file paths to all install programs.

For an end user to run an application or use a library that contains compiled M-code, there are two sets of tasks. Some tasks are for the programmer 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 Programmer Need?” on page 4-4 for more details.

Note The package needed for end users must include the `.ctf` file, which includes all the files in your preferences folder. Thus, you should make sure that you do not have files in your preferences folder that you do not want to expose to end users. MATLAB preferences set at compile time are inherited by a compiled application. 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.

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.`

Caution 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.

Note The package needed for end users must include the `.ctf` file, which includes all the files in your preferences folder. Thus, you should make sure that you do not have files in your preferences folder that you do not want to expose to end users. MATLAB preferences set at compile time are inherited by a compiled application. 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.

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.`

Caution 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.

- 2 Write instructions for the end user. See “Steps by the End User” on page 4-11.
- 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 4-12.

- 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 MCRInstaller 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 4-13
- “.NET Application” on page 4-14
- “COM Application” on page 4-14
- “Java Application” on page 4-15
- “Microsoft® Excel Add-In” on page 4-15

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
MCRInstaller.exe (Windows)	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user’s platform.

Component	Description
matrixdriver.exe (Windows)	Application
matrixdriver (UNIX)	
libmatrix	Shared library; extension varies by platform. Extensions are: <ul style="list-style-type: none"> • Windows — .dll • Solaris, 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
<i>componentName.xml</i>	Documentation files
<i>componentName.pdb</i> (if Debug option is selected)	Program Database File, which contains debugging information
<i>componentName.dll</i>	Component assembly file
MCRInstaller.exe	MCR Installer (if not already installed on the target machine)
<i>application.exe</i>	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
<i>componentname.ctf</i>	Component Technology File (ctf) archive. This is a platform-dependent file that must correspond to the end user's platform.
<i>componentname_version.dll</i>	Component that contains compiled M-code
<i>_install.bat</i>	Script run by the self-extracting executable
<i>MCRInstaller.exe</i>	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform. <i>MCRInstaller.exe</i> installs MATLAB Compiler Runtime (MCR), which users of your component need to install on the target machine once per release.
<i>application.exe</i>	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
<i>componentname_version.dll</i>	Component that contains compiled M-code
<i>_install.bat</i>	Script run by the self-extracting executable

Software Module	Description
MCRInstaller.exe	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform
*.xla	Any Excel add-in files found in <i>projectdirectory\distrib</i>

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 (M-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 “Overriding Default CTF Archive Embedding Using the MCR Component Cache” on page 5-20). To do this, ensure that you compile your component with the “-C Do Not Embed CTF Archive by Default” on page 12-30 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/deploy/arch` folder, where *arch* is your system architecture, Windows = win32|win64, Linux = glnx86, Solaris = sol64, 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/deploy/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.

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

MATLAB Compiler uses a dependency analysis function (`depfun`) to determine the list of necessary files to include in the CTF package. In some

cases, this process includes an excessive number of files, for example, when MATLAB object classes are included in the compilation and it cannot resolve overloaded methods at compile time. The dependency analysis is an iterative process that also processes `include/exclude` information on each pass. Consequently, this process can lead to very large CTF archives resulting in long compilation times for relatively small applications.

`depfun` searches for “executable” content such as:

- M-files
- P-files
- Java classes and `.jar` files
- `.fig` files
- MEX-files and dependent binaries

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

M-files are encrypted by `depfun`, and `depfun` creates authorization files for each MEX-file. This file guards against MEX-files of the same name (but of different functionality) being placed into exploded CTF folders.

The most effective way to reduce the number of files is to constrain the MATLAB path that `depfun` uses at compile time. MATLAB Compiler includes features that enable you to manipulate the path. Currently, there are three ways to interact with the compilation path:

- `addpath` and `rmpath` in MATLAB
- Passing `-I <directory>` on the `mcc` command line
- Passing `-N` and `-p` folders on the `mcc` command line

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.

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 4-38 for more information.

Working with the MCR

In this section...

“Understanding the MCR” on page 4-21

“Installing the MCR and MATLAB on the Same Machine” on page 4-22

“Installing Multiple MCRs on One Machine” on page 4-24

“Retrieving MCR Attributes” on page 4-24

“Improving Data Access Using the MCR User Data Interface” on page 4-26

“Displaying MCR Initialization Start-Up and Completion Messages For Users” on page 4-33

Understanding 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.

If you do not have MATLAB installed on the target machine and you want to run components created by MATLAB Compiler, you still need to install the MCR on the target machine, whether you are a developer or end user. You have to install the MCR only once. There is no way to distribute your application with any subset of the files that are installed by the MCRInstaller.

You can install the MCR by running `MCRInstaller.exe`.

On platforms other than Windows, you must also set paths and environment variables. See “Directories Required for Run-Time Deployment” on page 10-5 for more information about these settings.

Note 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.

Note If you are deploying .NET component applications to programmers or end users, make sure to tell them to install .NET Framework before installing the MCR. The `MCRinstaller.exe` must detect the presence of .NET Framework on a system for it to install MCR .NET support. Alternatively, you can package .NET Framework with the component installer that you provide to them as part of your deployment package.

Note `MCRInstaller.exe` has obsoleted the need for the function `buildmcr` or the creation of `MCRInstaller.zip`. See “Replacement of `MCRInstaller.zip` and `BUILDMCR` Functionality” on page 1-19 for more details including complete file paths to all install programs.

See “Deploying to End Users” on page 4-9 for more information about the general steps for installing the MCR as part of the deployment process.

See also “Using MCR Installer Command Line Options” on page 10-9 for more information.

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.

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 configurations 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 Solaris, the `<mcr_root>/runtime/<arch>` folder must appear on your `LD_LIBRARY_PATH` before `matlabroot/runtime/<arch>`, and `XAPPLRESDIR` should point to `<mcr_root>/X11/app-defaults`. See “Directories Required for Run-Time Deployment” on page 10-5 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`, and `XAPPLRESDIR` should point to `<mcr_root>/X11/app-defaults`.

To run MATLAB on Mac OS X or Intel® Mac, `matlabroot/runtime` must appear on your `DYLD_LIBRARY_PATH` before the `<mcr_root>/bin` folder, and `XAPPLRESDIR` should point to `matlabroot/X11/app-defaults`.

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 4-24 for more information.

Retrieving MCR Attributes

Use these new 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
<pre>bool mclIsMCRInitialized()</pre>	Use <code>mclIsMCRInitialized()</code> 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.
<pre>bool mclIsJVMEEnabled()</pre>	Use <code>mclIsJVMEEnabled()</code> 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.
<pre>const char* mclGetLogFileName()</pre>	Use <code>mclGetLogFileName()</code> to retrieve the name of the log file used by the MCR	Character string representing log file name used by MCR
<pre>bool mclIsNoDisplaySet()</pre>	Use <code>mclIsNoDisplaySet()</code> to determine if <code>-nodisplay</code> option is enabled.	Boolean (true or false). Returns true if <code>-nodisplay</code> is enabled, else returns false. <hr/> <p>Note false is always returned on Windows systems since the <code>-nodisplay</code> option is not supported on Windows systems.</p> <hr/> <p>Caution When running on Mac, if <code>-nodisplay</code> is used as one of the options included in <code>mclInitializeApplication</code>, then the call to <code>mclInitializeApplication</code> must occur before calling <code>mclRunMain</code>.</p> <hr/>

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

Example: Retrieving Information from MCR State

```
const char* options[4];
options[0] = "-logfile";
options[1] = "logfile.txt";
options[2] = "-nojvm";
options[3] = "-nodisplay";
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", mclIsJVMEEnabled());
printf("Logfile name : %s\n", mclGetLogFileName());
printf("nodisplay set : %d\n", mclIsNoDisplaySet());
fflush(stdout);
```

Improving Data Access 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 M-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 configuration information to a client running an application created with the Parallel Computing Toolbox. You supply and change configuration information on a per-execution basis. For example, two instances of the same application may run simultaneously with different configuration files. See “Deploying Applications Created Using Parallel Computing Toolbox” on page 4-33 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 M-functions callable from within deployed application M-code
- Four external C functions callable from within deployed application wrapper code

Note The M-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 User's Guide*, *MATLAB Builder JA User's Guide*, and *MATLAB Builder EX User's Guide*, respectively.

MATLAB Functions

Use the M-language functions `getmcuserdata` and `setmcuserdata` from deployed M applications. They are loaded by default only in applications created with the MATLAB Compiler or builder products. See Chapter 12, “Functions — Alphabetical List”, for more information.

Caution 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 `getmcuserdata` and `setmcuserdata`. For more information about the `isdeployed` function, see the `isdeployed` reference page.

External C Functions

Use the following C functions in deployed C/C++ applications. See Chapter 12, “Functions — Alphabetical List” for more information.

- `libnameGetMcrID`
- `mclSetMCRUserData`
- `mclGetMCRUserData`
- `mclSetCmdLineUserData`

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 configuration `.mat` file:

```
parallelapp.exe -mcruserdata  
                ParallelConfigurationFile: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` `ParallelConfigurationFile` *before* calling any Parallel Computing Toolbox™ code. Once this code has been called, setting `ParallelConfigurationFile` to point to a different file has no effect.

Setting and Retrieving MCR Data for Shared Libraries

As mentioned in “Improving Data Access Using the MCR User Data Interface” on page 4-26, 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 Outside the scope of your main code, use `libnameGetMcrID` to retrieve the key value of the MCR data you want to update.
- 2 In your code, Include the MCR header file and the library header generated by MATLAB Compiler.
- 3 Properly initialize your application using `mclInitializeApplication`.
- 4 After creating your input data, write or “set” it to the MCR with `setmcruserdata` or `mclSetMCRUserData`, as appropriate. Use `mclSetCmdLineUserData` to set data from the command line.
- 5 After calling functions or performing other processing, retrieve the new MCR data with `getmcruserdata` or `mclGetMCRUserData`, as appropriate.
- 6 Free up storage memory in work areas by disposing of unneeded arrays with `mxDestroyArray`.
- 7 Shut down your application properly with `mclTerminateApplication`.

Example: MagicMatrix. This following is an end-to-end example showing how to set and retrieve MCR data with the `magicmatrix` application and the MCR User Data interface API:

Building on UNIX

```
mbuild -gv -output magicmatrix magicmatrix.c libmagicmatrix.so
```

Building on Windows

```
mbuild -gv -output magicmatrix magicmatrix.c libmagicmatrix.lib
```

Running on UNIX

```
% ./magicmatrix
```

Running on Windows

```
C:\> magicmatrix.exe
```

Running on 64-bit Macintosh

```
./magicmatrix.app/Contents/MacOS/magicmatrix
```

magicmatrix.c

```
#include <stdio.h>

/* Include the MCR header file and the library specific header
 * file as generated by MATLAB Compiler */
#include "libmagicmatrix.h"

/* This function is used to display a double matrix stored in
 * an mxArray */
void display(const mxArray* in);

int run_main(int argc, char **argv)
{
    /* Input and output parameters. For testing, keep inputValue
     * and outputValue in separate arrays, so a dangling
     * pointer can't
     * cause a false positive.
     */
    mxArray *key, *inputValue;
    mxArray *outputValue = NULL;

    /* Column-major 3x3 magic square. */
    double data[] = {8, 3, 4, 1, 5, 9, 6, 7, 2};

    /* Initialize the application */
    if( !mclInitializeApplication(NULL,0) )
    {
        fprintf(stderr,
            "Could not initialize the application.\n");
        return -1;
    }

    /* Call the library initialization routine and make sure
     * that the
     * library was initialized properly. */
    if (!libmagicmatrixInitialize()){
```



```
        fprintf(stderr, "Could not initialize the library.\n");
        return -2;
    }
    else
    {
        /* Declare an array to hold the inputs */
        mxArray *key = 0, *inputValue = 0;

        /* Create the input data */
        inputValue = mxCreateDoubleMatrix(3,3,mxREAL); /* key */
        key = mxCreateString("MagicMatrix");          /* value */
        memcpy(mxGetPr(inputValue), data, 9*sizeof(double));

        /* Set the user data: 2 inputs, no outputs */
        mlfSetmcruserdata(key, inputValue);

        /* Call the library function */
        mlfMagicmatrix(0, NULL, NULL);

        /* Get the MCR user data - it should be different now */
        mlfGetmcruserdata(1, &outputValue, key);

        display(outputValue);

        /* Clean up */
        mxDestroyArray(outputValue); outputValue=0;

        /* Call the library termination routine */
        libmagicmatrixTerminate();

        /* Free the memory created for the inputs */
        mxDestroyArray(key); key = 0;
        mxDestroyArray(inputValue); inputValue = 0;
    }

    /* Shut everything down */
    mclTerminateApplication();
    return 0;
}
```

```
/*DISPLAY This function will display the double matrix stored
 * in an mxArray.
 * This function assumes that the mxArray passed as
 * input contains double array.
 */
void display(const mxArray* in)
{
    int i=0, j=0; /* loop index variables */
    int r=0, c=0; /* variables to store the row and column
 * length of the matrix */
    double *data; /* variable to point to the double data
 * stored within the mxArray */

    /* Get the size of the matrix */
    r = mxGetM(in);
    c = mxGetN(in);
    /* Get a pointer to the double data in mxArray */
    data = mxGetPr(in);

    /* Loop through the data and display the same in
 * matrix format */
    for( i = 0; i < c; i++ ){
        for( j = 0; j < r; j++){
            printf("%4.2f\t",data[j*c+i]);
        }
        printf("\n");
    }
    printf("\n");
}

int main()
{
    mclmcrInitialize();
    return mclRunMain((mclMainFcnType)run_main,0,NULL);
}
```

Deploying Applications Created Using Parallel Computing Toolbox

For information about creating and exporting configurations from Parallel Computing Toolbox applications, see “Programming with User Configurations”.

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 guides.

For a complete working example describing how to use Parallel Computing Toolbox with MATLAB Builder JA, see “Example: Supplying Run-Time Configuration Information for Parallel Computing Toolbox Applications”.

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:
<code>mcc -R -startmsg</code>	Default start-up message Initializing MATLAB Compiler Runtime version <code>x.xx</code>

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

Deploying a Standalone Application on a Network Drive

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.

- 1** On any Windows machine, execute `MCRInstaller.exe` to install the MATLAB Compiler Runtime (MCR).
- 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 “Enabling MATLAB Compiler Deployment Messages” in *MATLAB Desktop Tools and Development Environment*.

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 4-38
“Requirements for Using MATLAB Memory Shielding” on page 4-39
“Invoking MATLAB Memory Shielding for Your Deployed Application” on page 4-39

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 run-time memory requests, if possible. See “What Is MATLAB Memory Shielding and When Should You Use It?” on page 4-38 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 Component 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_application_path [user-defined_application_arguments]
```

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

```
MemShieldStarter  
fully-qualified_application_path [user-defined_application_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 4-41 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_application_path [user-defined_application_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 4-42 for more details.
-shield <i>shield_level</i>	See “Shield Level Options” on page 4-41.
<i>fully-qualified_application_path</i>	The fully qualified path to your user application
<i>user-defined_application_arguments</i>	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.

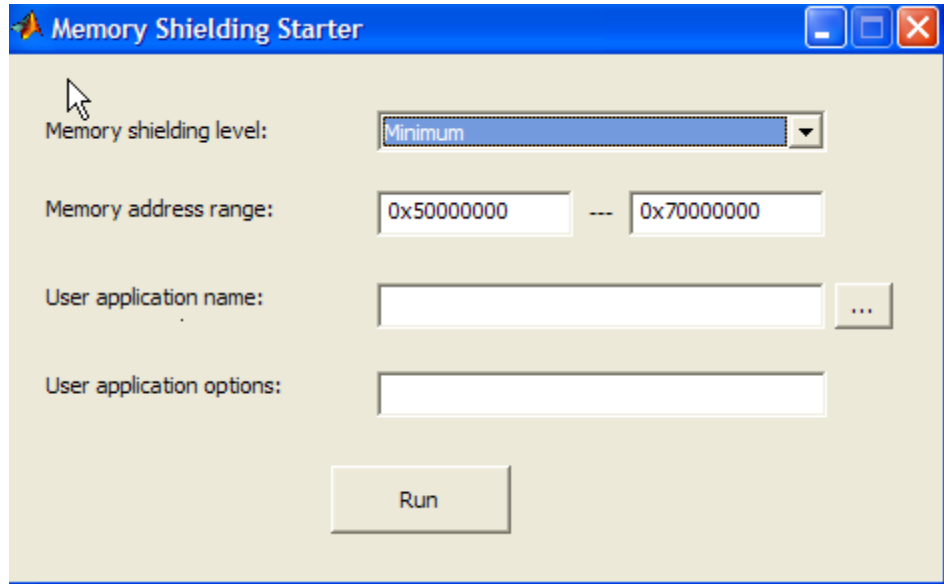
Using the GUI

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 4-41. 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 4-41 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” on page 5-2
- “Using Macros to Simplify Compilation” on page 5-5
- “Using Path Names” on page 5-8
- “Using Bundle Files” on page 5-9
- “Using Wrapper Files” on page 5-11
- “Interfacing M-Code to C/C++ Code” on page 5-14
- “Overriding Default CTF Archive Embedding Using the MCR Component Cache” on page 5-20
- “Using Pragmas” on page 5-22
- “Using `mxArray`” on page 5-24
- “Script Files” on page 5-25
- “Compiler Tips” on page 5-27

Command Overview

In this section...
“Compiler Options” on page 5-2
“Combining Options” on page 5-2
“Conflicting Options on the Command Line” on page 5-3
“Using File Extensions” on page 5-3

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 -g 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 “Using Macros to Simplify Compilation” on page 5-5.

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 -mg 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 5-5,

```
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 M-files, therefore if both P-files and M-files reside in a folder, and a file name is specified without an extension, the P-file will be selected.

