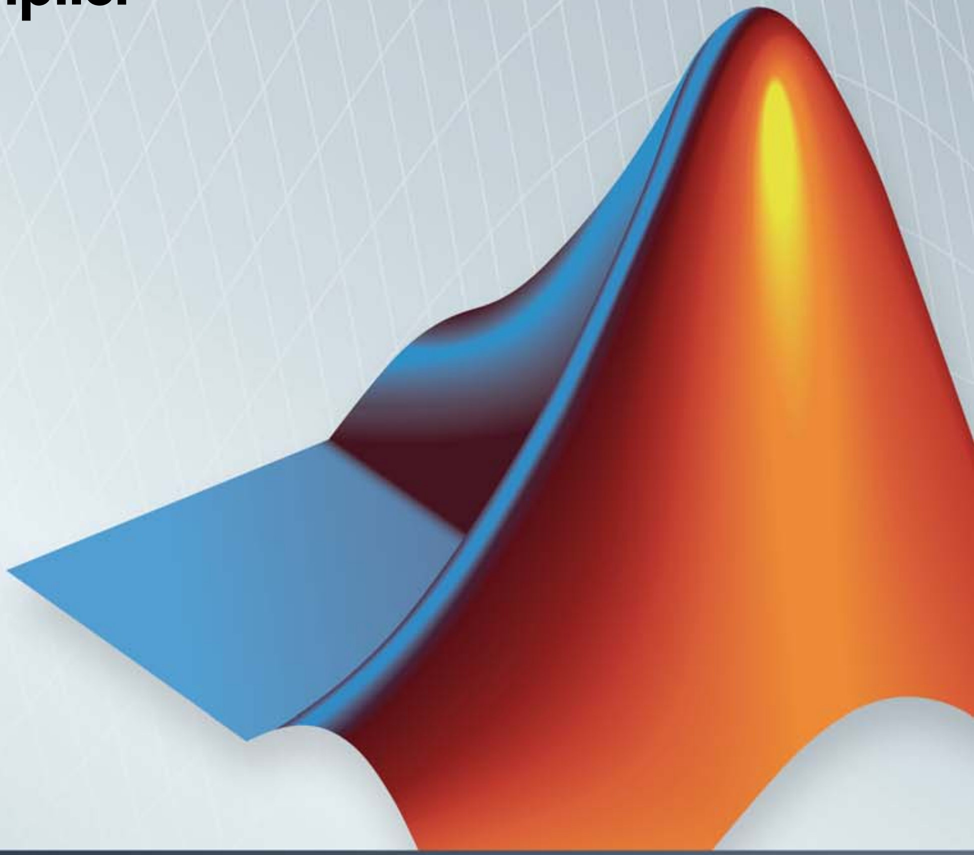
MATLAB® Compiler™

User's Guide

R2014**a**



MATLAB®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB® Compiler™ User's Guide

© COPYRIGHT 1995–2014 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

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)
March 2010	Online only	Revised for Version 4.13 (Release 2010a)
September 2010	Online only	Revised for Version 4.14 (Release 2010b)
April 2011	Online only	Revised for Version 4.15 (Release 2011a)
September 2011	Online only	Revised for Version 4.16 (Release 2011b)
March 2012	Online only	Revised for Version 4.17 (Release 2012a)
September 2012	Online only	Revised for Version 4.18 (Release 2012b)
March 2013	Online only	Revised for Version 4.18.1 (Release 2013a)
September 2013	Online only	Revised for Version 5.0 (Release 2013b)
March 2014	Online only	Revised for Version 5.1 (Release R2014a)

Contents

1 Getting Started

MATLAB Compiler Product Description **1-2**
- Key Features **1-2**

Appropriate Tasks for MATLAB Compiler and Builder Products **1-3**

MATLAB Application Deployment Products **1-5**

Roles in Deploying as a Standalone Application **1-7**

Roles in Deploying in a C/C++ Shared Library **1-8**

Roles in Deploying to MATLAB Production Server ... **1-10**

Create and Install a Standalone Application from MATLAB Code **1-12**
- Create a Standalone Application in MATLAB **1-12**
- Install a MATLAB Generated Standalone Application ... **1-16**

Create a C/C++ Shared Library from MATLAB Code .. **1-21**

Integrate a C/C++ Shared Library into an Application **1-26**

Create a Deployable Archive for MATLAB Production Server **1-32**

For More Information **1-37**

Installation and Configuration

2

Install an ANSI C or C++ Compiler **2-2**
Supported ANSI C and C++ Windows Compilers **2-2**
Supported ANSI C and C++ UNIX Compilers **2-2**
Common Installation Issues and Parameters **2-3**