Using Macros to Simplify Compilation

In this section...

“Macro Options” on page 5-5

“Working With Macro Options” on page 5-5

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 C application	-W main	-T link:exe

Working With Macro Options

The `-m` option tells MATLAB Compiler to produce a standalone C 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
```

Using Path Names

If you specify a full path name to an M-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>`”. 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 M-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 M-file MATLAB Compiler parses. The `-v` option prints the full path name to the M-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

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
```

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.

Using Wrapper Files

In this section...
“What Are Wrapper Files?” on page 5-11
“Main File Wrapper” on page 5-11
“C Library Wrapper” on page 5-12
“C++ Library Wrapper” on page 5-13

What Are Wrapper Files?

Wrapper files encapsulate, or wrap, the M-files in your application with an interface that enables the M-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.

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 M-file function(s) as MATLAB strings. They are meant to translate “command-like” M-files into POSIX main applications.

POSIX Main Wrapper

Consider this M-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

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 M-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 M-files. The generated header file contains a C function declaration for each

of the compiled M-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 M-files into a library. The generated header file contains all of the entry points for all of the compiled M-functions.

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

Interfacing M-Code to C/C++ Code

In this section...

“Overview” on page 5-14

“Code Proper Return Types From C and C++ Methods” on page 5-14

“C Example” on page 5-14

“C++ Example” on page 5-16

Overview

MATLAB Compiler supports calling arbitrary C/C++ functions from your M-code. To use this feature, provide an M-function stub that determines how the code will behave in M, and then provide an implementation of the body of the function in C or C++.

Code Proper Return Types From C and C++ Methods

When coding, keep in mind that LCC compilers can be more strict in enforcing `bool` return types from C and `void` returns from C++ than Microsoft compilers. To avoid potential problems, ensure all C methods you write (and reference from within M) return a `bool` return type indicating the status, and any C++ methods return `void`.

C Example

Suppose you have a C function that reads data from a measurement device. In M-code, you want to simulate the device by providing a sine wave output, so you provide a function that returns the measurement obtained from the device. This C function, `measure_from_device()`, returns a `double`, which is the current measurement.

To replace the implementation of the `collect_one` function with a C implementation, use the `extern` pragma:

- 1 Compile the MATLAB code with the `extern` pragma once to generate the header file `function_name_external.h`, where `function_name` is the name of the initial M-function containing the `extern` pragma. This

header file will contain the `extern` declaration of the function that you must provide. This function must conform to the same interface as code generated by MATLAB Compiler.

MATLAB Compiler will generate the interface for any functions that contain the `extern` pragma into a separate file called `function_name_external.h`. The C or C++ file generated by MATLAB Compiler will include this header file to get the declaration of the function being provided.

Place the pragma in the `collect.m` local function as follows:

```
function collect

    y = zeros(1,100); % pre-allocate the matrix
    for i = 1:100
        y(i) = collect_one;
    end

function y = collect_one
    %EXTERNAL
    persistent t;

    if (isempty(t))
        t = 0;
    else
        t = t+0.05;
    end
    y = sin(t);
```

2 Compile the `collect` function.

When this file is compiled, MATLAB Compiler creates the additional header file `collect_one_external.h`, which contains the interface between MATLAB Compiler generated code and your code. In this example, it would contain:

```
extern bool mlxCollect_one(int nlhs, mxArray *plhs[],
                          int nrhs, mxArray *prhs[]);
```

- 3** It is recommended that you include this header file when defining the function. This function can be implemented in this C file, `measure.c`, using the `measure_from_device()` function.

```
#include "collect_one_external.h"
#include <math.h>

extern double measure_from_device(void);
bool mlxCollect_one(int nlhs, mxArray *plhs[],
                   int nrhs, mxArray *prhs[])
{
    plhs[0] = mxCreateDoubleMatrix(1,1,mxREAL);
    *(mxGetPr(plhs[0])) = measure_from_device();
}

double measure_from_device(void)
{
    static double t = 0.0;
    t = t + 0.05;
    return sin(t);
}
```

To generate the application, use

```
mcc -m collect.m measure.c
```

C++ Example

Suppose you have a C function that reads data from a measurement device. In M-code, you want to simulate the device by providing a sine wave output, so you provide a function that returns the measurement obtained from the device. For example, this C function, `measure_from_device()`, returns a double, which is the current measurement.

`collect.m` contains the M-code for the simulation of your application.

```
function collect

    y = zeros(1,100); % pre-allocate the matrix
```

```

for i = 1:100
    y(i) = collect_one;
end

```

```

function y = collect_one
    %#EXTERNAL
    persistent t;

    if (isempty(t))
        t = 0;
    else
        t = t+0.05;
    end
    y = sin(t);

```

To replace the implementation of the `collect_one` function with a C implementation, use the `%#external` pragma.

- 1 Compile the MATLAB code with the `%#external` pragma once to generate the header file `function_name_external.h`, where `function_name` is the name of the initial M-function containing the `%#external` pragma. This header file will contain the `extern` declaration of the function that you must provide. This function must conform to the same interface as code generated by MATLAB Compiler.

MATLAB Compiler will generate the interface for any functions that contain the `%#external` pragma into a separate file called `function_name_external.h`. The C or C++ file generated by MATLAB Compiler will include this header file to get the declaration of the function being provided.

In this example, place the pragma in the `collect_one` local function.

```

function collect

y = zeros(1, 100); % preallocate the matrix
for i = 1:100
    y(i) = collect_one;

```

```
end
disp (y)

function y = collect_one

%#external
persistent t;
if (isempty(t))
    t = 0;
end
t = t + 0.05;
y = sin(t);
```

2 Compile the collect function.

When this file is compiled, MATLAB Compiler creates the additional header file `collect_one_external.h`, which contains the interface between MATLAB Compiler generated code and your code. In this example, it would contain:

```
extern void mlxCollect_one(int nlhs, mxArray *plhs[],
                          int nrhs, mxArray *prhs[]);
```

3 It is recommended that you include this header file when defining the function. This function can be implemented in this C++ file, `measure.cpp`, using the `measure_from_device()` function.

```
#include "collect_one_external.h"
#include <math.h>

extern double measure_from_device(void);

void mlxCollect_one(int nlhs, mxArray *plhs[],
                   int nrhs, mxArray *prhs[])

{
    plhs[0] = mxCreateDoubleMatrix(1,1,mxREAL);
    *(mxGetPr(plhs[0])) = measure_from_device();
}
```



```
double measure_from_device(void)
{
    static double t = 0.0;
    t = t + 0.05;
    return sin(t);
}
```

To generate the application, use

```
mcc -m collect.m measure.cpp
```

Overriding Default CTF Archive Embedding Using the MCR Component Cache

As of R2008a, CTF data is automatically embedded directly in the C/C++, main and Winmain, shared libraries and standalones by default. In order to override this default functionality, you must compile using the option “-C Do Not Embed CTF Archive by Default” on page 12-30.

If you do not use the `mcc -C` option to specify that a separate CTF file be generated, you can add environment variables to specify various options, such as:

- Defining the location where you want the CTF archive to be extracted
- Adding diagnostic error printing options that can be used when 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

Environment Variable	Purpose	Notes
MCR_CACHE_VERBOSE	When set, this variable prints details about the component cache for diagnostic reasons. This can be very helpful if problems are encountered during CTF archive extraction.	Does not apply
MCR_CACHE_SIZE	When set, this variable overrides the default component cache size.	The initial limit for this variable is 32M (megabytes). This may, however, be changed after you have set the variable the first time. Edit the file <code>.max_size</code> , which resides in the file designated by running the <code>mcrcachedir</code> command, with the desired cache size limit.

Note 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.

Using Pragmas

In this section...

“Using feval” on page 5-22

“Example: Using %#function” on page 5-22

Using feval

In standalone C and C++ modes, 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 M-files used in your application. This pragma adds the top-level function as well as all the subfunctions in the file to the compilation.

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

Example: 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;
```

Using mxArray

For full documentation on the mxArray API, see “MX Array Manipulation” in *MATLAB C and Fortran API Reference*.

For a complete description of data types used with mxArray, see “MATLAB Data” in *MATLAB External Interfaces*.

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

Script Files

In this section...
“Converting Script M-Files to Function M-Files” on page 5-25
“Including Script Files in Deployed Applications” on page 5-26

Converting Script M-Files to Function M-Files

MATLAB provides two ways to package sequences of MATLAB commands:

- Function M-files
- Script M-files

Some things to remember about script and function M-files:

- Variables used inside function M-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 M-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 M-files or can compile function M-files that call scripts. You can either specify an script M-file explicitly on the `mcc` command line, or you can specify function M-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 M-file.

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

If desired, convert this script M-file into a function M-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 M-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 M-file. In libraries, the interface layer consists of the M-files specified on the `mcc` command line.

The second layer of M-files in compiled applications includes those M-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 M-file by writing a new M-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 M-file, use

```
mcc -m houdini_fcn
```


Compiler Tips

In this section...

“Calling Built-In Functions from C or C++” on page 5-27

“Calling a Function from the Command Line” on page 5-28

“Using MAT-Files in Deployed Applications” on page 5-28

“Compiling a GUI That Contains an ActiveX Control” on page 5-28

“Debugging MATLAB® Compiler Generated Executables” on page 5-29

“Deploying Applications That Call the Java Native Libraries” on page 5-29

“Locating .fig Files in Deployed Applications” on page 5-29

“Blocking Execution of a Console Application That Creates Figures and Terminating Figures by Force” on page 5-30

“Passing Arguments to and from a Standalone Application” on page 5-31

“Using Graphical Applications in Shared Library Targets” on page 5-33

“Using the VER Function in a Compiled MATLAB Application” on page 5-33

Calling Built-In Functions from C or C++

To enable a C or C++ program to call a built-in function directly, you must write an M-file wrapper around each built-in function you want to access outside of MATLAB. This is necessary because there are no C callable interfaces to built-in functions. For example, to use the `magic` function in a deployed application, you can use this M-file:

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

% Copyright 2003 The MathWorks, Inc.

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

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 (isstr(a)), a = str2num(a), end;
    if (isstr(b)), b = str2num(b), end;

    % The rest of your M-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 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 12-27.

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 M-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 an M-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.

Blocking Execution of a Console Application That Creates Figures and Terminating Figures by Force

- “Blocking Execution of a Console Application with the `mclWaitForFiguresToDie` Method” on page 5-30
- “Terminating Figures by Force with the `mclKillAllFigures` Method” on page 5-31

Blocking Execution of a Console Application with the `mclWaitForFiguresToDie` Method

The purpose of `mclWaitForFiguresToDie` is to block execution of a calling program as long as figures created in encapsulated M-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.

Terminating Figures by Force with the `mclKillAllFigures` Method

`mclKillAllFigures` finds all open figures and deletes them. This function uses the same internal algorithm to locate open figures as `mclWaitForFiguresToDie`. The published signature is:

```
void mclKillAllFigures(HMCRINSTANCE inst)
```

Typically you use `mclKillAllFigures` when:

- You need to kill figures that are being displayed as the result of a programming problem, such as an endless loop.
- You want to ensure all figures are closed before the execution of another application.

Example: Terminating and Deleting Open Figures Using

`mclKillAllFigures`. In this example, `mclKillAllFigures` is used to terminate and delete three figures displayed by the shared library calls `showknot()`, `showpeak()`, and `showbottle()` using three methods, all of which have the same result of terminating all figures (knot, peak, and bottle).

```
showknot();  
showpeak();  
showbottle();  
mclKillAllFigures((HMCRINSTANCE)0);
```

```
showknot();  
showpeak();  
showbottle();  
mclKillAllFigures(0);
```

```
showknot();  
showpeak();  
showbottle();  
mclKillAllFigures(NULL);
```

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 M-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 M-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 M-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 `mc1WaitForFiguresToDie` 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” on page 6-2
- “C Standalone Application Target” on page 6-3
- “Coding with M-Files Only” on page 6-8
- “Mixing M-Files and C or C++” on page 6-10

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 M-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 C applications consists either entirely of M-files or some combination of M-files, MEX-files, and C or C++ source code files.

MATLAB Compiler takes your M-files and generates C source code functions that allow your M-files to be invoked from outside of interactive MATLAB. After compiling this C source code, the resulting object file is linked with the run-time libraries. A similar process is used to create C++ standalone applications.

You can call MEX-files from MATLAB Compiler generated standalone applications. The MEX-files will then be loaded and called by the standalone code.

C Standalone Application Target

In this section...

“Compiling the Application” on page 6-3

“Testing the Application” on page 6-3

“Deploying the Application” on page 6-4

“Running the Application” on page 6-6

Compiling the Application

This example takes an M-file, `magicsquare.m`, and creates a standalone C application, `magicsquare`.

- 1 Copy the file `magicsquare.m` from

```
matlabroot/extern/examples/compiler
```

to your work folder.

- 2 To compile the M-code, use

```
mcc -mv magicsquare.m
```

The `-m` option tells MATLAB Compiler (`mcc`) to generate a C 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 3-6 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 “UNIX Settings for Development and Testing” on page 10-3
- 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.

Note MCRInstaller.exe has obsoleted the need for the function buildmcr or the creation of MCRInstaller.zip. See “Replacement of MCRInstaller.zip and BUILDMCR Functionality” on page 1-19 for more details including complete file paths to all install programs.

Windows

Gather and package the following files and distribute them to the deployment machine.

Component	Description
MCRInstaller.exe	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user’s platform.
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
MCRInstaller.bin	MATLAB Compiler Runtime library archive; platform-dependent file that must correspond to the end user’s platform
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
MCRInstaller.bin	MATLAB Compiler Runtime library archive; platform-dependent file that must correspond to the end user's platform
magicsquare	Application
magicsquare.app	<p>Application bundle</p> <p>Assuming foo is a folder within your current folder:</p> <ul style="list-style-type: none"> • Distribute by copying: <pre>cp -R myapp.app foo</pre> • Distribute by tarring: <pre>tar -cvf myapp.tar myapp.app cd foo tar -xvf ../ myapp.tar</pre> • Distribute by zipping: <pre>zip -ry myapp myapp.app cd foo unzip ../myapp.zip</pre>

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

- 1 Install the MCR by running the MCRInstaller in a folder. For example, run MCRInstaller.exe in C:\MCR. For more information on running the MCR Installer utility, see “Working with the MCR” on page 4-21 and “Replacement of MCRInstaller.zip and BUILDMCR Functionality” on page 1-19.

- 2** See “Path Modifications Required for Accessibility” on page 10-2 for information on setting your path.

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 5-31.

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
```

Coding with M-Files Only

In this section...
“M-File Advantages” on page 6-8
“Example” on page 6-8

M-File Advantages

One way to create a standalone application is to write all the source code in one or more M-files or MEX-files as in the previous magic square example. Coding an application with M allows you to take advantage of the MATLAB interactive development environment. Once the M-file version of your program works properly, compile the code and build it into a standalone application.

Example

Consider a simple application whose source code consists of two M-files, `mrank.m` and `main.m`. This example generates C code from your M-files.

mrank.m

`mrank.m` returns a vector of integers, `r`. Each element of `r` represents the rank of a magic square. For example, after the function completes, `r(3)` contains the rank of a 3-by-3 magic square.

```
function r = mrank(n)
r = zeros(n,1);
for k = 1:n
    r(k) = rank(magic(k));
end
```

In this example, the line `r = zeros(n,1)` preallocates memory to help the performance of MATLAB Compiler.

main.m

`main.m` contains a “main routine” that calls `mrank` and then prints the results.


```
function main
r = mrank(5)
```

Compiling the Example

To compile these functions into code that can be built into a standalone application, invoke MATLAB Compiler.

```
mcc -m main mrank
```

The `-m` option causes MATLAB Compiler to generate C source code suitable for standalone applications. For example, MATLAB Compiler generates C source code files `main_main.c` and `main_mcc_component_data.c`. `main_main.c` contains a C function named `main`; `main_mcc_component_data.c` contains data needed by the MCR to run the application.

To build an application, you can use `mbuild` to compile and link these files. Or, you can automate the entire build process (invoke MATLAB Compiler on both M-files, use `mbuild` to compile the files with your ANSI C compiler, and link the code) by using the command

```
mcc -m main mrank
```

If you need to combine other code with your application (Fortran, for example, a language not supported by MATLAB Compiler), or if you want to build a makefile that compiles your application, you can use the command

```
mcc -mc main mrank
```

The `-c` option inhibits invocation of `mbuild`. You will probably need to examine the verbose output of `mbuild` to determine how to set the compiler options in your makefile. Run

```
mcc -mv main mrank
```

to see the switches and options that `mbuild` uses on your platform.

Mixing M-Files and C or C++

In this section...
“Examples Overview” on page 6-10
“Simple Example” on page 6-10
“Advanced C Example” on page 6-15

Examples Overview

The examples in this section illustrate how to mix M-files and C or C++ source code files:

- The first example is a simple application that mixes M-files and C code.
- The second example illustrates how to write C code that calls a compiled M-file.

One way to create a standalone application is to code some of it as one or more function M-files and to code other parts directly in C or C++. To write a standalone application this way, you must know how to do the following:

- Call the external C or C++ functions generated by MATLAB Compiler.
- Handle the results these C or C++ functions return.

Note If you include compiled M-code into a larger application, you must produce a library wrapper file even if you do not actually create a separate library. For more information on creating libraries, see Chapter 7, “Libraries”.

For more information on mxArray, see “Using mxArray” on page 5-24.

Simple Example

This example involves mixing M-files and C code. Consider a simple application whose source code consists of `mrank.m`, `mrankp.c`, `main_for_lib.c`, and `main_for_lib.h`.

mrnk.m

`mrnk.m` contains a function that returns a vector of the ranks of the magic squares from 1 to `n`.

```
function r = mrnk(n)
r = zeros(n,1);
for k = 1:n
    r(k) = rank(magic(k));
end
```

Copy `mrnk.m`, `printmatrix.m`, `mrnkp.c`, `main_for_lib.c`, and `main_for_lib.h` into your current folder.

Build Process

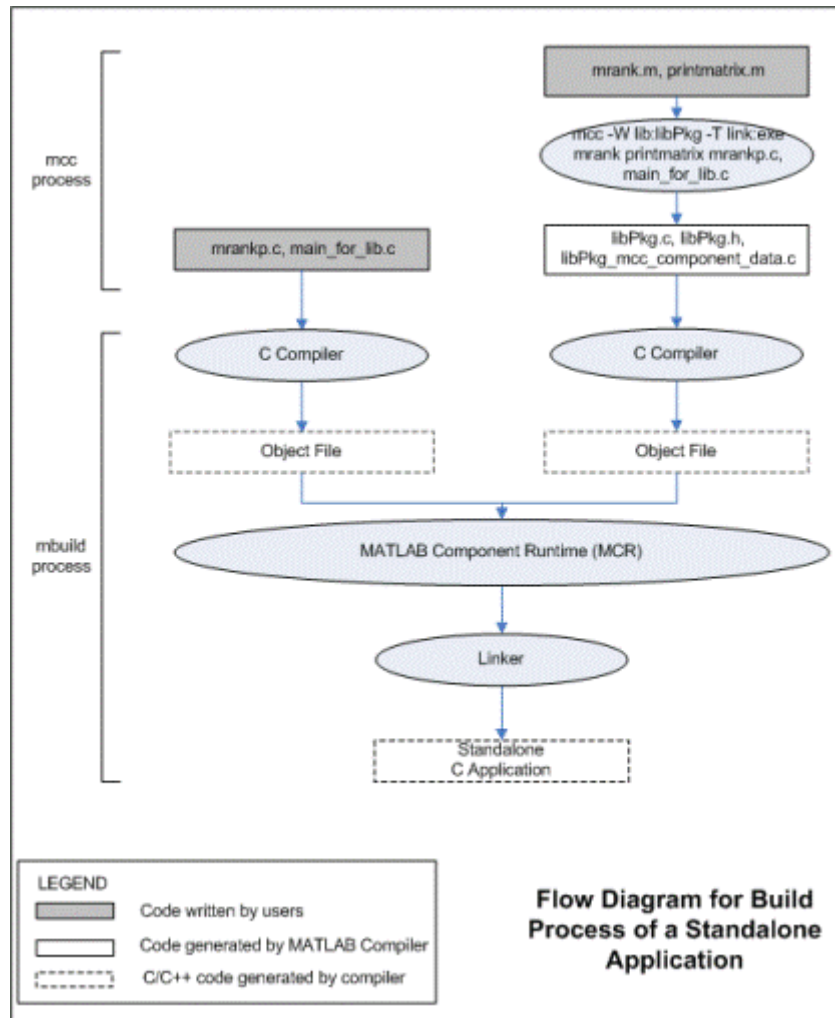
The steps needed to build this standalone application are:

- 1** Compile the M-code.
- 2** Generate the library wrapper file.
- 3** Create the binary.

To perform these steps, enter the following on a single line:

```
mcc -W lib:libPkg -T link:exe mrnk printmatrix mrnkp.c
main_for_lib.c
```

The following flow diagram shows the mixing of M-files and C-files that forms this sample standalone application. The top part of the diagram shows the `mcc` process and the lower part shows the `mbuild` process.



MATLAB Compiler generates the following C source code files:

- `libPkg.c`
- `libPkg.h`
- `libPkg_mcc_component_data.c`

This command invokes `mbuild` to compile the resulting MATLAB Compiler generated source files with the existing C source files (`mrankp.c` and `main_for_lib.c`) and link against the required libraries.

MATLAB Compiler provides two different versions of `mrankp.c` in the `matlabroot/extern/examples/compiler` folder:

- `mrankp.c` contains a POSIX-compliant main function. `mrankp.c` sends its output to the standard output stream and gathers its input from the standard input stream.
- `mrankwin.c` contains a Windows version of `mrankp.c`.

mrankp.c

The code in `mrankp.c` calls `mrank` and outputs the values that `mrank` returns.

```

/*
 * MRANKP.C
 * "Posix" C main program
 * Calls mlfMrank, obtained by using MCC to compile mrank.m.
 *
 * $Revision: 1.1.4.57 $
 *
 */

#include <stdio.h>
#include <math.h>
#include "libPkg.h"

main( int argc, char **argv )
{
    mxArray *N;          /* Matrix containing n. */
    mxArray *R = NULL;  /* Result matrix. */
    int      n;         /* Integer parameter from command line.*/

    /* Get any command line parameter. */
    if (argc >= 2) {
        n = atoi(argv[1]);
    } else {
        n = 12;
    }
}

```

```
    }
    mclInitializeApplication(NULL,0);
    libPkgInitialize(); /* Initialize library of M-Functions */

    /* Create a 1-by-1 matrix containing n. */
    N = mxCreateDoubleScalar(n);

    /* Call mlfMrank, the compiled version of mrank.m. */
    mlfMrank(1, &R, N);

    /* Print the results. */
    mlfPrintmatrix(R);

    /* Free the matrices allocated during this computation. */
    mxDestroyArray(N);
    mxDestroyArray(R);

    libPkgTerminate(); /* Terminate library of M-functions */
    mclTerminateApplication();
}
```

Explanation of mrankp.c

The heart of `mrankp.c` is a call to the `mlfMrank` function. Most of what comes before this call is code that creates an input argument to `mlfMrank`. Most of what comes after this call is code that displays the vector that `mlfMrank` returns. First, the code must initialize the MCR and the generated `libPkg` library.

```
mclInitializeApplication(NULL,0);
libPkgInitialize(); /* Initialize the library of M-Functions */
```

To understand how to call `mlfMrank`, examine its C function header, which is

```
void mlfMrank(int nargout, mxArray** r, mxArray* n);
```

According to the function header, `mlfMrank` expects one input parameter and returns one value. All input and output parameters are pointers to the `mxArray` data type. (See the External Interfaces documentation for details on the `mxArray` data type.)

To create and manipulate `mxArray *` variables in your C code, you can call the `mx` routines described in the External Interfaces documentation. For example, to create a 1-by-1 `mxArray *` variable named `N` with real data, `mrankp` calls `mxCreateDoubleScalar`.

```
N = mxCreateDoubleScalar(n);
```

`mrankp` can now call `mlfMrank`, passing the initialized `N` as the sole input argument.

```
R = mlfMrank(1,&R,N);
```

`mlfMrank` returns its output in a newly allocated `mxArray *` variable named `R`. The variable `R` is initialized to `NULL`. Output variables that have not been assigned to a valid `mxArray` should be set to `NULL`. The easiest way to display the contents of `R` is to call the `mlfPrintmatrix` function.

```
mlfPrintmatrix(R);
```

This function is defined in `Printmatrix.m`.

Finally, `mrankp` must free the heap memory allocated to hold matrices and call the termination functions.

```
mxDestroyArray(N);
mxDestroyArray(R);
libPkgTerminate(); /* Terminate the library of M-functions */
mclTerminateApplication(); /* Terminate the MCR */
```

Advanced C Example

This section provides an advanced example that illustrates how to write C code that calls a compiled M-file. Consider a standalone application whose source code consists of the files:

- `multarg.m`, which contains a function named `multarg`
- `multargp.c`, which contains C wrapper code that calls the C interface function for the M-code
- `printmatrix.m`, which contains the helper function to print a matrix to the screen

- `main_for_lib.c`, which contains one main function
- `main_for_lib.h`, which is the header for structures used in `main_for_lib.c` and `multargp.c`

`multarg.m` specifies two input parameters and returns two output parameters.

```
function [a,b] = multarg(x,y)
a = (x + y) * pi;
b = svd(svd(a));
```

The code in `multargp.c` calls `mlfMultarg` and then displays the two values that `mlfMultarg` returns.

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "libMultpkg.h"

/*
 * Function prototype; MATLAB Compiler creates mlfMultarg
 * from multarg.m
 */

void PrintHandler( const char *text )
{
    printf(text);
}

int main( ) /* Programmer-written coded to call mlfMultarg */
{
#define ROWS 3
#define COLS 3
    mclOutputHandlerFcn PrintHandler;
    mxArray *a = NULL, *b = NULL, *x, *y;
    double x_pr[ROWS * COLS] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    double x_pi[ROWS * COLS] = {9, 2, 3, 4, 5, 6, 7, 8, 1};
    double y_pr[ROWS * COLS] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    double y_pi[ROWS * COLS] = {2, 9, 3, 4, 5, 6, 7, 1, 8};
    double *a_pr, *a_pi, value_of_scalar_b;
```



```

/* Initialize with a print handler to tell mlfPrintMatrix
 * how to display its output.
 */
mclInitializeApplication(NULL,0);
libMultpkgInitializeWithHandlers(PrintHandler,PrintHandler);

/* Create input matrix "x" */
x = mxCreateDoubleMatrix(ROWS, COLS, mxCOMPLEX);
memcpy(mxGetPr(x), x_pr, ROWS * COLS * sizeof(double));
memcpy(mxGetPi(x), x_pi, ROWS * COLS * sizeof(double));

/* Create input matrix "y" */
y = mxCreateDoubleMatrix(ROWS, COLS, mxCOMPLEX);
memcpy(mxGetPr(y), y_pr, ROWS * COLS * sizeof(double));
memcpy(mxGetPi(y), y_pi, ROWS * COLS * sizeof(double));

/* Call the mlfMultarg function. */
mlfMultarg(2, &a, &b, x, y);

/* Display the entire contents of output matrix "a". */
mlfPrintmatrix(a);

/* Display the entire contents of output scalar "b" */
mlfPrintmatrix(b);

/* Deallocate temporary matrices. */
mxDestroyArray(a);
mxDestroyArray(b);
libMultpkgTerminate();
mclTerminateApplication();
return(0);
}

```

You can build this program into a standalone application by entering this command on a single line:

```

mcc -W lib:libMultpkg -T link:exe multarg printmatrix
multargp.c main_for_lib.c

```

The program first displays the contents of a 3-by-3 matrix *a*, and then displays the contents of scalar *b*.

```
6.2832 +34.5575i  25.1327 +25.1327i  43.9823 +43.9823i
12.5664 +34.5575i  31.4159 +31.4159i  50.2655 +28.2743i
18.8496 +18.8496i  37.6991 +37.6991i  56.5487 +28.2743i

143.4164
```

Explanation of This C Code

Invoking MATLAB Compiler on `multarg.m` generates the C function prototype.

```
extern void mlfMultarg(int nargout, mxArray** a, mxArray** b,
mxArray* x, mxArray* y);
```

This C function header shows two input arguments (`mxArray* x` and `mxArray* y`) and two output arguments (the return value and `mxArray** b`).

Use `mxCreateDoubleMatrix` to create the two input matrices (*x* and *y*). Both *x* and *y* contain real and imaginary components. The `memcpy` function initializes the components, for example:

```
x = mxCreateDoubleMatrix(,ROWS, COLS, mxCOMPLEX);
memcpy(mxGetPr(x), x_pr, ROWS * COLS * sizeof(double));
memcpy(mxGetPi(y), x_pi, ROWS * COLS * sizeof(double));
```

The code in this example initializes variable *x* from two arrays (`x_pr` and `x_pi`) of predefined constants. A more realistic example would read the array values from a data file or a database.

After creating the input matrices, `main` calls `mlfMultarg`.

```
mlfMultarg(2, &a, &b, x, y);
```

The `mlfMultarg` function returns matrices *a* and *b*. *a* has both real and imaginary components; *b* is a scalar having only a real component. The program uses `mlfPrintmatrix` to output the matrices, for example:

```
mlfPrintmatrix(a);
```

Libraries

This chapter describes how to use MATLAB Compiler to create libraries.

- “Introduction” on page 7-2
- “Addressing mxArray Arrays Above the 2 GB Limit” on page 7-3
- “C Shared Library Target” on page 7-4
- “C++ Shared Library Target” on page 7-18
- “MATLAB® Compiler Generated Interface Functions” on page 7-24
- “Using C/C++ Shared Libraries on a Mac OS X System” on page 7-34
- “About Memory Management and Cleanup” on page 7-38

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 C, “C++ Utility Library Reference”, for detailed changes to `mwArray` classes and method signatures.

C Shared Library Target

In this section...
“C Shared Library Wrapper” on page 7-4
“C Shared Library Example” on page 7-4
“Calling a Shared Library” on page 7-11

C Shared Library Wrapper

The C library wrapper option allows you to create a shared library from an arbitrary set of M-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 M-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 cplib` when including any MATLAB Compiler generated code into a larger application.

Note `mclmcrtr.lib` is required for successful linking. For more information, see the MathWorks Support database and search for information on the MSVC shared library.

C Shared Library Example

This example takes several M-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 the Driver Application

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.)

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.

Note You must call `mclInitializeApplication` once at the beginning of your driver application. You must make this call before calling any other MathWorks functions or when linking to a MATLAB library such as `mclmcrmt.lib` (for example, before accessing an `MWArray`). See “Calling a Shared Library” on page 7-11 for complete details on using a MATLAB Compiler generated library in your application.

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 `m1f` functions generated by MATLAB Compiler from your M-file routines has changed from earlier versions of the product. The generic signature of the exported `m1f` functions is

- M-functions with no return values

```
bool MW_CALL_CONV m1f<function-name>(<list_of_input_variables>);
```

- M-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 7-4 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.

Note `MCRInstaller.exe` has obsoleted the need for the function `buildmcr` or the creation of `MCRInstaller.zip`. See “Replacement of `MCRInstaller.zip` and `BUILDMCR` Functionality” on page 1-19 for more details including complete file paths to all install programs.

Component	Description
MCRInstaller.exe (Windows)	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform.
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: <ul style="list-style-type: none">• Windows — .dll• Solaris, 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.

Note MCRInstaller.exe has obsoleted the need for the function buildmcr or the creation of MCRInstaller.zip. See “Replacement of MCRInstaller.zip and BUILDMCR Functionality” on page 1-19 for more details including complete file paths to all install programs.

Component	Description
MCRInstaller.exe	(Windows) Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform
libmatrix	Shared library; extension varies by platform, for example, DLL on Windows
libmatrix.h	Library header file

Calling a Shared Library

At run-time, 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 run-time.

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`

You may also have a need to use `mclInhibitShutdown` when initializing and terminating an application repeatedly.

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`.

The function signatures are

```
bool mclInitializeApplication(const char **options, int count);
bool mclTerminateApplication(void);
```

mclInitializeApplication. Takes an array of strings of user-settable options (these are the very 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.

This C example shows typical usage of the functions:

```
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;
mlfMultiplmatrix(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.

Initializing and Terminating Your Application Multiple Times With `mclInhibitShutdown`

Sometimes, repeated initialization and termination of an application may cause the application to hang. To avoid this, call `mclInhibitShutdown()` immediately after calling `mclInitializeApplication()`. For example:

```
    if( !mclInitializeApplication(NULL,0) )
    {
        fprintf(stderr,
            "Could not initialize the application.\n");
        *err = -1;
        return(x);
    }
    mclInhibitShutdown();
```

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 `libnameInitialize()`, 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.