Configuring Your Options File with mbuild **2-5**
What Is mbuild? **2-5**
Locating and Customizing the Options File **2-5**

Solving Installation Problems **2-8**

Deploying Standalone Applications

3

Compile a Standalone Application with the Standalone Compiler App **3-2**

Customize the Application's Run Time Behavior **3-8**

Compile a Standalone Application from the Command Line **3-9**
Execute Compiler Projects with deploytool **3-9**
Compile a Standalone Application with mcc **3-9**

Working with Standalone Applications and Arguments **3-11**
Overview **3-11**
Passing File Names, Numbers or Letters, Matrices, and MATLAB Variables **3-11**
Running Standalone Applications that Use Arguments ... **3-12**

Compile and Deploy Standalone Applications with the Parallel Computing Toolbox **3-15**

Standalone Applications with a Profile Passed at Run-Time 3-15
Standalone Applications with an Embedded Profile 3-16

Run a Mac OS X Application 3-18
Overview 3-18
Installing the Macintosh Application Launcher Preference Pane 3-18
Configuring the Installation Area 3-18
Launching the Application 3-21

Deploying C/C++ Shared Libraries

4

Compile a C/C++ Shared Library with the Library Compiler App 4-2

Compile a C/C++ Shared Library from the Command Line 4-7
Execute Compiler Projects with deploytool 4-7
Compile a Shared Library with mcc 4-7

What Are Wrapper Files? 4-9
C Library Wrapper 4-9
C++ Library Wrapper 4-9

Compiling Deployable Archives for MATLAB Production Server

5

State-Dependent Functions 5-2
Does My MATLAB Function Carry State? 5-2
Defensive Coding Practices 5-2
Techniques for Preserving State 5-3

Unsupported MATLAB Data Types for Client and Server Marshaling **5-5**

Compile a Deployable Archive with the Production Server Compiler App **5-6**

Compile a Deployable Archive from the Command Line **5-12**
  Execute Compiler Projects with deploytool **5-12**
  Compile a Deployable Archive with mcc **5-12**

6 Customizing a Compiler Project

Customize the Installer **6-2**
  Change the Application Icon **6-2**
  Add Application Information **6-3**
  Change the Splash Screen **6-4**
  Change the Installation Path **6-4**
  Change the Logo **6-5**
  Edit the Installation Notes **6-5**

Manage Required Files in a Compiler Project **6-6**
  Dependency Analysis **6-6**
  Using the Compiler Apps **6-6**
  Using mcc **6-7**

Specify Files to Install with the Application **6-8**

Manage Support Packages **6-9**

7 MATLAB Code Deployment

Application Deployment Products and the Compiler Apps **7-2**
What Is the Difference Between the Compiler Apps and the mcc Command Line? **7-2**
How Does MATLAB Compiler Software Build My Application? **7-2**
Dependency Analysis Function **7-5**
MEX-Files, DLLs, or Shared Libraries **7-6**
Component Technology File (CTF Archive) **7-6**

Write Deployable MATLAB Code **7-10**
Compiled Applications Do Not Process MATLAB Files at Runtime **7-10**
Do Not Rely on Changing Directory or Path to Control the Execution of MATLAB Files **7-11**
Use ismcc and isdeployed Functions To Execute Deployment-Specific Code Paths **7-12**
Gradually Refactor Applications That Depend on Noncompilable Functions **7-12**
Do Not Create or Use Nonconstant Static State Variables **7-12**
Get Proper Licenses for Toolbox Functionality You Want to Deploy **7-13**

How the Deployment Products Process MATLAB Function Signatures **7-14**
MATLAB Function Signature **7-14**
MATLAB Programming Basics **7-14**

Load MATLAB Libraries using loadlibrary **7-16**
Restrictions on Using MATLAB Function loadlibrary with MATLAB Compiler **7-17**

Use MATLAB Data Files (MAT Files) in Compiled Applications **7-18**
Explicitly Including MAT files Using the %#function Pragma **7-18**
Load and Save Functions **7-18**

MATLAB Objects 7-21

8 C and C++ Standalone Executable and Shared Library Creation

Input and Output Files 8-2
  Standalone Executable 8-2
  C Shared Library 8-3
  C++ Shared Library 8-5
  Macintosh 64 (Maci64) 8-7