Loading Libraries in a Compiled Function

With MATLAB Compiler version 4.0 (R14) and later, you can use M-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 command at the MATLAB command prompt:

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

where *mylibrarymfile* is the name of an M-file you would like to use when loading this library. This step only needs to be performed once to generate an M-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)
```

With MATLAB Compiler versions 4.0.1 (R14+) and later, generated M-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 M-file in the compilation with the `-a` option with `mcc`.

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

Note You can use your operating system's `loadlibrary` function to call a MATLAB Compiler shared library function as long as you first call the initialization and termination functions `mclInitializeApplication()` and `mclTerminateApplication()`.

C++ Shared Library Target

In this section...
“C++ Shared Library Wrapper” on page 7-18
“C++ Shared Library Example” on page 7-18

C++ Shared Library Wrapper

The C++ library wrapper option allows you to create a shared library from an arbitrary set of M-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 M-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 “Mixing M-Files and C or C++” on page 6-10.

C++ Shared Library Example

This example rewrites the previous C shared library example using C++. The procedure for creating a C++ shared library from M-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.

This example uses a C++ version of the `matrixdriver` application, `matrixdriver.cpp`. In the C++ version, 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.

```

/*=====
 *
 *  MATRIXDRIVER.CPP
 *  Sample driver code that calls a C++ shared library created
 *  using MATLAB Compiler. Refer to MATLAB Compiler documentation
 *  for more information on this
 *
 *  This is the wrapper CPP code to call a shared library
 *  created using the MATLAB Compiler.
 *
 *  Copyright 1984-2005 The MathWorks, Inc.
 *
 *=====*/

// Include the library specific header file as generated by the
// MATLAB Compiler
#include "libmatrixp.h"

int run_main(int argc, char **argv)
{
    // Call application and library initialization.

```

```
// Perform init before calling any API functions or
// Compiler-generated libraries.
if (!mclInitializeApplication(NULL,0))
{
    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;
}
else
{
    try
    {
        // Create input data
        double data[] = {1,2,3,4,5,6,7,8,9};
        mxArray in1(3, 3, mxDOUBLE_CLASS, mxREAL);
        mxArray in2(3, 3, mxDOUBLE_CLASS, mxREAL);
        in1.SetData(data, 9);
        in2.SetData(data, 9);

        // Create output array
        mxArray out;

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

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

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

```

        std::cout << out << std::endl;

        eigmatrix(1, out, in1);
        std::cout << "Eigenvalues of the first matrix are:"
                    << std::endl;
        std::cout << out << std::endl;
    }
    catch (const mwException& e)
    {
        std::cerr << e.what() << std::endl;
        return -2;
    }
    catch (...)
    {
        std::cerr << "Unexpected error thrown" << std::endl;
        return -3;
    }
    // Call the application and library termination routine
    libmatrixpTerminate();
}
/* You should call mclTerminate application at the end of
 * your application.
 */
    mclTerminateApplication();
    return 0;
}

int main()
{
    mclmcrInitialize();
    return mclRunMain((mclMainFcnType)run_main,0,NULL);
}

```

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 7-14. 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 M-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:

M-Functions with No Return Values.

```
bool MW_CALL_CONV mlx<function-name>(<list_of_input_variables>);
```

M-Functions with at Least One Return Value.

```
bool MW_CALL_CONV mlx<function-name>(int number_of_return_values),
```



```
<list_of_return_variables>, <list_of_input_variables>);
```

In this case, `<list_of_input_variables>` represents a comma-separated list of type `const mxArray&` and `<list_of_return_variables>` represents a comma-separated list of type `mxArray&`. For example, in the `libmatrix` library, the C++ interfaces to the `addmatrix` M-function is generated as:

```
void addmatrix(int nargout, mxArray& a , const mxArray& a1,  
              const mxArray& 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.

MATLAB Compiler Generated Interface Functions

In this section...

“Functions in the Shared Library” on page 7-24

“Type of Application” on page 7-24

“Structure of Programs That Call Shared Libraries” on page 7-26

“Library Initialization and Termination Functions” on page 7-27

“Print and Error Handling Functions” on page 7-28

“Functions Generated from M-Files” on page 7-30

“Retrieving MCR State Information While Using Shared Libraries” on page 7-33

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 M-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.
```

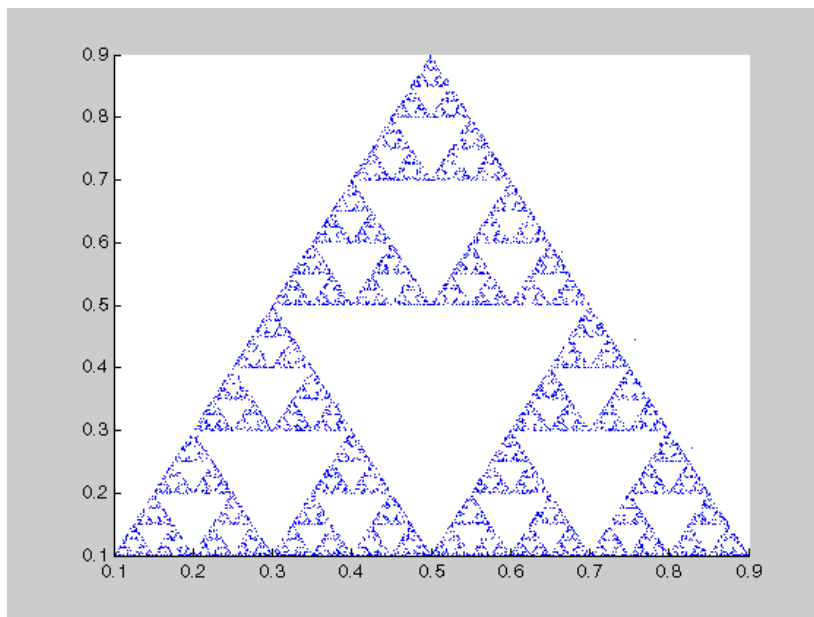
For a C++ Application on Windows

```
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 7-28). 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 7-11 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 M-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 M-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);
```

Omitting `extern "C"` will generate warnings on the Solaris platform.

Functions Generated from M-Files

For each M-file specified on the MATLAB Compiler command line, the product generates two functions, the `m1x` function and the `m1f` function. Each of these generated functions performs the same action (calls your M-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 M-file (`sierpinski`, in this example); each function begins with a different three-letter prefix.

Note For C shared libraries, MATLAB Compiler generates the `m1x` and `m1f` functions as described in this section. For C++ shared libraries, the product generates the `m1x` function the same way it does for the C shared library. However, the product generates a modified `m1f` function with these differences:

- The `m1f` before the function name is dropped to keep compatibility with R13.
 - The arguments to the function are `mwArray` instead of `mxArray`.
-

m1x Interface Function

The function that begins with the prefix `m1x` 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 M-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 noninitialized (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 an M-Function Interface

If your M-function interface uses `varargin` or `varargout`, you must pass them as cell arrays. For example, if you have `N` `varargin`s, you need to create one cell array of size 1-by-`N`. Similarly, `varargout`s 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 `varargout` has to be the last return variable (the variable preceding the first input variable) and the cell array representing `varargin` 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 M-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 `m1fFOO(NULL)` (the compiled M-code interprets this as `nargin=0`). However, calling `FOO((mwArray)NULL)` with the C++ shared library interface causes the compiled M code to see an empty array as the first input and interprets `nargin=1`.

For example, compile some M code as a C++ shared library using `varargin` as the M function's list of input arguments. Have the M 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`.

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 “Retrieving MCR Attributes” on page 4-24.

Using C/C++ Shared Libraries on a Mac OS X System

To use a MATLAB Compiler generated library on a Mac OS X system, you must create a separate thread that initializes the shared library and call that library's functions. The main thread of the application is the thread that calls your driver program's `main()` function. The body of your `main()` function must create a new thread, passing to it the address of a thread-function containing the library initialization and necessary calls to the shared library generated by MATLAB Compiler. The new thread does the main work of the application, including calling MATLAB Compiler generated libraries.

In past releases, it was necessary to create and execute a `CFRunLoop` in the main thread, as well as to call `mclSetExitCode`. Now, however, the same functionality can be accomplished with a single call to `mclRunMain`.

The following 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 the earlier example to build and run this application.

```
/*=====
 *
 * MATRIXDRIVER.C Sample driver code that calls the shared
 *      library created using MATLAB Compiler. Refer to the
 *      documentation of MATLAB Compiler for more info
 *      on this
 *
 * This is the wrapper C code to call a shared library created
 * using MATLAB Compiler.
 *
 * Copyright 1984-2005 The MathWorks, Inc.
 *
 *=====*/

#include <stdio.h>

#ifdef __APPLE_CC__
#include <CoreFoundation/CoreFoundation.h>
#endif

/* Include the MCR header file and the library specific header
```

```
* file as generated by MATLAB Compiler */
#include "libmatrix.h"

/* This function displays double matrix stored in mxArray */
void display(const mxArray* in);

void *run_main(void *x)
{
    int *err = x;
    mxArray *in1, *in2; /* Define input parameters */
    mxArray *out = NULL; /* and output parameters to be
                          * passed to lib functions */

    double data[] = {1,2,3,4,5,6,7,8,9};

    /* Call the mclInitializeApplication routine. Make sure that
     * the application was initialized properly by checking the
     * return status. This initialization has to be done before
     * calling any MATLAB API's or MATLAB Compiler generated
     * shared library functions. */
    if( !mclInitializeApplication(NULL,0) )
    {
        fprintf(stderr, "Could not initialize application.\n");
        *err = -1;
        return(x);
    }

    /* 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 initialization routine and make sure that
     * the library was initialized properly. */
    if (!libmatrixInitialize()){
        fprintf(stderr, "Could not initialize the library.\n");
        *err = -2;
    }
    else
```

```
    {
        /* 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 the return value since this variable will be reused
     * in the next function call. Since we are going to reuse the
     * variable, we have to set it to NULL. Refer to
     * 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 eigenvalues of the first matrix are:\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;
    }
    /* On MAC, you need to call mclRunMain with the appropriate
     * exit status. Also, you should call mclmcrInitialize
     * application before you call mclRunMain.
     * mclTerminateApplication terminates the entire application. */
    mclTerminateApplication();
    return 0
    }
    int main()
    {
        mclmcrInitialize();
        return mclRunMain((mclmainFcn)run_main,0,NULL);}
}
```

```
/*DISPLAY This function will display the double matrix
 * stored in an mxArray. This function assumes that
 * the mxArray passed as input contains double array.
 */
void display(const mxArray* in)
{
    int i=0, j=0; /* loop index variables */
    int r=0, c=0; /* variables to store the row
                  * and column length of the matrix */
    double *data; /* variable to point to the double data stored
                  * within the mxArray */

    /* Get the size of the matrix */
    r = mxGetM(in);
    c = mxGetN(in);
    /* Get a pointer to the double data in mxArray */
    data = mxGetPr(in);

    /* Loop through data and display in matrix format */
    for( i = 0; i < c; i++ ){
        for( j = 0; j < r; j++){
            printf("%4.2f\t",data[j*c+i]);
        }
        printf("\n");
    }
    printf("\n");
}
```

The Mac version of the `matrixdriver` application differs from the version on other platforms in these significant ways:

- 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.
- In this example, the `main()` function creates a new thread using `pthread_create`, and passes the address of the `run_main()` function to it.

About Memory Management and Cleanup

In this section...
“Overview” on page 7-38
“Passing mxArray to Shared Libraries” on page 7-38

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 M. This will reduce the number of calls to the memory manager, and the degree to which the heap fragments.

Passing mxArray 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 mxArray (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 7-11.

For more information about mxArray, see “Using mxArray” on page 5-24.

Troubleshooting

- “Introduction” on page 8-2
- “Common Issues” on page 8-3
- “Failure Points and Possible Solutions” on page 8-4
- “mbuild” on page 8-14
- “MATLAB® Compiler” on page 8-16
- “Deployed Applications” on page 8-20

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 (M-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 M-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 M-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 Chapter 7, “Libraries” 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 8-4 for common solutions.

Failure Points and Possible Solutions

In this section...
“How to Use this Section” on page 8-4
“Does the Failure Occur During Compilation?” on page 8-4
“Does the Failure Occur When Testing Your Application?” on page 8-8
“Does the Failure Occur When Deploying the Application to End Users?” on page 8-11

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 M-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 M-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 8-16.

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.

Does your code compile with the LCC compiler?

The LCC compiler is a free compiler provided with MATLAB on Windows. If there are installation or path problems with other system compilers, you may be able to compile your application with LCC.

Did the M-code compile successfully before this failure?

The three most common reasons for M-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 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.

If you are compiling a driver application, are you using mbuild?

The 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 `mclmcrst.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 `mclmcrst.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 Chapter 7, “Libraries” 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.

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 Chapter 4, “Deployment Process”, 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 M-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 “Directories Required for Development and Testing” on page 10-2.

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 M-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 M-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 8-16 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 `mclmcr7x.dll` or `mclmcr7x.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 4-21.

Caution Do not solve these problems by moving libraries or other files within the MCR folder structure. The runtime 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 “MATLAB® Compiler Generated Interface Functions” on page 7-24.

Some key points to consider to avoid errors at run time:

- Ensure that the calls to `mclinitializeApplication` and `libnameInitialize` 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 reinitialize (call `mclinitializeApplication`) after terminating it with `mclTerminateApplication`. The `mclinitializeApplication` and `libnameInitialize` 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 execute `MCRInstaller` in order 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.

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 4-21 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 4-21.

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 4-21.

Caution Do not solve these problems by moving libraries or other files within the MCR folder structure. The runtime 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 M-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.

mbuild

This section identifies some of the more common problems that might occur when configuring mbuild to create standalone applications.

Options File Not Writeable. 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 writeable, 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 writeable, 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

```
gcc -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 reenale your antivirus software.

Verification of `mbuild` Fails. If none of the previous solutions addresses your difficulty with `mbuild`, contact Technical Support at The MathWorks at http://www.mathworks.com/contact_TS.html.

MATLAB Compiler

Typically, problems that occur when building standalone C and C++ 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.

libmwapack: 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 The MathWorks. A list of contacts at The MathWorks is provided at the beginning of this document.

MATLAB Compiler Does Not Generate the Application. If you experience other problems with MATLAB Compiler, contact Technical Support at The 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 M-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 9-3 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.

Unable to Run MCRInstaller.exe on a Target Windows Machine. If you receive the message

```
This advertised application would not be installed because it
might be Unsafe. Contact your administrator to change the
installation user interface option of the package to basic.
```

when you try to install MATLAB Compiler Runtime (MCR) using `MCRInstaller.exe` on a Windows machine, you need to log in as an administrator. If this is not possible and you have no objection to installing the MCR in the default location, try the following command from a DOS window:

```
msiexec /qb /I MCRInstaller.msi
```

`MCRInstaller.msi` should have been placed in the installation folder after your first attempt to install the MCR. This command will start the installer using the basic UI configuration, which will execute at a lower security level.

Warning C:\WORK\R2008B~1\LCC\foo_delay_load.c: 21 static 'void function(void) FailedToLoadMCR' is not referenced. This warning message is produced as indirect output from of an internal delay load job that is only seen by Microsoft Visual C++ compiler users. The message is benign and should be ignored.

warning LNK4248: unresolved typeref token (01000028) for 'mxArray_tag'; image may not run test3.obj. If you receive this message while compiling an MSVC application that calls a MATLAB Compiler generated shared library, you can safely ignore it. The message is due to changes in the Visual C++[®] 2005 compiler and will not interfere with successful running of your application. If you desire, you can suppress the message by including an empty definition for `mxArray_tag` inside your `.cpp` file (`test3.cpp`, in this case). For example, if you add the line

```
struct mxArray_tag {};
```

at the beginning of your code and after the `include` statements, the warning will not recur.

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 M-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/<M-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 M-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 M-file <M-file> cannot be executed.

MATLAB:err_parse_cannot_run_m_file. This message is an indication that the MCR has found nonencrypted M-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 M-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 mclmcr7x.dll was not found. Re-installing the application may fix this problem.

`mclmcr7x.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:

```
mclmcr rt.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`, `watcom`, or `lcc`.

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 mclmcr rt76.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.

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.

Limitations and Restrictions

- “Limitations About What May Be Compiled” on page 9-2
- “Unsupported Functions” on page 9-9

Limitations About What May Be Compiled

In this section...

- “Compiling MATLAB and Toolboxes” on page 9-2
- “Fixing Callback Problems: Missing Functions” on page 9-3
- “Finding Missing Functions in an M-File” on page 9-5
- “Suppressing Warnings on the UNIX System” on page 9-5
- “Cannot Use Graphics with the -nojvm Option” on page 9-6
- “Cannot Create the Output File” on page 9-6
- “No M-File Help for Compiled Functions” on page 9-6
- “No MCR Versioning on Mac OS X” on page 9-6
- “Older Neural Networks Not Deployable with MATLAB® Compiler” on page 9-7
- “Restrictions on Calling PRINTDLG with Multiple Arguments in Compiled Mode” on page 9-7
- “Compiling a Function with WHICH Does Not Search Current Working Directory” on page 9-8

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 “Unsupported Functions” on page 9-9 for a list of functions that cannot be compiled.

Fixing Callback Problems: Missing Functions

When MATLAB Compiler creates a standalone application, it compiles the M-file(s) you specify on the command line and, in addition, it compiles any other M-files that your M-file(s) calls. MATLAB Compiler uses a dependency analysis, which determines all the functions on which the supplied M-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 M-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` M-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 “Specifying the Value of Callback Function Properties” in the MATLAB Programming Fundamentals documentation.

Using the -a Option. Instead of using the `%#function` pragma, you can specify the name of the missing M-file on the MATLAB Compiler command line using the `-a` option.

Finding Missing Functions in an M-File

To find functions in your application that may need to be listed in a `%#function` pragma, search your M-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 M-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 `app-defaults` warnings, set `XAPPLRESDIR` to point to `<mc_r_root>/<ver>/X11/app-defaults`.
- To suppress the `libjvm.so` warning, make sure you set the dynamic library path properly for your platform. See “Directories Required for Run-Time Deployment” on page 10-5.

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 M-File Help for Compiled Functions

If you create an M-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, M-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 a 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`:

- 1 Use the `pwd` function to explicitly point to the file in the current folder, as follows:

```
open([pwd 'myFile.mat'])
```

- 2 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
```

- 3 Use `addpath(pwd)` to add the current folder to the deployed application's file search path. This folder should not include any unencrypted M-files, as these cannot be executed by the compiled application.
- 4 Include your file using the **Other Files** area of your project using `deploytool` (and the `-a` flag using `mcc`).

Unsupported Functions

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 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

`add_block`
`add_line`

List of Unsupported Functions (Continued)

applescript
close_system
colormapeditor
createClassFromWsd1
dbclear
dbcont
dbdown
dbquit
dbstack
dbstatus
dbstep
dbstop
dbtype
dbup
delete_block
delete_line
depfun
doc
echo
edit
fields
figure_palette
get_param
help
home
inmem

List of Unsupported Functions (Continued)

keyboard
linmod
mislocked
mlock
more
munlock
new_system
open_system
pack
plotbrowser
plotedit
plottools
profile
profsave
propedit
propertyeditor
publish
rehash
restoredefaultpath
run
segment
set_param
sim
simget
simset
sldebug
type

Reference Information

- “Directories Required for Development and Testing” on page 10-2
- “Directories Required for Run-Time Deployment” on page 10-5
- “MATLAB® Compiler Licensing” on page 10-8
- “Using MCR Installer Command Line Options” on page 10-9

Directories Required for Development and Testing

In this section...
“Overview” on page 10-2
“Path for Java Development on All Platforms ” on page 10-2
“Path Modifications Required for Accessibility” on page 10-2
“Windows Settings for Development and Testing” on page 10-3
“UNIX Settings for Development and Testing” on page 10-3

Overview

The following information is for programmers developing applications that use libraries or components that contain compiled M-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 “Directories Required for Run-Time Deployment” on page 10-5.

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 5-29.

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
```

UNIX 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

```
setenv LD_LIBRARY_PATH
matlabroot/runtime/glnx86:
matlabroot/bin/glnx86:
matlabroot/sys/os/glnx86:
matlabroot/sys/java/jre/glnx86/jre/lib/i386/native_threads:
matlabroot/sys/java/jre/glnx86/jre/lib/i386/server:
matlabroot/sys/java/jre/glnx86/jre/lib/i386:
setenv XAPPLRESDIR matlabroot/X11/app-defaults
```

Linux x86-64

```
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:
setenv XAPPLRESDIR matlabroot/X11/app-defaults
```

Solaris 64

```
setenv LD_LIBRARY_PATH
matlabroot/runtime/sol64:
```

```
matlabroot/bin/sol64:  
matlabroot/sys/os/sol64:  
matlabroot/sys/java/jre/sol64/jre/lib/sparcv9/native_threads:  
matlabroot/sys/java/jre/sol64/jre/lib/sparcv9/server:  
matlabroot/sys/java/jre/sol64/jre/lib/sparcv9:  
setenv XAPPLRESDIR matlabroot/X11/app-defaults
```

Intel Mac (Maci)

```
setenv DYLD_LIBRARY_PATH  
matlabroot/runtime/maci:  
matlabroot/bin/maci:  
matlabroot/sys/os/maci:  
/System/Library/Frameworks/JavaVM.framework/JavaVM:  
/System/Library/Frameworks/JavaVM.framework/Libraries  
setenv XAPPLRESDIR matlabroot/X11/app-defaults
```

Intel Mac (Maci64)

```
setenv DYLD_LIBRARY_PATH  
matlabroot/runtime/maci64:  
matlabroot/bin/maci64:  
matlabroot/sys/os/maci64:  
/System/Library/Frameworks/JavaVM.framework/JavaVM:  
/System/Library/Frameworks/JavaVM.framework/Libraries  
setenv XAPPLRESDIR matlabroot/X11/app-defaults
```


Directories Required for Run-Time Deployment

In this section...

- “General Path Guidelines” on page 10-5
- “Path for Java Applications on All Platforms” on page 10-5
- “Windows Path for Run-Time Deployment” on page 10-5
- “UNIX Paths for Run-Time Deployment” on page 10-6

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 runtime errors caused by path-related issues.

Path for Java Applications on All Platforms

When your users run applications that contain compiled M-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.

UNIX Paths for Run-Time Deployment

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 “Compiling and Linking MAT-File Programs” for more information.

Linux

```
setenv LD_LIBRARY_PATH
  mcr_root/version/runtime/glnx86:
  mcr_root/version/bin/glnx86:
  mcr_root/version/sys/os/glnx86:
  mcr_root/version/sys/java/jre/glnx86/jre/lib/i386/native_threads:
  mcr_root/version/sys/java/jre/glnx86/jre/lib/i386/server:
  mcr_root/version/sys/java/jre/glnx86/jre/lib/i386:
setenv XAPPLRESDIR mcr_root/version/X11/app-defaults
```

Linux x86-64

```
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:  
setenv XAPPLRESDIR mcr_root/version/X11/app-defaults
```

Solaris 64

```
setenv LD_LIBRARY_PATH  
mcr_root/version/runtime/sol64:  
mcr_root/version/bin/sol64:  
mcr_root/version/sys/os/sol64:  
mcr_root/version/sys/java/jre/sol64/jre/lib/sparcv9/native_threads:  
mcr_root/version/sys/java/jre/sol64/jre/lib/sparcv9/server:  
mcr_root/version/sys/java/jre/sol64/jre/lib/sparcv9:  
setenv XAPPLRESDIR mcr_root/version/X11/app-defaults
```

Intel Mac (Maci)

```
setenv DYLD_LIBRARY_PATH  
mcr_root/version/runtime/maci:  
mcr_root/version/bin/maci:  
mcr_root/version/sys/os/maci:  
/System/Library/Frameworks/JavaVM.framework/JavaVM:  
/System/Library/Frameworks/JavaVM.framework/Libraries  
setenv XAPPLRESDIR mcr_root/version/X11/app-defaults
```

Intel Mac (Maci64)

```
setenv DYLD_LIBRARY_PATH  
mcr_root/version/runtime/maci64:  
mcr_root/version/bin/maci64:  
mcr_root/version/sys/os/maci64:  
/System/Library/Frameworks/JavaVM.framework/JavaVM:  
/System/Library/Frameworks/JavaVM.framework/Libraries  
setenv XAPPLRESDIR mcr_root/version/X11/app-defaults
```

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).

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.

Using MCR Installer Command Line Options

In this section...

“Overview” on page 10-9

“Displaying MCR Installer Location and Related Information” on page 10-9

“Accessing MCR Installer Command Line Options on Windows Systems” on page 10-10

“Accessing MCR Installer Command Line Options on UNIX and Linux Systems” on page 10-12

Overview

If you distribute your compiled components to users who do not have MATLAB installed on their systems, they must install the MATLAB Compiler Runtime (MCR). To install the MCR, users of your component must run the MCR Installer for their platform: `MCRInstaller.exe` for Windows systems and `MCRInstaller.bin` for UNIX and Linux systems. (The MathWorks uses a different MCR installation program on Macintosh systems.)

The MCR Installer supports command line options that can be helpful in certain instances. For example, if you want to automate installation of the MCR for users of your component, you might want to run the MCR Installer in silent mode, where no interaction with the user is required. To accomplish this, you must use MCR Installer command line options. The following sections describe some commonly used command line options for Windows, UNIX, and Linux systems.

Displaying MCR Installer Location and Related Information

Use the `mcrinstaller` command to display information about the location of available MCR installers on your system. See the Chapter 11, “Function Reference” for more information about using this command.

Accessing MCR Installer Command Line Options on Windows Systems

The following table lists some frequently used command line options for the MCR Installer on Windows systems (`MCRInstaller.exe`). The MCR installer is implemented using both an InstallShield wizard and the Windows Installer tool, `Msiexec.exe`. Both installers support command line options and you can specify these options on the MCR installer command line. To specify `Msiexec` command line arguments, you must use the InstallShield `/v` argument. For more information about these tools, see the appropriate documentation from their vendors: InstallShield (the Basic MSI project options only) and `Msiexec.exe`. Examples of using these command line options follow.

Frequently Used MCR Installation Options on Windows Systems

Option	Description
<code>/a</code>	Performs installation as an administrator. This option is useful if you want to uncompress the installation so you can extract the MSI install and repackage it with your options. The <code>/a</code> option requires Windows Administrator access.
<code>/Ldecimal_language_ID</code>	Specifies the language (<i>decimal_language_ID</i>) to be used by a multi-language installation program.
<code>/n</code>	Runs without a GUI. This is an <code>Msiexec.exe</code> option that must be passed using the <code>/v</code> option.
<code>/q</code>	Runs in quiet mode. This is an <code>Msiexec.exe</code> option that must be passed using the <code>/v</code> option.
<code>/s</code>	Runs the installation in silent mode. There must be a space after <code>/s</code> .
<code>/v</code>	Pass command-line options and values of public properties to <code>Msiexec.exe</code> . Make sure there is no space after <code>/v</code> . Also, you can use double quotation marks (" ") to delimit the arguments to <code>/v</code> , but you still cannot have a space between <code>/v</code> and the enclosing quotation marks.

Frequently Used MCR Installation Options on Windows Systems (Continued)

Option	Description
/x	Uninstall the MCR.
/w	Wait until the installation is complete before exiting.

Example: Performing a Silent Installation

To perform a silent installation, you must use the /s option, to suppress the InstallShield wizard, and specify the /q and /n options to suppress the Msiexec GUI. In the following example, you must use the /v option to pass the Msiexec options, and there is no space between the /v and the quotation mark that encloses the Msiexec options. The example specifies the folder where you want to install the MCR. You only need to specify this if you don't want to use the default installation folder: *matlabroot*\MATLAB Compiler Runtime*mcrversion*, where *matlabroot* is the root folder for the MATLAB installation and *mcrversion* is the version number of the MCR. The example also specifies the language, using the /L option

```
MCRInstaller.exe /L1033 /s /v"/qn INSTALLDIR=D:\MCR\release"
```

Example: Blocking the Command Prompt

By using the /w option, you can initiate the MCR installer without blocking the command prompt. This is useful for silent installations.

To block the command prompt:

```
start /WAIT MCRInstaller.exe /w
```

To perform a silent install from a script and block command prompt:

```
start /WAIT MCRInstaller.exe /w /s /v/qn
```

To perform a silent uninstall from a script and block command prompt:

```
start /WAIT MCRInstaller.exe /w /x /v/qn
```

Example: Requesting a Verbose Log of the Install Process

The following command causes the installer to create a verbose log of the install process in C:\log.txt. For information about the Msiexec options passed with /v, see the Msiexec documentation.

```
MCRInstaller.exe /v"/L*v \"C:\log.txt\" "
```

Accessing MCR Installer Command Line Options on UNIX and Linux Systems

The following table lists some frequently used command line options for the MCR Installer on UNIX and Linux systems (MCRInstaller.bin). The MCR Installer on UNIX and Linux systems is implemented using the InstallShield Universal Java installer. The universal installer supports two types of command line options: installation launcher options and run-time options. You can specify any mix of these options on the MCR Installer command line. If you extract the Jar file from the installation launcher (using installation launcher options) and call it directly, you must use the run-time options. Examples of using these command line options follow.

Frequently Used MCR Installer Options on UNIX or Linux Systems

Option	Description
-cp:a <classpath>	Appends (add to the end) <classpath> to the launcher's classpath.
-cp:p <classpath>	Prepends (add to the beginning) <classpath> to the launcher's classpath
-is:extract	Extracts the contents of the archive.
-is:help	Displays command line option help text and exits.
-is:log <filename>	Specifies file in which to write debug messages.

Frequently Used MCR Installer Options on UNIX or Linux Systems (Continued)

Option	Description
-log	Specifies file in which to write debug messages (run-time version)
-is:nospacecheck	Turns off launcher disk space checking.
-P	Specifies properties of a product bean, such as the bean ID.
-is:silent	Prevents the display of the application launcher UI. To run in silent mode, use the <code>-silent</code> run-time option.
-silent	Specifies to install or uninstall the product in silent mode, where the installation/uninstallation is performed with no user interaction.
-is:tempdir <dir>	Specifies the temporary folder to be used by the launcher.

Uninstalling the MCR on UNIX Systems

To remove an MCR installation on UNIX systems, run the `uninstaller.bin` program.

It is important to run the uninstaller, rather than just removing the MCR installation folder using the `rm` command, because an MCR installation affects other aspects of your environment. For example, the MCR installer creates a folder named `InstallShield` in your UNIX home folder that stores information about your particular MCR installation. When you use the MCR uninstaller, it removes some of these additional folders.

To run the MCR uninstaller, follow this procedure:

- 1** Navigate to your MCR installation folder using the `cd` command.
- 2** `cd` into the `_uninst` folder

- 3** Run the `uninstaller.bin` program. If you want to run the uninstaller noninteractively, specify the `-silent` option on the command line.

Example: Extracting the Jar File from the MCR Installer

The MathWorks includes the JRE required by the MCR Installer. However, your Linux system might not support this version of the JRE. To work around this issue, you can extract the Jar file from the MCR installer (`MCRInstaller.bin`) and call the Jar file directly, using the JRE that works with your system. The following provides a step-by-step instructions.

- 1** Extract the Java Jar file from the application launcher, using the `is:extract` installation launcher option.

```
MCRInstaller.bin -is:extract
```

This option extracts the components, including the `Jar filesetup.jar`.

- 2** Run the Jar file using the JRE that works with your system.

```
java -jar setup.jar
```

This starts the installer GUI.

Example: Performing a Noninteractive (Silent) Installation of the MCR

To perform a noninteractive (silent) installation, you must use the `-silent` run-time option. This example also uses the `-P` option to specify a particular Java bean.

```
MCRInstaller.bin -P bean421.installLocation="desiredInstallPath" -silent
```

Function Reference

Pragmas (p. 11-2)

Command-Line Tools (p. 11-2)

API Functions (p. 11-3)

Directives to MATLAB Compiler

Deployment-related commands

Deployment API-related commands

Pragmas

`##external`

`##function`

Pragma to help MATLAB Compiler locate functions called through `feval`, `eval`, or Handle Graphics® callback

Command-Line Tools

`builder2prj`

Convert project files with suffixes `.cbl`, `.nbl`, and `.mxl` to `.prj` (`deploytool`) format

`ctfroot`

Root folder of application in deployed mode

`deployprint`

Use to print (as substitute for MATLAB print function) when working with deployed Windows applications

`deploytool`

Open Deployment Tool, GUI for MATLAB Compiler

`hardcopy`

Saves figure window to file

`isdeployed`

Determine whether code is running in deployed or MATLAB mode

`ismcc`

Test if code is running during compilation process (using `mcc`)

`mbuild`

Compile and link source files into standalone application or shared library

`mcc`

Invoke MATLAB Compiler

<code>mcrinstaller</code>	Displays version and location information for MCR installer(s) corresponding to current platform.
<code>mcrversion</code>	Determine version of installed MATLAB Compiler Runtime (MCR)

API Functions

<code><library>Initialize[WithHandlers]</code>	Initializes MCR instance associated with <i>library</i>
<code><library>Terminate</code>	Frees all resources allocated by MCR instance associated with <i>library</i>
<code>GetMcrID</code>	Return identifier of MCR instance associated with software component <code>libname</code>
<code>getmcruserdata</code>	Retrieve MATLAB array value associated with given string key
<code>mclGetLogFileName</code>	Retrieves name of log file used by the MCR
<code>mclGetMCRUserData</code>	Retrieve MATLAB data associated with string key of MCR instance uniquely identified by <code>mcrID</code>
<code>mclInitializeApplication</code>	Sets up application state shared by all (future) MCR instances created in current process. Called only once per process.
<code>mclIsJVMEEnabled</code>	Determines if MCR was launched with instance of a Java Virtual Machine (JVM)
<code>mclIsMCRInitialized</code>	Determines if MCR has been properly initialized

<code>mclIsNoDisplaySet</code>	Determines if <code>-nodisplay</code> mode is enabled.
<code>mclKillAllFigures</code>	Finds and deletes open figures
<code>mclSetCmdLineUserData</code>	Associate MATLAB data value with string key of MCR instance uniquely identified by <code>mcrID</code>
<code>mclSetMCRUserData</code>	Associate MATLAB data value with string key of MCR instance uniquely identified by <code>mcrID</code>
<code>mclTerminateApplication</code>	Closes down all MCR-internal application state. Called only once per process.
<code>mclWaitForFiguresToDie</code>	Enables deployed applications to process Handle Graphics events, enabling figure windows to remain displayed.
<code>setmcuserdata</code>	Associate MATLAB data value with string key

Functions — Alphabetical List

`%#external`
`%#function`
`builder2prj`
`ctfroot`
`deployprint`
`deploytool`
`GetMcrID`
`getmcruserdata`
`hardcopy`
`isdeployed`
`ismcc`
`<library>Initialize[WithHandlers]`
`<library>Terminate`
`mbuild`
`mcc`
`mclGetLogFileName`
`mclGetMCRUserData`
`mclInitializeApplication`
`mclIsJVMEEnabled`
`mclIsMCRInitialized`
`mclIsNoDisplaySet`
`mclKillAllFigures`
`mclSetCmdLineUserData`
`mclSetMCRUserData`
`mclTerminateApplication`
`mclWaitForFiguresToDie`

mcrinstaller
mcversion
setmcuserdata

Informs MATLAB® Compiler™ that the implementation of the MATLAB function containing this pragma is supplied by the user as a C/C++ file.