Dependency Analysis Function and User Interaction with the Compilation Path 8-8
  addpath and rmpath in MATLAB 8-8
  Passing -I <directory> on the Command Line 8-8
  Passing -N and -p <directory> on the Command Line 8-8

9 Deployment Process

Overview 9-2
  Watch a Video 9-2

Deploying to Developers 9-3
  Procedure 9-3
  What Software Does a Developer Need? 9-3
  Ensuring Memory for Deployed Applications 9-5

Deploying to End Users 9-6
  Steps by the Developer to Deploy to End Users 9-6
  What Software Does the End User Need? 9-9
  Using Relative Paths with Project Files 9-12
  Porting Generated Code to a Different Platform 9-12

Extracting a CTF Archive Without Executing the Component **9-12**
Ensuring Memory for Deployed Applications **9-13**

Working with the MCR **9-14**
About the MATLAB Compiler Runtime (MCR) **9-14**
The MCR Installer **9-15**
Installing the MCR Non-Interactively **9-23**
Uninstalling the MCR **9-26**
MCR Startup Options **9-28**
Using the MCR User Data Interface **9-32**
Displaying MCR Initialization Start-Up and Completion Messages For Users **9-34**

Deploy Applications Created Using Parallel Computing Toolbox **9-37**
Compile and Deploy a Shared Library with the Parallel Computing Toolbox **9-37**

Deploying a Standalone Application on a Network Drive (Windows Only) **9-38**

MATLAB Compiler Deployment Messages **9-40**

Using MATLAB Compiler Generated DLLs in Windows Services **9-41**

Reserving Memory for Deployed Applications with MATLAB Memory Shielding **9-42**
What Is MATLAB Memory Shielding and When Should You Use It? **9-42**
Requirements for Using MATLAB Memory Shielding **9-43**
Invoking MATLAB Memory Shielding for Your Deployed Application **9-43**

10 Distributing Code to an End User

Share MATLAB Code Using the MATLAB Compiler Runtime (MCR) **10-2**
Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR) **10-2**

11 Compiler Commands

Command Overview **11-2**
Compiler Options **11-2**
Combining Options **11-2**
Conflicting Options on the Command Line **11-3**
Using File Extensions **11-3**
Interfacing MATLAB Code to C/C++ Code **11-4**

Simplify Compilation Using Macros **11-5**
Macro Options **11-5**
Working With Macro Options **11-5**

Invoke MATLAB Build Options **11-8**
Specifying Full Path Names to Build MATLAB Code **11-8**
Using Bundle Files to Build MATLAB Code **11-9**

MCR Component Cache and CTF Archive Embedding **11-11**
Overriding Default Behavior **11-12**
For More Information **11-13**

Explicitly Including a File for Compilation Using the %#function Pragma **11-14**
Using feval **11-14**
Using %#function **11-14**

Use the mxArray API to Work with MATLAB Types . . . **11-16**

Script Files **11-17**
  Converting Script MATLAB Files to Function MATLAB Files **11-17**
  Including Script Files in Deployed Applications **11-18**

Compiler Tips **11-20**
  Calling a Function from the Command Line **11-20**
  Using winopen in a Deployed Application **11-21**
  Using MAT-Files in Deployed Applications **11-21**
  Compiling a GUI That Contains an ActiveX Control **11-21**
  Debugging MATLAB Compiler Generated Executables ... **11-22**
  Deploying Applications That Call the Java Native Libraries **11-22**
  Locating .fig Files in Deployed Applications **11-22**
  Terminating Figures by Force In a Standalone Application **11-23**
  Passing Arguments to and from a Standalone Application **11-23**
  Using Graphical Applications in Shared Library Targets .. **11-25**
  Using the VER Function in a Compiled MATLAB Application **11-25**

12 Standalone Applications

Introduction **12-2**

Deploying Standalone Applications **12-3**
  Compiling the Application **12-3**
  Testing the Application **12-3**
  Deploying the Application **12-4**
  Running the Application **12-6**

13 Libraries

Introduction **13-2**

Addressing mwArrays Above the 2 GB Limit **13-3**

Integrate C Shared Libraries **13-4**
- C Shared Library Wrapper **13-4**
- C Shared Library Example **13-4**
- Calling a Shared Library **13-12**
- Using C Shared Libraries On a Mac OS X System **13-17**

Integrate C++ Shared Libraries **13-18**
- C++ Shared Library Wrapper **13-18**
- C++ Shared Library Example **13-18**

Call MATLAB Compiler API Functions (mcl*) from C/C++ Code **13-23**
- Functions in the Shared Library **13-23**
- Type of Application **13-23**
- Structure of Programs That Call Shared Libraries **13-25**
- Library Initialization and Termination Functions **13-26**
- Print and Error Handling Functions **13-27**
- Functions Generated from MATLAB Files **13-29**
- Retrieving MCR State Information While Using Shared Libraries **13-34**

About Memory Management and Cleanup **13-35**
- Overview **13-35**
- Passing mxArrays to Shared Libraries **13-35**

14 Troubleshooting

Introduction **14-2**

Common Issues **14-4**

Failure Points and Possible Solutions **14-5**
How to Use this Section **14-5**
Does the Failure Occur During Compilation? **14-5**
Does the Failure Occur When Testing Your Application? . . **14-9**
Does the Failure Occur When Deploying the Application to End Users? **14-13**

Troubleshooting mbuild **14-16**

MATLAB Compiler **14-18**

Deployed Applications **14-22**

15 Limitations and Restrictions

MATLAB Compiler Limitations **15-2**
Compiling MATLAB and Toolboxes **15-2**
Fixing Callback Problems: Missing Functions **15-3**
Finding Missing Functions in a MATLAB File **15-5**
Suppressing Warnings on the UNIX System **15-5**
Cannot Use Graphics with the -nojvm Option **15-6**
Cannot Create the Output File **15-6**
No MATLAB File Help for Compiled Functions **15-6**
No MCR Versioning on Mac OS X **15-7**
Older Neural Networks Not Deployable with MATLAB Compiler **15-7**
Restrictions on Calling PRINTDLG with Multiple Arguments in Compiled Mode **15-7**
Compiling a Function with WHICH Does Not Search Current Working Directory **15-8**
Restrictions on Using C++ SETDATA to Dynamically Resize an MWArray **15-8**

Licensing Terms and Restrictions on Compiled Applications **15-9**

MATLAB Functions That Cannot Be Compiled **15-10**

16 Reference Information

MCR Path Settings for Development and Testing **16-2**
Overview **16-2**
Path for Java Development on All Platforms **16-2**
Path Modifications Required for Accessibility **16-2**
Windows Settings for Development and Testing **16-3**
Linux Settings for Development and Testing **16-3**
Mac Settings for Development and Testing **16-3**

MCR Path Settings for Run-time Deployment **16-4**
General Path Guidelines **16-4**
Path for Java Applications on All Platforms **16-4**
Windows Path for Run-Time Deployment **16-4**
Linux Paths for Run-Time Deployment **16-5**
Mac Paths for Run-Time Deployment **16-5**

MATLAB Compiler Licensing **16-6**
Using MATLAB Compiler Licenses for Development **16-6**

Deployment Product Terms **16-8**

17 Functions — Alphabetical List

A MATLAB Compiler Quick Reference

Common Uses of MATLAB Compiler **A-2**

Create a Standalone Application A-2
Create a Library A-2

mcc Command Arguments Listed Alphabetically A-4

mcc Command Line Arguments Grouped by Task A-8

B Using MATLAB Compiler on Mac or Linux

Overview B-2

Installing MATLAB Compiler on Mac or Linux B-3
Installing MATLAB Compiler B-3
Custom Configuring Your Options File B-3
Install Apple Xcode from DVD on Maci64 .... B-3

Writing Applications for Mac or Linux B-4
Objective-C/C++ Applications for Apple's Cocoa API B-4
Where's the Example Code? B-4
Preparing Your Apple Xcode Development Environment .. B-4
Build and Run the Sierpinski Application B-5
Running the Sierpinski Application B-7

Building Your Application on Mac or Linux B-10
Compiling Your Application with the Compiler Apps B-10
Compiling Your Application with the Command Line B-10

Testing Your Application on Mac or Linux B-11

Set MCR Paths on Mac or Linux with Scripts B-12
Solving Problems Related to Setting MCR Paths on Mac or Linux B-12

C Error and Warning Messages

About Error and Warning Messages **C-2**

Compile-Time Errors **C-3**

Warning Messages **C-7**

Dependency Analysis Errors **C-10**
- MCR/Dispatcher Errors **C-10**
- XML Parser Errors **C-10**