Syntax

```
 %#external
```

Description

This pragma affects only the single function in which it appears, and any MATLAB function can contain this pragma (`local`, `global`, `private`, or `method`).

If you compile a program that contains the `%#external` pragma, you must explicitly pass each file that contains this pragma on the `mcc` command line. `mcc` also expects a `.c` or `.cpp` file with the implementation to be passed in.

This pragma is not executed in MATLAB. It is treated as a comment. Only the MATLAB code is executed.

When you use this pragma, MATLAB Compiler generates an additional header file called `function_name_external.h`. In this header file, `function_name` is the name of the initial MATLAB function containing the `%#external` pragma. This header file contains the `extern` declaration of the function that the user must provide. This function must conform to the same interface as code generated by MATLAB Compiler.

Examples

In this example, the `collect` function calls a user-supplied helper function called `collect_one`, implemented in C/C++.

```
function collect

    y = zeros(1,100); % pre-allocate the matrix
    for i = 1:100
        y(i) = collect_one;
    end

function y = collect_one
    %#EXTERNAL
```

```
persistent t;  
  
if (isempty(t))  
    t = 0;  
else  
    t = t+0.05;  
end  
y = sin(t);
```

For complete examples, see “Interfacing M-Code to C/C++ Code” on page 5-14.

Related Links

- “Interfacing M-Code to C/C++ Code” on page 5-14
- External Interfaces documentation

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 C and C++ 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 M-files used in your application. This pragma adds the top-level function as well as all the subfunctions 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
```

%#function

```
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`.

Purpose Convert project files with suffixes `.cb1`, `.nb1`, and `.mx1` to `.prj` (deploytool) format

Syntax

```
builder2prj
builder2prj([project.cb1,project.nb1,project.mx1])
builder2prj([project.cb1,project.nb1,project.mx1],
            new_project.prj)
```

Description This function converts project files in older formats such as `.cb1`, `.nb1`, and `.mx1`, to a format usable by `deploytool` (`.prj`).

Examples **Example 1**

```
builder2prj;
```

Entering this command opens the Builder Project File dialog box, which enables you to browse for the project you wish to convert. Navigate to the `.cb1`, `.nb1`, or `.mx1` project file, select the file name, and click Open to start the conversion process.

Example 2

```
builder2prj(my_project.cb1);
```

In this example, `builder2prj` locates `my_project.cb1` in your present working folder and converts the file to `deploytool`-compatible format (`.prj` format). In this example, after `builder2prj` runs, only the file suffix (`.cb1`) has changed. The new project name is the same as the old project name, but with a new suffix (`my_project.prj`).

Example 3

```
builder2prj(my_project.mx1,renamed_project.prj);
```

By specifying a second file name argument, you can choose a specific name for your `deploytool`-compatible project. In this example, `my_project.mx1` is located in your present working folder and

builder2prj

builder2prj is run, converting the .mxl project to a .prj project. The new project is named renamed_project.pj.

Purpose Root folder of application in deployed mode

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.

To determine the location of various toolbox folders in deployed mode, use the `toolboxdir` function.

Example `appRoot = ctfroot;` will return the root of your applications 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).

deployprint

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.). <code>deployprint</code> only produces .BMP files. <hr/> Note To print to a file, use the <code>print</code> function. <hr/>
-noui	Used to suppress printing of user interface controls. Similar to use in MATLAB <code>print</code> function.
-setup	The <code>-setup</code> option is not supported.
-s <i>windowtitle</i>	MATLAB Compiler does not support Simulink®.

`deployprint` supports a subset of the figure properties supported by `print`. The following are supported:

- PaperPosition
- PaperSize

- PaperUnits
- 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`, `print`

deploytool

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.

See Chapter 1, “Getting Started”, to get started using the Deployment Tool to create standalone applications and libraries.

Purpose Return identifier of MCR instance associated with software component *libname*

Syntax `extern "C" long libnameGetMcrID`

Description The extern "C" long *libname*GetMcrID returns the identifier of the MCR instance associated with the software component *libname*. It allows applications that use more than one MATLAB Compiler generated software component (and hence, have more than one MCR instance active in-process) to distinguish between MCR instances when setting or getting MCR instance specific data. This function is automatically generated into every component created by MATLAB Compiler. This command is part of the MCR User Data interface API. For information about this function, as well as complete examples of usage, see "Improving Data Access Using the MCR User Data Interface" on page 4-26.

Example

```
if( !mclInitializeApplication(NULL,0) )
{
    fprintf(stderr, "Could not initialize the application.
        \n");
    return -1;
}

if (!libmatrixInitialize())
{
    fprintf(stderr, "Could not initialize the library.\n");
    return -2;
}

long mcrID = libmatrixGetMcrID();
mxArray *value = mxCreateString("/home/user/config.xml");
if (!mclSetMCRUserData(mcrID, "DCTConfigFile", value) )
{
    fprintf(stderr, "Could not set DCTConfigFile
        MCR user data. \n");
}
```

GetMcrID

```
        return -3;  
    }
```

```
    ...
```

See Also

`mclGetMCRUserData`, `mclSetMCRUserData`

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 “Improving Data Access Using the MCR User Data Interface” on page 4-26.

Example

```
function_value =  
    getmcruserdata('ParallelConfigurationFile');
```

See Also `setmcruserdata`

hardcopy

Purpose Saves figure window to file

Syntax `hardcopy(handle, 'filename', 'format')`

Description The `hardcopy(handle, 'filename', 'format')` command saves the figure window with *handle* to the designated *filename* in the specified *format*. Legal formats are:

- -dps
- -deps
- -dps2
- -deps2
- -dill
- -dhppl

Caution

Call this function when converting a figure to pixel data. This can be necessary in some Web deployments. Do *not* use this function merely as a substitute for the `print` function.

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 M-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.

Example

```
`% startup.m
    if( ismcc || isdeployed)
        addpath(fullfile(matlabroot, 'work'));
```

See Also `isdeployed`, `mcc`, `matlabrc`

<library>Initialize[WithHandlers]

Purpose

Initializes MCR instance associated with *library*

Syntax

```
bool libraryInitialize(void)
bool libraryInitializeWithHandlers(
    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.

Example

```
bool libtriangleInitialize(void)

bool libtriangleInitializeWithHandlers(
    mclOutputHandlerFcn error_handler,
    mclOutputHandlerFcn print_handler
)
```

See Also

“Library Initialization and Termination Functions” on page 7-27

<library>Terminate

<library>Terminate

Purpose Frees all resources allocated by MCR instance associated with *library*

Syntax `void libraryTerminate(void)`

Description This function should be called after you finish calling the functions in this MATLAB Compiler-generated library, but before `mclTerminateApplication` is called.

Example Call `libmatrixInitialize` to initialize `libmatrix` library properly near the start of your program:

```
/* Call the library initialization routine and ensure the
 * library was initialized properly. */
if (!libmatrixInitialize()){
    fprintf(stderr,"Could not initialize the library.\n");
    *err = -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 Initialization and Termination Functions” on page 7-27

<library>Initialize[WithHandlers]

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.

mbuild

Option	Description
-D<name>	Define a symbol name to the C preprocessor. Equivalent to a <code>#define <name></code> directive in the source.
-D<name>=<value>	Define a symbol name and value to the C preprocessor. Equivalent to a <code>#define <name> <value></code> 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 <code>-O</code> option).
-h[elp]	Print help for mbuild.
-I<pathname>	Add <pathname> to the list of folders to search for <code>#include</code> files.
-inline	Inline matrix accessor functions (mx*). The executable generated may not be compatible with future versions of MATLAB.
-l<name>	Link with object library. On Windows, <name> will be expanded to <name>.lib or lib<name>.lib and on UNIX to lib<name>. Note When linking with a library, it is essential that you first specify the path (with <code>-I<pathname></code> , for example).
-L<directory>	Add <directory> to the list of folders to search for libraries specified with the <code>-l</code> option.

Option	Description
-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>.
-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.)

Option	Description
-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>	<p>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., <code>COMPFLAGS="opt1 opt2"</code>), and on UNIX single quotes are used (e.g., <code>CFLAGS='opt1 opt2'</code>).</p> <p>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., <code>COMPFLAGS="\$COMPFLAGS opt2"</code> on Windows or <code>CFLAGS='\$CFLAGS opt2'</code> on UNIX).</p>

Caution

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
```

Note 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.

Note 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.

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.

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 M-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 M-file is specified on the command line, MATLAB Compiler generates a C or C++ function for each M-file. If C or object files are specified, they are passed to mbuild along with any generated C files.

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.

Note Currently, `*` is the only supported wildcard.

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.

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 M-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” on page 5-9 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 “Overriding Default CTF Archive Embedding Using the MCR Component Cache” on page 5-20 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,logfile' -v function_name
```

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 The MathWorks recommends that you use `mbuild -setup`.

-F Specified Project File

Specify that `mcc` use settings contained in the specified project file. Use

```
-F project_name.prj
```

to specify `project_name` as the project file name when calling `mcc`. This option enables the `.prj` file, along with all of its associated settings, to be fed back to `mcc`. Project files created using either `mcc` or `deploytool` are eligible to use this option. When using `-F`, no other arguments may be invoked against `mcc`.

-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 M-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 M-files, followed by `directory2`. This option is important for standalone compilation where the MATLAB path is not available.

-l Generate a Function Library

Macro to create a function library. This option generates a library wrapper function for each M-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
<code>-logfile filename</code>	Specify a log file name.
<code>-nodisplay</code>	Suppress the MATLAB <code>nodisplay</code> run-time warning.
<code>-nojvm</code>	Do not use the Java Virtual Machine (JVM).
<code>-startmsg</code>	Customizable user message displayed at MCR initialization time. See “Displaying MCR Initialization Start-Up and Completion Messages For Users” on page 4-33.
<code>-completemsg</code>	Customizable user message displayed when MCR initialization is complete. See “Displaying MCR Initialization Start-Up and Completion Messages For Users” on page 4-33.

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 7-11.

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.

-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`.

Note `MCRInstaller.exe` has obsoleted the need for the function `buildmcr` or the creation of `MCRInstaller.zip`. See “Replacement of `MCRInstaller.zip` and `BUILDMCR` Functionality” on page 1-19 for more details including complete file paths to all install programs.

-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
<code>-w list</code>	Generates a table that maps <code><string></code> to warning message for use with <code>enable</code> , <code>disable</code> , and <code>error</code> . Appendix B, “Error and Warning Messages”, lists the same information.
<code>-w enable</code>	Enables complete warnings.

Syntax	Description
-w disable[:<string>]	Disables specific warning associated with <string>. Appendix B, “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 B, “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.

-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 M-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 <code>main()</code> function.

Type	Description
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 M-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 M-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.

-Y License File

Use

```
-Y license.dat_file
```

to override the default license.dat file with the specified argument.

-z Specify Path

Specify the path for library and include files. Use

```
-z path
```

to specify path to use for the compiler libraries and include files instead of the path returned by matlabroot.

-? Help Message

Display MATLAB Compiler help at the command prompt.

Linux mcc Cache Management Command Options

The Bourne shell front-end interface to MATLAB Compiler uses a cache file to speed execution. The cache file contains precomputed values of critical environment variables. The cache is automatically built whenever the back-end interface is called, providing that the cache file doesn't already exist and the `-nocache` option is not used. Later executions of `mcc` will use it unless overridden by `-nocache`. Special command-line options are available to manage this cache, but they can only be executed on Linux in standalone command-line mode (not through MATLAB). The table summarizes these options.

Linux Command Option	Description
<code>-cache</code>	Prints the cache if used as the only argument. Can be used to rebuild the cache when used with other arguments in this table.
<code>-rmcache</code>	Removes the cache.
<code>-nocache</code>	Ignores the cache.

Examples

Make a standalone executable for `myfun.m`.

```
mcc -m myfun
```

Make a standalone executable for `myfun.m`, but look for `myfun.m` in the `/files/source` folder and put the resulting C files and in the `/files/target` folder.

```
mcc -m -I /files/source -d /files/target myfun
```

Make the standalone `myfun1` from `myfun1.m` and `myfun2.m` (using one `mcc` call).

```
mcc -m myfun1 myfun2
```

See Also
deploytool

Purpose Retrieves name of log file used by the 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 “Retrieving MCR Attributes” on page 4-24 in the user guide.

Example

```
printf("Logfile name : %s\n",mclGetLogFileName());
```

mclGetMCRUserData

Purpose Retrieve MATLAB data associated with string key of MCR instance uniquely identified by *mcrID*

Syntax

```
extern "C"  
mxAArray *mclGetMCRUserData(  
    long mcrID,  
    const char *key  
);
```

Description This external C function retrieves the MATLAB data associated with the string key of an MCR instance uniquely identified by *mcrID*. *mclGetMCRUserData* returns an empty matrix if no data has been associated with the specified key. The returned *mxArray* is a shared copy of the *mxArray* stored in the user data table. Dispose of this shared copy properly by calling *mxDestroyArray*.

This command is part of the MCR User Data interface API. For information about this function, as well as complete examples of usage, see “Improving Data Access Using the MCR User Data Interface” on page 4-26.

Example

```
mxArray *value = mclGetMCRUserData(mcrID,  
    "ParallelConfigurationFile");
```

See Also *mclSetCmdLineUserData*, *mclSetMCRUserData*

Purpose Sets up application state shared by all (future) MCR instances created in current process. Called only once per 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. 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

mclInitializeApplication

- -nodisplay
- -noFigureWindows
- -nojit
- -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.

Example

To start all MCRs in a given process with the -nodisplay option, for example, use the following code:

```
char *args[] = { "-nodisplay" };  
bool success = mclInitializeApplication(args, 1);
```

See Also

“Initializing and Terminating Your Application with mclInitializeApplication and mclTerminateApplication” on page 7-11

mclTerminateApplication

mclIsJVMEEnabled

Purpose Determines if MCR was launched with instance of a Java Virtual Machine (JVM)

Syntax `bool mclIsJVMEEnabled()`

Description Use `mclIsJVMEEnabled()` 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 “Retrieving MCR Attributes” on page 4-24 in the user guide.

Example `printf("JVM initialized : %d\n", mclIsJVMEEnabled());`

Purpose Determines 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 “Retrieving MCR Attributes” on page 4-24 in the user guide.

Example

```
printf("MCR initialized : %d\n", mclIsMCRInitialized());
```

mclIsNoDisplaySet

Purpose Determines 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 “Retrieving MCR Attributes” on page 4-24 in the user guide.

Note Always returns `false` on Windows systems since the `-nodisplay` option is not supported on Windows systems.

Example `printf("nodisplay set : %d\n",mclIsNoDisplaySet());`

Purpose Finds and deletes open figures

Syntax

Description This function uses the same internal algorithm to locate open figures as `mclWaitForFiguresToDie`.

Typically you use `mclKillAllFigures` when you need to kill figures that are being displayed as the result of a programming problem, such as an endless loop. Alternately, you might use it to ensure all figures are closed before the execution of another application.