D C++ Utility Library Reference

Data Conversion Restrictions for the C++ MWArray API **D-2**

Primitive Types **D-3**

C++ Utility Classes **D-4**

Getting Started

- "MATLAB® Compiler™ Product Description" on page 1-2
- "Appropriate Tasks for MATLAB® Compiler™ and Builder Products" on page 1-3
- "MATLAB Application Deployment Products " on page 1-5
- "Roles in Deploying as a Standalone Application" on page 1-7
- "Roles in Deploying in a C/C++ Shared Library" on page 1-8
- "Roles in Deploying to MATLAB® Production Server™" on page 1-10
- "Create and Install a Standalone Application from MATLAB Code" on page 1-12
- "Create a C/C++ Shared Library from MATLAB Code" on page 1-21
- "Integrate a C/C++ Shared Library into an Application" on page 1-26
- "Create a Deployable Archive for MATLAB® Production Server™" on page 1-32
- "For More Information" on page 1-37

MATLAB Compiler Product Description

Build standalone applications and software components from MATLAB® programs

MATLAB Compiler™ lets you share MATLAB programs as standalone applications or shared libraries for integration with common programming languages. Applications and libraries created with MATLAB Compiler use the MATLAB Compiler Runtime (MCR), which enables royalty-free deployment to users who do not have MATLAB. You can package the MATLAB Compiler Runtime with the application or have your users download it during installation

Learn more about MATLAB Compiler support for MATLAB and toolboxes.

Key Features

- Packaging of your MATLAB programs as standalone applications or shared libraries
- Royalty-free distribution of applications to users who do not have MATLAB
- Integration of MATLAB programs into Java®, Microsoft® .NET, and Excel®applications using MATLAB builder products
- Large-scale deployment of MATLAB programs using MATLAB Production Server™
- Encryption of MATLAB code to protect your intellectual property

Appropriate Tasks for MATLAB Compiler and Builder Products

MATLAB Compiler compiles MATLAB code into standalone applications, libraries that can be integrated into other applications, or into deployable archives for use with MATLAB Production Server. By default, MATLAB Compiler can generate standalone applications, C/C++ shared libraries, and deployable archives for use with MATLAB Production Server. Additional builders are available for Java, .NET, and Microsoft® Excel®.

While MATLAB Compiler lets you run your MATLAB application outside the MATLAB environment, it is not appropriate for all external tasks you may want to perform. Some tasks require either the MATLAB Coder™ product or MATLAB external interfaces. Use the following table to determine if MATLAB Compiler and builder products are appropriate to your needs.

MATLAB Compiler Task Matrix

Task	MATLAB Compiler and Builders	MATLAB Coder	MATLAB External Interfaces
Package MATLAB applications for deployment to users who do not have MATLAB	■		
Package MATLAB applications for deployment to MATLAB Production Server	■		
Build non-MATLAB applications that include MATLAB functions	■		
Generate readable, efficient, and embeddable C code from MATLAB code		■	

MATLAB Compiler Task Matrix (Continued)

Task	MATLAB Compiler and Builders	MATLAB Coder	MATLAB External Interfaces
Generate MEX functions from MATLAB code for rapid prototyping and verification of generated C code within MATLAB		■	
Integrate MATLAB code into Simulink®		■	
Speed up fixed-point MATLAB code		■	
Generate hardware description language (HDL) from MATLAB code		■	
Integrate custom C code into MATLAB with MEX files			■
Call MATLAB from C and Fortran programs			■

For information on MATLAB Coder see “MATLAB Coder”.

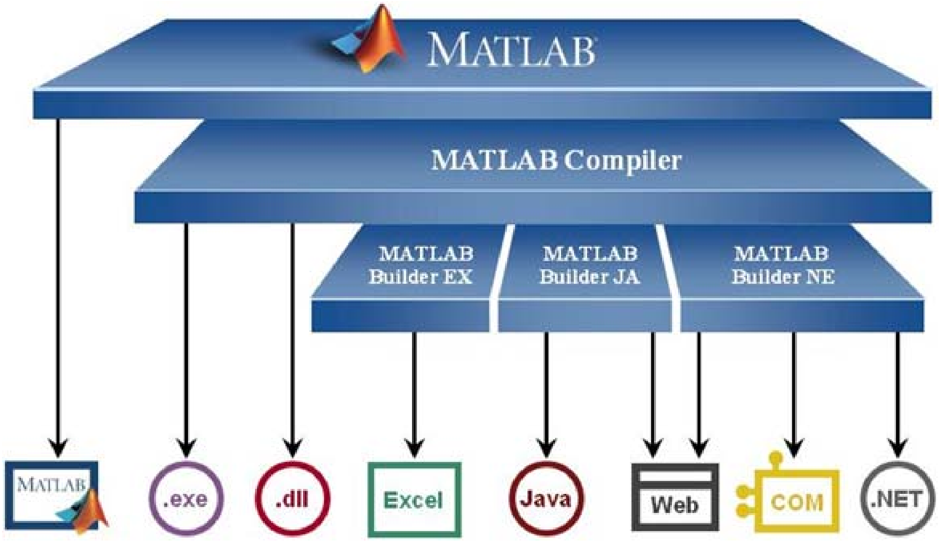
For information on MATLAB external interfaces see “External Code Integration”.

MATLAB Application Deployment Products

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

MATLAB Suite of Application Deployment Products

Product	Target	Standalone Applications	Function Libraries	Graphical Apps	Web Apps	WebFigures
MATLAB Compiler	Standalone applications and C and C++ shared libraries	Yes	Yes	Yes	No	No
MATLAB Builder™ NE	C# .NET components Visual Basic COM components	No	Yes	Yes	Yes	Yes
MATLAB Builder JA	Java components	No	Yes	Yes	Yes	Yes
MATLAB Builder EX	Microsoft Excel add-ins	No	Yes	Yes	No	No


MATLAB
MATLAB Compiler
MATLAB Builder EX
MATLAB Builder JA
MATLAB Builder NE
MATLAB
.exe
.dll
Excel
Java
Web
COM
.NET

MATLAB® Application Deployment Products

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

Roles in Deploying as a Standalone Application

Deploying MATLAB functionality as a standalone application is a multistep process that may involve one or more team members. Each step requires that you perform a specific role, as shown in Standalone Application Deployment Roles on page 1-7.

Standalone Application Deployment Roles

Role	Knowledge Base	Responsibilities
MATLAB programmer	• MATLAB expert • No IT experience	• Develop functions and implements them in MATLAB. • Create a standalone applications that can be used without being integrated into any 3rd party applications.
IT professional	• Little to no MATLAB experience • Moderate IT experience • Familiarity with IT systems	• Ensure that systems using the generated applications meet the required specifications. • Install any required software on target machines. • Install the generated applications on target machines.

Roles in Deploying in a C/C++ Shared Library

Deploying MATLAB functionality through C/C++ applications is a multistep process that may involve one or more team members. Each step requires that you perform a specific role, as shown in Shared Library Deployment Roles on page 1-8.

Shared Library Deployment Roles

Role	Knowledge Base	Responsibilities
MATLAB programmer	• MATLAB expert • Little to no C/C++ knowledge • No IT experience	• Develop functions and implements them in MATLAB. • Create shared libraries that are delivered to a C/C++ developer for integration into custom applications.
C/C++ developer	• Little to no MATLAB experience • Some knowledge of IT systems • C/C++ expert	• Develop C/C++ applications using MATLAB generated shared libraries. • Test C/C++ applications. • Package C/C++ applications for distribution.
IT professional	• Little to no MATLAB experience • Moderate IT experience • Familiarity with IT systems	• Ensure that systems using MATLAB application have the required specifications. • Install any required software on target machines.

Shared Library Deployment Roles (Continued)

Role	Knowledge Base	Responsibilities
		• Install MATLAB applications on target machines.

Roles in Deploying to MATLAB Production Server

Deploying MATLAB functionality using MATLAB Production Server is a multistep process that might involve one or more team members. Each step requires fulfilling specific roles, as shown in MATLAB® Production Server™ Deployment Roles on page 1-10 .

MATLAB Production Server Deployment Roles

Role	Knowledge Base	Responsibilities
MATLAB programmer	• MATLAB expert • Little to no software development experience • Little to no IT experience	• Develop functions and implements them in MATLAB. • Create deployable archives that run in MATLAB Production Server instances.
Application developer	• Little to no MATLAB experience • Some knowledge of IT systems • Familiarity with developing applications using a client/server architecture	• Develop applications using one of the MATLAB Production Server client APIs. • Test applications. • Package applications for distribution.

MATLAB Production Server Deployment Roles (Continued)

Role	Knowledge Base	Responsibilities
Server administrator	• Little to no MATLAB experience • Moderate IT experience • Familiarity with IT systems	• Ensure that systems running MATLAB Production Server instances have the required specifications. • Install MATLAB Production Server instances. • Tune MATLAB Production Server instances. • Install compiled MATLAB applications into MATLAB Production Server instances. • Monitor MATLAB Production Server instances.
Application installer	• Little to no MATLAB experience • Moderate IT experience • Familiarity with IT systems	• Ensure that systems using MATLAB Production Server client applications meet the required specifications. • Install any required software on target machines. • Install MATLAB Production Server client applications on target machines.