Examples Following are three alternate ways of using `mclKillAllFigures` to terminate three figures displayed by shared library calls to `showknot()`, `showpeak()`, and `showbottle()`.

```
showknot();
showpeak();
showbottle();
mclKillAllFigures((HMCINSTANCE)0);
```

```
showknot();
showpeak();
showbottle();
mclKillAllFigures(0);
```

```
showknot();
showpeak();
showbottle();
mclKillAllFigures(NULL);
```

See Also `mclWaitForFiguresToDie`

“Terminating Figures by Force with the `mclKillAllFigures` Method”
on page 5-31

mclSetCmdLineUserData

Purpose Associate MATLAB data value with string key of MCR instance uniquely identified by `mcrID`

Syntax

```
extern "C"  
size_t mclSetCmdLineUserData(  
    long mcrID,  
    int argc,  
    const char **argv  
);
```

Description This external C function examines the arguments on the command line for the switch `-mcruserdata` and calls `mclSetMCRUserData` on the argument of any such switches that it finds. It then returns the new size of `argv` (the new value for `argc`). MCR user data is specified on the command line using the `-mcruserdata` switch, as in the following example:

```
% myapp -mcruserdata  
"ParallelConfigurationFile:/usr/userdir/config.mat"
```

Users may pass MCR user data to an application on the command line with the `-mcruserdata` switch. The switch argument syntax is `key:value`, where both `key` and `value` are strings. The `key` string may not contain any colons, but the `value` string may contain them. The string is split into `key` and `value` at the first colon. Multiple switches may appear on a command line.

Caution

Be aware that all standalone applications attempt to parse the command line. Use the `-mcruserdata` switch with great care.

Note The generated main function for a C/C++ application changes to call `mclSetCmdLineUserData` after initializing the MCR. Generated code for a software component or shared library does not change. It is the user's responsibility to call `mclSetMCRUserData` after calling the library initialization function.

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

This command is part of the MCR User Data interface API. For information about this function, as well as complete examples of usage, see “Improving Data Access Using the MCR User Data Interface” on page 4-26.

Examples

Call using this basic structure:

```
int run_main(int argc, const char **argv)
{
    // Other run_main code here ...

    // Get -mcruserdata switches from the command line
    size_t dataCount =
        mclSetCmdLineUserData(_mcr_inst, argc, argv);

    _retval = mclMain(_mcr_inst, argc, argv, "tctf", 0);

    // Other run_main code here ...
}
```

Use the following code sample as a reference:

mclSetCmdLineUserData

```
int run_main(int argc, const char **argv)
{
    int _retval;
    /* Generate and populate the path_to_component. */
    char path_to_component[(PATH_MAX*2)+1];
    separatePathName(argv[0],
        path_to_component, (PATH_MAX*2)+1);
    __MCC_tctf_component_data.path_to_component =
        path_to_component;
    if (!tctfInitialize()) {
        return -1;
    }

    size_t dataCount =
        mclSetCmdLineUserData(_mcr_inst, argc, argv);

    _retval = mclMain(_mcr_inst, argc, argv, "tctf", 0);
    if (_retval == 0 /* no error */)
        mclWaitForFiguresToDie(NULL);
    tctfTerminate();
    mclTerminateApplication();
    return _retval;
}
```

Purpose

Associate MATLAB data value with string key of MCR instance uniquely identified by `mcrID`

Syntax

```
extern "C"  
bool mclSetMCRUserData(  
    long mcrID,          // Integer MCR instance identifier  
    const char *key,    // Store user data under this key  
    mxArray *value      // User data value  
);
```

Description

This external C function associates a MATLAB data value with the string key of the MCR instance uniquely identified by `mcrID`. `mclSetMCRUserData` registers the `mxArray` value under the string key, which may later be used by `mclGetMCRInstanceData` to retrieve the data. `mclSetMCRUserData` is defined in the `MCLMCR` module and is only available to wrapper code in deployed applications. This function makes a shared copy of the input data and does not assume ownership of the `mxArray` value. You must call `mxDestroyArray` on `value` eventually in your application or memory leak problems may occur.

This command is part of the MCR User Data interface API. For information about this function, as well as complete examples of usage, see “Improving Data Access Using the MCR User Data Interface” on page 4-26.

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

Example

```
mxArray *value = mxCreateString("/usr/userdir/config.mat");  
if (!mclSetMCRUserData(mcrID, "ParallelConfigurationFile",  
    "/usr/userdir/config.mat") )  
{  
    fprintf(stderr, "Could not set PCTConfigFile MCR
```

mclSetMCRUserData

```
        user data. \n");
    return -3;
}
```

See Also

mclSetCmdLineUserData, mclGetMCRUserData

Purpose Closes down all MCR-internal application state. Called only once per process.

Syntax `bool mclTerminateApplication(void)`

Description Call this function once at the end of your program to close down all MCR-internal application state. 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.

Example At the start of your program, call `mclInitializeApplication` to ensure your library was properly initialized:

```
mclInitializeApplication(NULL,0);
if (!libmatrixInitialize()){
    fprintf(stderr,"could not initialize the library
properly\n");
    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;
```

mcITerminateApplication

See Also

“Initializing and Terminating Your Application with mcIInitializeApplication and mcITerminateApplication” on page 7-11

mcIInitializeApplication

Purpose Enables deployed applications to process Handle Graphics events, enabling figure windows to remain displayed.

Syntax `void mclWaitForFiguresToDie(HMCRINSTANCE inst)`

Description Calling `void mclWaitForFiguresToDie(HMCRINSTANCE inst)` enables the deployed application to process Handle Graphics events. If this function is not called, any figure windows initially displayed by the application will briefly appear and then the application will exit.

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 the input argument, an MCR instance, is null, the function monitors the figures of the current MCR.

This function can only be called after `<library>Initialize` has been called and before `<library>Terminate` has been called.

Note `WaitForFiguresToDie` 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.

Example

```
int run_main(int argc, const char** argv)
{
    int some_variable = 0;

    if (argc > 1)
        test_to_run = atoi(argv[1]);
    /* Initialize application */
    if( !mclInitializeApplication(NULL,0) )
    {
        fprintf(stderr, "Could not initialize the
```



```
        return(0);  
    }
```

See Also

`mclKillAllFigures`

“Blocking Execution of a Console Application with the `mclWaitForFiguresToDie` Method” on page 5-30

mcrinstaller

Purpose Displays version and location information for MCR installer(s) 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 the Chapter 10, “Reference Information” chapter for more information about the MCR installer.

Example `mcrinstaller`

The WIN32 MCR Installer, version 7.8, is:

```
X:\user\buildversion\matlab\toolbox\compiler\  
    deploy\win32\MCRInstaller.exe
```

MCR installers for other platforms are located in:

```
X:\user\buildversion\matlab\toolbox\compiler\  
    deploy\<ARCH>
```

<ARCH> is the value of COMPUTER('arch') on the
target machine.

For more information, read your local MCR Installer help.
Or see the online documentation at The MathWorks' web site.

ans =

```
X:\user\buildversion\matlab\toolbox\compiler\  
    deploy\win32\MCRInstaller.exe
```

mcrversion

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.

Example

```
mcrversion
ans =
    7
```

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 “Improving Data Access Using the MCR User Data Interface” on page 4-26.

Examples

```
setmcruserdata('ParallelConfigurationFile','config.mat')

mxArray *value = mxCreateString("/usr/userdir/config.mat");
if (!SetMCRUserData(mcrID, "ParallelConfigurationFile",
    "/usr/userdir/config.mat") )
{
    fprintf(stderr, "Could not set PCTConfigFile MCR user
        data.\n");
    return -3;
}
```

See Also getmcruserdata

setmcuserdata

MATLAB Compiler Quick Reference

- “Common Uses of MATLAB® Compiler” on page A-2
- “mcc” on page A-4

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

Bold entries in the Comment/Options column indicate default values.

Option	Description	Comment/Options
-a <i>filename</i>	Add <i>filename</i> to the CTF archive.	None
-b	Generate Excel compatible formula function.	Requires MATLAB Builder EX
-B <i>filename[:arg[,arg]]</i>	Replace -B <i>filename</i> on the <code>mcc</code> command line with the contents of <i>filename</i> .	The file should contain only <code>mcc</code> command-line options. These are MathWorks included options files: <ul style="list-style-type: none"> • -B <code>csharedlib:foo</code> — C shared library • -B <code>cpplib:foo</code> — C++ library
-c	Generate C wrapper code.	Equivalent to -T <code>codegen</code>
-C	Directs <code>mcc</code> to not embed the CTF archive in C/C++ and main/Winmain shared libraries and standalone binaries by default.	See “Overriding Default CTF Archive Embedding Using the MCR Component Cache” on page 5-20 for more information.
-d <i>directory</i>	Place output in specified folder.	None

Option	Description	Comment/Options
-e	Suppresses appearance of the MS-DOS Command Window when generating a standalone application.	<p>Use <code>-e</code> in place of the <code>-m</code> option. Available for Windows only. Use with <code>-R</code> option to generate error logging. Equivalent to <code>-W WinMain -T link:exe</code></p> <hr/> <p>Note This feature requires the application to successfully compile with a Microsoft Compiler (such as that offered with the free Microsoft Visual Studio Express).</p> <hr/>
-f filename	Use the specified options file, <code>filename</code> , when calling <code>mbuild</code> .	<code>mbuild -setup</code> is recommended.
-F project_name.prj	Use the specified project file as input to <code>mcc</code> .	When using <code>-F</code> , no other arguments can be invoked against <code>mcc</code> .
-g	Generate debugging information.	None
-G	Same as <code>-g</code>	None
-I directory	Add folder to search path for M-files.	MATLAB path is automatically included when running from MATLAB, but not when running from a DOS/UNIX shell.
-l	Macro to create a function library.	Equivalent to <code>-W lib -T link:lib</code>
-m	Macro to generate a C standalone application.	Equivalent to <code>-W main -T link:exe</code>
-M string	Pass string to <code>mbuild</code> .	Use to define compile-time options.
-N	Clear the path of all but a minimal, required set of folders.	None

Option	Description	Comment/Options
-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 <i>option</i>	Specify run-time options for MCR.	<i>option</i> = -nojvm -nodisplay -logfile -startmsg -completemsg <i>filename</i>
-S	Create Singleton MCR.	Requires MATLAB Builder NE
-v	Verbose; display compilation steps.	None
-w <i>option</i>	Display warning messages.	<i>option</i> = list <i>level</i> <i>level:string</i> where <i>level</i> = disable enable error
-W <i>type</i>	Control the generation of function wrappers.	<i>type</i> = main cpplib:<string> lib:<string> none com:comname,cname,version
-Y licensefile	Use licensefile when checking out a MATLAB Compiler license.	None
-z path	Specify path for library and include files.	None
-?	Display help message.	None

Error and Warning Messages

- “About Error and Warning Messages” on page B-2
- “Compile-Time Errors” on page B-3
- “Warning Messages” on page B-6
- “depfun Errors” on page B-9

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 M-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 = errno). 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: Could not find license file "filename". (*Windows only*) The license.dat file can not be found in *matlabroot\bin*.

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 M-file and cannot be compiled with the current Compiler. MATLAB Compiler cannot compile script M-files. To learn how to convert script M-files to function M-files, see “Converting Script M-Files to Function M-Files” on page 5-25.

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 Chapter 11, “Function Reference”, 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 Chapter 11, “Function Reference”, 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 Chapter 11, “Function Reference”, 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 B-6.

Error: Unrecognized option: -option. The option is not a valid option. See Chapter 11, “Function Reference”, 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 demo MATLAB Compiler license to create your standalone application, you can use:

```
mcc -w error:demo_license -mvg hello
```

To enable all warnings except those generated by the `save` command, use:

```
mcc -w enable -w disable:demo_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 Chapter 11, “Function Reference”.

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: M-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 M-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.

DEMO Compiler license. The generated application will expire 30 days from today, on date. (*demo_license*) This warning displays the date that the deployed application will expire. This warning is enabled.

depfun Errors

In this section...

“About depfun Errors” on page B-9

“MCR/Dispatcher Errors” on page B-9

“XML Parser Errors” on page B-9

“depfun-Produced Errors” on page B-10

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 The 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 The 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 The 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 The 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

- “Primitive Types” on page C-2
- “Utility Classes” on page C-3
- “mwString Class” on page C-4
- “mwException Class” on page C-20
- “mwException Class Functions” on page C-21
- “mwArray Class” on page C-29
- “mwArray Class Functions” on page C-33

Primitive Types

The `mxArray` API supports all primitive types that can be stored in a MATLAB array. This table lists all the types.

Type	Description	mxClassID
<code>mxChar</code>	Character type	<code>mxCHAR_CLASS</code>
<code>mxLogical</code>	Logical or Boolean type	<code>mxLOGICAL_CLASS</code>
<code>mxDouble</code>	Double-precision floating-point type	<code>mxDOUBLE_CLASS</code>
<code>mxSingle</code>	Single-precision floating-point type	<code>mxSINGLE_CLASS</code>
<code>mxInt8</code>	1-byte signed integer	<code>mxINT8_CLASS</code>
<code>mxUInt8</code>	1-byte unsigned integer	<code>mxUINT8_CLASS</code>
<code>mxInt16</code>	2-byte signed integer	<code>mxINT16_CLASS</code>
<code>mxUInt16</code>	2-byte unsigned integer	<code>mxUINT16_CLASS</code>
<code>mxInt32</code>	4-byte signed integer	<code>mxINT32_CLASS</code>
<code>mxUInt32</code>	4-byte unsigned integer	<code>mxUINT32_CLASS</code>
<code>mxInt64</code>	8-byte signed integer	<code>mxINT64_CLASS</code>
<code>mxUInt64</code>	8-byte unsigned integer	<code>mxUINT64_CLASS</code>

Utility Classes

The following are C++ utility classes:

- “mwString Class” on page C-4
- “mwException Class” on page C-20
- “mwArray Class” on page C-29

mwString Class

In this section...
“About mwString” on page C-4
“Constructors” on page C-4
“Methods” on page C-4
“Operators” on page C-4

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==(const mwString& str) const`
- `bool operator!=(const mwString& str) const`
- `bool operator<(const mwString& str) const`
- `bool operator<=(const mwString& str) const`

- `bool operator>(const mwString& str) const`
- `bool operator>=(const mwString& str) const`
- `friend std::ostream& operator<<(std::ostream& os, const mwString& str)`

mwString()

Purpose	Construct empty string
C++ Syntax	<pre>#include "mclcppclass.h" mwString str;</pre>
Arguments	None
Return Value	None
Description	Use this constructor to create an empty string.

Purpose	Construct new string and initialize strings data with supplied char buffer
C++ Syntax	<pre>#include "mclcppclass.h" mwString str("This is a string");</pre>
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.

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.

operator const char* () const

Purpose	Return pointer to internal buffer of string
C++ Syntax	<pre>#include "mclcppclass.h" mwString str("This is a string"); const char* pstr = (const char*)str;</pre>
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	<pre>#include "mclcppclass.h" mwString str("This is a string"); mwString new_str = str; // new_str contains a copy of // the data in str.</pre>
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	<pre>#include "mclcppclass.h" const char* pstr = "This is a string"; mwString str = pstr; // str contains copy of data in pstr.</pre>
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==(const mwString& str) const

Purpose

Test two mwStrings 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!=(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
Value** The result of the comparison.

Description Use this operator to test two strings for inequality.

bool operator<(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<=(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 Value The result of the comparison.

Description Use this operator to test two strings for order.

bool operator>(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>=(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	<pre>#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.</pre>
Arguments	str String to compare.
Return Value	The result of the comparison.
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

In this section...
“About mwException” on page C-20
“Constructors” on page C-20
“Methods” on page C-20
“Operators” on page C-20

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()`

Operators

- `mwException& operator=(const mwException& e)`
- `mwException& operator=(const std::exception& e)`

mwException Class Functions

The functions on the following pages are in the mwException class.

mwException()

Purpose	Construct new mwException with default error message
C++ Syntax	<pre>#include "mclcppclass.h" throw mwException();</pre>
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
Value**

None

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`.

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
Value** A reference to the invoking mwException.

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 C-29
“Constructors” on page C-29
“Methods” on page C-30
“Operators” on page C-31
“Static Methods” on page C-32

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 cplx = mxREAL)`
- `mwArray(mwSize num_dims, const mwSize* dims, mxClassID mxID, mxComplexity cplx = 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`

- `int GetHashCode() 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 mxArray Deserialize(const mxArray& 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)`

mwArray Class Functions

The functions on the following pages are in the mwArray class:

mwArray()

Purpose	Construct empty array of type mxDOUBLE_CLASS
C++ Syntax	<pre>#include "mclcppclass.h" mwArray a;</pre>
Return Value	None
Description	Use this constructor to create an empty array of type mxDOUBLE_CLASS.

Purpose	Construct empty array of specified type
C++ Syntax	<pre>#include "mclcppclass.h" mwArray a(mxDOUBLE_CLASS);</pre>
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	<pre>#include "mclcppclass.h" double x = 5.0; mwArray a(x); // Creates 1X1 double array with value 5.0</pre>
Arguments	re Scalar value to initialize array with.
Return Value	None
Description	<p>Use this constructor to create a real scalar array. <type> can be any of the following:</p> <ul style="list-style-type: none">• 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

mwArray(<type> re, <type> im)

- 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	<pre>#include "mclcppclass.h" mwArray a(2, 2, mxDOUBLE_CLASS); mwArray b = a.Clone();</pre>
Arguments	None
Return Value	New <code>mwArray</code> 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.

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 <code>mxUINT8_CLASS</code>
C++ Syntax	<pre>#include "mclcppclass.h" mwArray a(2, 2, mxDOUBLE_CLASS); mwArray s = a.Serialize();</pre>
Arguments	None
Return Value	New <code>mwArray</code> of type <code>mxUINT8_CLASS</code> containing the serialized data.
Description	Use this method to serialize an array into bytes. A 1-by-n numeric matrix of type <code>mxUINT8_CLASS</code> is returned containing the serialized data. The data can be deserialized back into the original representation by calling <code>mwArray::Deserialize()</code> .

Purpose	Return type of array
C++ Syntax	<pre>#include "mclcppclass.h" mwArray a(2, 2, mxDOUBLE_CLASS); mxClassID id = a.ClassID();// Should return mxDOUBLE_CLASS</pre>
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	<pre>#include "mclcppclass.h" mwArray a(2, 2, mxDOUBLE_CLASS); int size = a.ElementSize();// Should return sizeof(double)</pre>
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	<pre>#include "mclcppclass.h" mwArray a(2, 2, mxDOUBLE_CLASS); int n = a.NumberOfElements();// Should return 4</pre>
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	<pre>#include "mclcppclass.h" mwArray a(2, 2, mxDOUBLE_CLASS); int n = a.NumberOfDimensions();// Should return 2</pre>
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
Value** mwString containing the field name.

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	<pre>#include "mclcppclass.h" mwArray a; bool b = a.IsEmpty(); // Should return true</pre>
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	<pre>#include "mclcppclass.h" mwArray a(2, 2, mxDOUBLE_CLASS); bool b = a.IsSparse(); // Should return false</pre>
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	<pre>#include "mclcppclass.h" mwArray a(2, 2, mxDOUBLE_CLASS); bool b = a.IsNumeric(); // Should return true.</pre>
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 mxArray& arr) const

Purpose

Test two arrays for equality

C++ Syntax

```
#include "mclcppclass.h"
mxArray a(1, 1, mxDOUBLE_CLASS);
mxArray 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 Value

Boolean value indicating the equality of the two arrays.