Create and Install a Standalone Application from MATLAB Code

In this section...

"Create a Standalone Application in MATLAB" on page 1-12

"Install a MATLAB Generated Standalone Application" on page 1-16

Create a Standalone Application in MATLAB

This example shows how to generate a standalone application from MATLAB. You package a pre-written function that prints a magic square to a computer's command console. The MATLAB Compiler produces an installer that installs the standalone application and all of the required dependencies on a target system. The target system does not require a licensed copy of MATLAB.

1. In MATLAB, examine the MATLAB code that you want deployed as a standalone application.
   - a Open `magicsquare.m`.

     ```
     function magicsquare(n)

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

   - b At the MATLAB command prompt, enter `magicsquare(5)`.

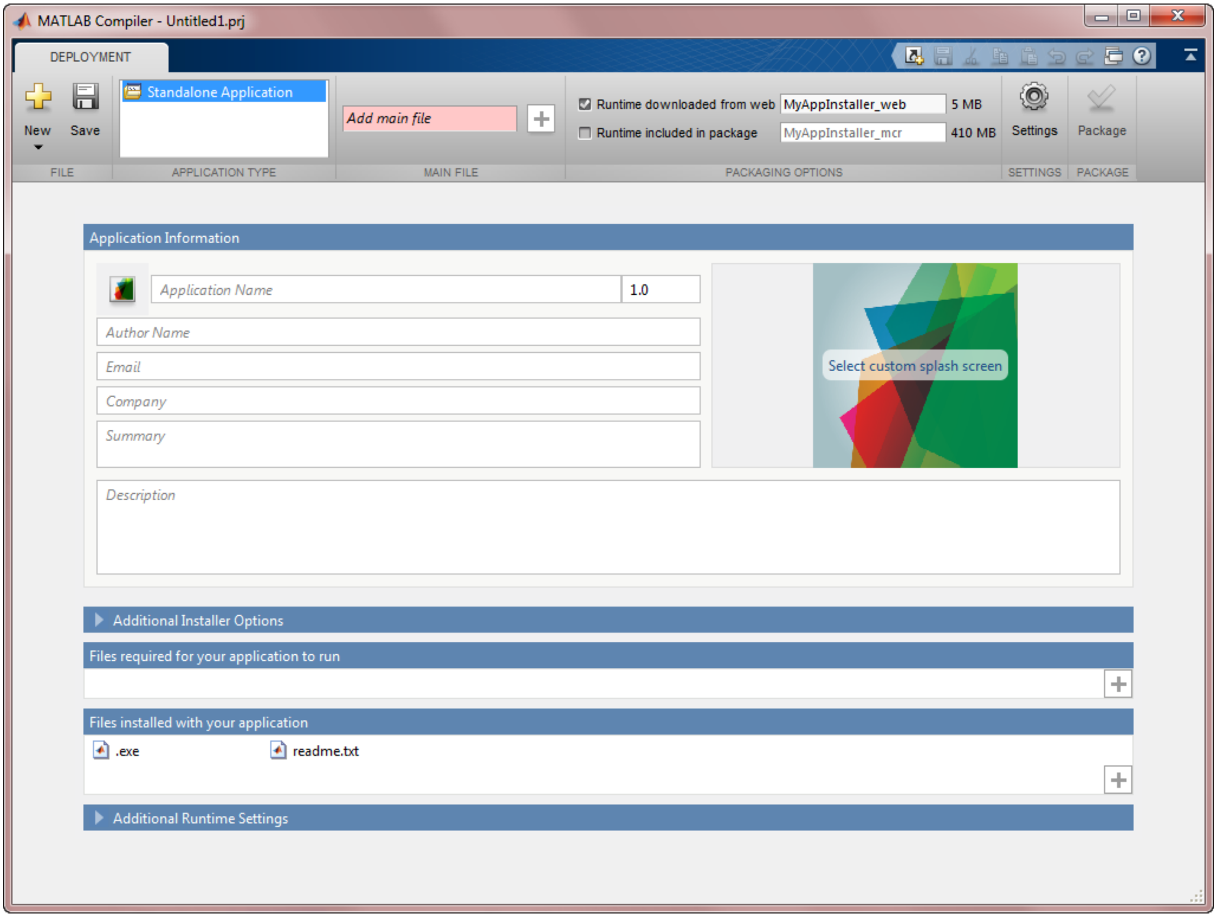
     The output appears as follows:

     ```
         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
     ```

2. Open the **Application Compiler**.
   - a On the toolstrip select the **Apps** tab.

b Click the arrow at the far right of the tab to open the apps gallery.

c Click **Application Compiler** to open the **MATLAB Compiler** project window.