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++
Syntax**

```
#include "mclcppclass.h"  
mwArray a(1, 1, mxDOUBLE_CLASS);  
int n = a.GetHashCode();
```

Arguments

None

**Return
Value**

An integer value representing a unique hash code for the array.

Description

This method constructs a unique hash value from 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 Value An mwString containing the string representation of the array.

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
Value** An mwArray containing the row indices.

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.

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

```
#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 $1 \leq \text{index} \leq \text{NumberOfElements}()$, for single subscript indexing. For multiple subscript indexing, the i th index has the valid range: $1 \leq \text{index}[i] \leq \text{GetDimensions}().\text{Get}(1, i)$. 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 $1 \leq \text{index} \leq \text{NumberOfElements}()$, for single subscript indexing. For multiple subscript indexing, the *i*th index has the valid range: $1 \leq \text{index}[i] \leq \text{GetDimensions}().\text{Get}(1, i)$. 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 sub-script indexing), in which case, the index list is used to access the specified element. The valid range for indices is $1 \leq \text{index} \leq \text{NumberOfElements}()$, for single subscript indexing. For multiple subscript indexing, the *i*th index has the valid range: $1 \leq \text{index}[i] \leq \text{GetDimensions}().\text{Get}(1, i)$. 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 ($N \times 2$), 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 ($N \times 2$), 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 mxArray& 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"
mxArray a(2, 2, mxDOUBLE_CLASS);
mxArray b(2, 2, mxINT16_CLASS);
mxArray 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
mxArray 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.SetData(data, 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

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', '\1', 'l', 'o', '\0'};
mxChar data_copy[6] ;
mwArray a(1, 6, mxCHAR_CLASS);
a.SetData(data, 6);
a.GetData(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.SetData(data, 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

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.SetData(data, 6);
a.GetData(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 $1 \leq \text{index} \leq \text{NumberOfElements}()$, for single subscript indexing. For multiple subscript indexing, the *i*th index has the valid range: $1 \leq \text{index}[i] \leq \text{GetDimensions}().\text{Get}(1, i)$. 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 $1 \leq \text{index} \leq \text{NumberOfElements}()$, for single subscript indexing. For multiple subscript indexing, the *i*th index has the valid range: $1 \leq \text{index}[i] \leq \text{GetDimensions}().\text{Get}(1, i)$. 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 mxArray Deserialize(const mxArray& arr)

Purpose Deserialize array that was serialized with `mxArray::Serialize`

C++ Syntax

```
#include "mclcppclass.h"
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
mxArray a(1,4,mxDOUBLE_CLASS);
a.SetData(rdata, 4);
mxArray b = a.Serialize();
a = mxArray::Deserialize(b); // a should contain same
                             // data as original
```

Arguments `arr`
mxArray that has been obtained by calling `mxArray::Serialize`.

Return Value A new mxArray containing the deserialized array.

Description Use this method to deserialize an array that has been serialized with `mxArray::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 mxArray, the behavior of this method is undefined.

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	<pre>#include "mclcppclass.h" double x = mxArray::GetEps();</pre>
Arguments	None
Return Value	The value of the MATLAB eps variable.
Description	Call <code>mxArray::GetEps</code> 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 <code>pinv</code> and <code>rank</code> functions use <code>eps</code> as a default tolerance.

Purpose	Get value of Inf (infinity)
C++ Syntax	<pre>#include "mclcppclass.h" double x = mxArray::GetInf();</pre>
Arguments	None
Return Value	The value of Inf (infinity) on your system.
Description	<p>Call <code>mxArray::GetInf</code> to return the value of the MATLAB internal <code>Inf</code> variable. <code>Inf</code> is a permanent variable representing IEEE arithmetic positive infinity. The value of <code>Inf</code> is built into the system; you cannot modify it.</p> <p>Operations that return <code>Inf</code> include</p> <ul style="list-style-type: none">• Division by 0. For example, <code>5/0</code> returns <code>Inf</code>.• Operations resulting in overflow. For example, <code>exp(10000)</code> returns <code>Inf</code> 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	<pre>#include "mclcppclass.h" bool x = mwArray::IsFinite(1.0); // Returns true</pre>
Arguments	Value to test for finiteness.
Return Value	Result of test.
Description	Call <code>mwArray::IsFinite</code> to determine whether or not a value is finite. A number is finite if it is greater than <code>-Inf</code> and less than <code>Inf</code> .

Purpose

Test if value is infinite and return true if value is infinite

C++**Syntax**

```
#include "mclcppclass.h"
bool x = mxArray::IsInf(1.0);           // Returns false
```

Arguments

Value to test for infinity.

Return Value

Result of test.

Description

Call `mxArray::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	<pre>#include "mclcppclass.h" bool x = mxArray::IsNaN(1.0); // Returns false</pre>
Arguments	Value to test for NaN.
Return Value	Result of test.
Description	<p>Call <code>mxArray::IsNaN</code> 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</p> <ul style="list-style-type: none">• <code>0.0/0.0</code>• <code>Inf - Inf</code> <p>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.</p>

A

- Accessibility
 - DLLs to add to path enabling 10-2
- addpath command 4-18
- Addressing
 - Extended
 - 2 GB Limit 7-3
- Advanced Encryption Standard (AES)
 - cryptosystem 3-2
- ANSI compiler
 - installing 2-5
- application
 - POSIX main 5-11
- application coding with
 - M-files and C/C++ files 6-10
 - M-files only 6-8
- Assistive technologies
 - DLLs to add to path enabling 10-2
- axes objects 9-5

B

- build process 3-3
- built-in function
 - calling from C/C++ 5-27
- bundle file 5-9

C

C

- interfacing to M-code 5-14
- shared library wrapper 5-12

C++

- interfacing to M-code 5-14
- library wrapper 5-13
- primitive types C-2
- utility classes C-3

C/C++ 3-5

- compilers

- supported on UNIX 2-3
- supported on Windows 2-3

- cache 12-39

- linux
 - mcc 12-39

- callback problems

- fixing 9-3

- callback strings

- searching M-files for 9-5

- code

- porting 4-16

- compilation path 4-17

- Compile 1-2

- Compiler

- license 10-8

- M-files 1-2

- MEX-files 1-2

- security 3-2

- compilers

- supported on UNIX 2-3

- supported on Windows 2-3

- compiling

- complete syntactic details 12-27

- shared library quick start 1-16

- Component Technology File (CTF) 3-2

- compopts.bat 2-12

- configuring

- C/C++ compiler 2-7

- using mbuild 2-7

- conflicting options

- resolving 5-3

- CTF (Component Technology File) archive 3-2

- determining files to include 4-17

- extracting without executing 4-17

- CTF Archive

- Controlling management and storage

- of. 5-20

- CTF file 3-2

D

- debugging 5-29
 - G option flag 12-31
- dependency analysis 3-4
- depfun 4-17
- deployed applications
 - troubleshooting 8-20
- deploying applications that call Java™ native libraries 5-29
- deploying components
 - from a network drive 4-35
- deploying GUIs with ActiveX controls 5-28
- deploying recompiled applications 4-24
- deploying to different platforms 4-16
- deployment 4-2
- deployprint function 12-10
- deploytool
 - quick start 1-10
- deploytool function 12-12
- directory
 - user profile 2-12
- DLL. See shared library 7-2
- double-clickable application
 - passing arguments 5-33

E

- encryption and compression 3-5
- error messages
 - compile-time B-2
 - Compiler B-2
 - depfun B-9
 - internal error B-2
 - warnings B-6
- export list 5-12
- extractCTF utility 4-17
- extracting
 - CTF archive without executing 4-17

F

- feval 12-5
 - using 5-22
- feval pragma 12-5
- .fig file
 - locating in deployed applications 5-29
- figure objects 9-5
- Figures
 - Keeping open by blocking execution of console application 5-30
 - Terminating by force 5-30
- file
 - license.dat 2-4
- file extensions 5-3
- files
 - bundle 5-9
 - license.dat 2-4
 - wrapper 1-5
- full path names
 - handling 5-8
- function
 - calling from command line 5-28
 - calling from M-code 5-14
 - comparison to scripts 5-25
 - unsupported in standalone mode 9-9
 - wrapper 5-11
- %#function 12-5
 - using 5-22
- function M-file 5-25
- functions
 - unsupported 9-9

G

- G option flag 12-31
- getmcuserdata function 12-13 12-15
- GUI
 - compiling with ActiveX controls 5-28
 - deploying
 - as shared library 5-33

displaying 5-33

H

Handle Graphics 9-5

hardcopy function 12-16

I

input/output files 3-6

 C shared library 3-7

 C++ shared library 3-9

 Macintosh 3-11

 standalone 3-6

interfacing

 M-code to C/C++ code 5-14

internal error B-2

isdeployed 8-16 12-17

ismcc 12-18

J

Java™ native libraries

 deploying applications that call 5-29

L

lcccomp.bat file 2-9

libraries

 overview 1-16

library

 shared C/C++ 7-2

 wrapper 5-13

<library>Initialize[WithHandlers] 12-19

<library>Terminate 12-20

license problem 2-4 8-16 10-8

license.dat file 2-4

licensing 10-8

limitations

 Windows compilers 2-11

linking

 stage of compilation 3-5

Load function 3-12

loadlibrary 7-16

locating

 .fig files in deployed applications 5-29

log

 installation process 10-12

M

-m option flag 6-9

-M option flag 12-33

M-file

 encrypting 3-2

 example

 main.m 6-8

 mrank.m 6-8

 function 5-25

 script 5-25

 searching for callback strings 9-5

Mac OS X

 using shared library 7-34

macros 5-5

main program 5-11

main wrapper 5-11

main.m 6-8

MAT files 3-12

MAT-files in deployed applications 5-28

MATLAB Compiler

 error messages B-2

 flags 5-2

 macro 5-5

 options 5-2

 summarized A-4

 syntax 12-27

 system requirements

 UNIX 2-2

 troubleshooting 8-16

 warning messages B-2

MATLAB® Compiler™

- installing
 - on Microsoft Windows 2-4
 - on UNIX 2-4
- MATLAB Compiler license 10-8
- MATLAB® Compiler™ Runtime (MCR) 3-2
- MATLAB data files 3-12
- matrixdriver
 - on Mac OS X 7-37
- mbuild 2-7
 - options 12-21
 - troubleshooting 8-14
- mcc 12-27
 - Compiler 2.3 options A-4
 - Overview 5-2
 - syntax 5-2
- mclGetLogFileName 12-41
- mclGetMCRUserData function 12-42
- mclInitializeApplication 12-43
- mclIsJVMEEnabled 12-46
- mclIsMCRInitialized 12-47
- mclIsNoDisplaySet 12-48
- mclKillAllFigures 5-30
- mclKillAllFigures function 12-49
- mclSetCmdLineUserData function 12-50
- mclSetMCRUserData function 12-53
- mclTerminateApplication 12-55
- mclWaitForFiguresToDie 5-30
- mclWaitForFiguresToDie function 12-57
- MCR
 - installing
 - options 10-9
- MCR (MATLAB® Compiler™ Runtime) 3-2
 - installing
 - multiple MCRs on same machine 4-24
 - on deployment machine 4-11
 - with MATLAB® on same machine 4-22
 - instance 7-11
 - options 7-11
- MCR Component Cache
 - How to use

- Overriding CTF embedding 5-20
- MCR initialization
 - start-up and completion user messages 4-33
- MCR Installer options
 - on Windows systems 10-10
- Memory Cleanup 7-38
- Memory Management 7-38
- Microsoft Visual C++ 2-3
- mlx interface function 7-30
- mrnk.m 6-8
- MSVC. See Microsoft Visual C++ 2-3
- msvc60compp.bat file 2-9
- msvc80compp.bat file 2-9
- msvc90compp.bat file 2-9
- MWComponentOptions 5-20
- MX_COMPAT_32_OFF 7-3
- mxArrays
 - Passing to shared libraries 7-38

N

- network drive
 - deploying from 4-35
- nocache 12-39

O

- objects (Handle Graphics) 9-5
- options 5-2
 - combining 5-2
 - Compiler 2.3 A-4
 - grouping 5-2
 - macros 5-5
 - resolving conflicting 5-3
 - specifying 5-2
- options file 2-12
 - changing 2-13
 - locating 2-12
 - modifying on
 - UNIX 2-14

- Windows 2-13
- UNIX 2-10
- Windows 2-8

P

- pass through
 - M option flag 12-33
- passing
 - arguments to standalone applications 5-31
- path
 - user interaction 4-17
 - I option 4-19
 - N and -p 4-19
- path names
 - handling full 5-8
- PLP (personal license password) 2-4
- porting code 4-16
- POSIX main application 5-11
- POSIX main wrapper 5-11
- pragma
 - feval 12-5
 - %#function 12-5
- primitive types C-2
- problem with license 2-4

Q

- quick start
 - compiling a shared library 1-16
- quotation marks
 - with mcc options 5-10
- quotes
 - with mcc options 5-10

R

- resolving
 - conflicting options 5-3
- rmpath 4-18

S

- Save function 3-12
- script file 5-25
 - including in deployed applications 5-26
- script M-file 5-25
 - converting to function M-files 5-25
- security 3-2
- setmcruserdata function 12-63
- shared library 7-4
 - calling structure 7-26
 - header file 5-12
 - using on Mac OS X 7-34
 - wrapper 5-12
- Shared Library
 - Creating 1-14
- Standalone application
 - Creating 1-14
- standalone application. See wrapper file 1-5
- standalone applications 6-1
 - passing arguments 5-31
 - restrictions on 9-9
 - restrictions on Compiler 2.3 9-9
- system requirements 2-2

T

- troubleshooting
 - Compiler problems 8-16
 - deployed applications 8-20
 - mbuild problems 8-14
 - missing functions 9-3

U

- uicontrol objects 9-5
- uimenu objects 9-5
- UNIX
 - options file 2-10
 - locating 2-12
 - supported compilers 2-3

- system requirements 2-2
- unsupported functions 9-9
- user messages
 - customizable 4-33
- user profile directory 2-12

V

- varargin 7-32
- varargout 7-32

W

- WaitForFiguresToDie 5-33
- warning message
 - Compiler B-2
- Windows

- options file 2-8
 - locating 2-12
- Windows compiler
 - limitations 2-11
- wrapper code generation 3-5
- wrapper file 1-5
- wrapper function 5-11
- wrappers
 - C shared library 5-12
 - C++ library 5-13
 - main 5-11

Z

- z option flag 12-38