MATLAB Compiler - Untitled1.prj

Standalone Application

☑ Runtime downloaded from web MyAppInstaller_web 5 MB

☐ Runtime included in package MyAppInstaller_mcr 410 MB

Files installed with your application: .exe, readme.txt

DEPLOYMENT
New
Save
Add main file
Settings
Package
FILE
APPLICATION TYPE
MAIN FILE
PACKAGING OPTIONS
SETTINGS
PACKAGE
Application Information
Application Name
1.0
Author Name
Email
Company
Summary
Select custom splash screen
Description
Additional Installer Options
Files required for your application to run
Files installed with your application
Additional Runtime Settings

3 Specify the main file of the MATLAB application you want to deploy.

a In the **Main File** section of the toolstrip, click the plus button.

Note If the **Main File** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

- **b** In the file explorer that opens, locate and select the `magicsquare.m` file.

  `magicsquare.m` is located in `*matlabroot*\extern\examples\compiler`.

- **c** Click **Open** to select the file and close the file explorer.

  magicsquare.m is added to the list of main files and the plus button will be replaced by a minus button.

4. In the **Packaging Options** section of the toolstrip, verify that the **Runtime downloaded from web** check box is selected.

   Note If the **Packaging Options** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

   This option creates an application installer that automatically downloadsthe MATLAB Compiler Runtime (MCR) and installs it along with the deployed MATLAB application.

5. Explore the main body of the **MATLAB Compiler** project window.

   The project window is divided into the following areas:

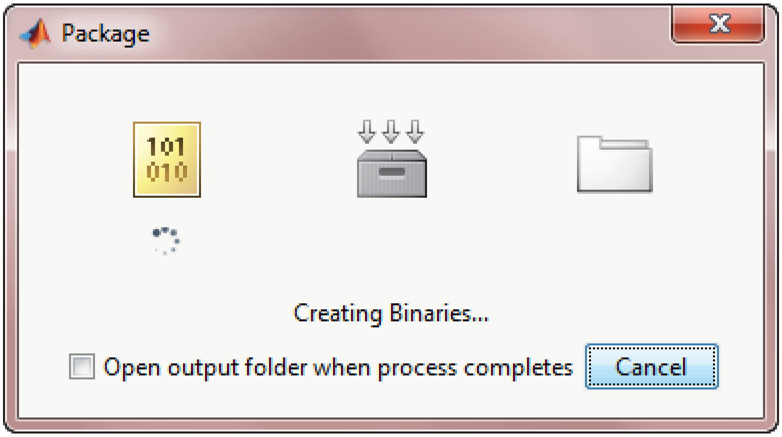
   - **Application Information** — Editable information about the deployed application. This information is used by the generated installer to populate the installed application's metadata. See "Customize the Installer" on page 6-2.
   - **Additional Installer Options** — The default installation path for the generated installer. See "Customize the Installer" on page 6-2.
   - **Files required for your application** — Additional files required by the generated application. These files will be included in the generated application installer. See "Manage Required Files in a Compiler Project" on page 6-6.

- **Files installed with your application** — Files that are installed with your application. These files include:
  - Generated `readme.txt`
  - Generated executable for the target platform

  See “Specify Files to Install with the Application” on page 6-8
- **Additional Runtime Settings** — Platform specific options for controlling the generated executable. See “Customize the Application’s Run Time Behavior” on page 3-8.

6 Click **Package**.

The Package window opens while the application is being generated.



7 Select the **Open output folder when process completes** check box.

When the deployment process is complete a file explorer opens and displays the generated output.

It should contain:

- `for_redistribution` — A folder containing the installer to distribute the application
- `for_testing` — A folder containing the raw files generated by the compiler

- `for_redistribution_files_only` — A folder containing only the files needed to redistribute the application
- `PackagingLog.txt` — A log file generated by the compiler.

8 Click **Close** on the Package window.

Install a MATLAB Generated Standalone Application

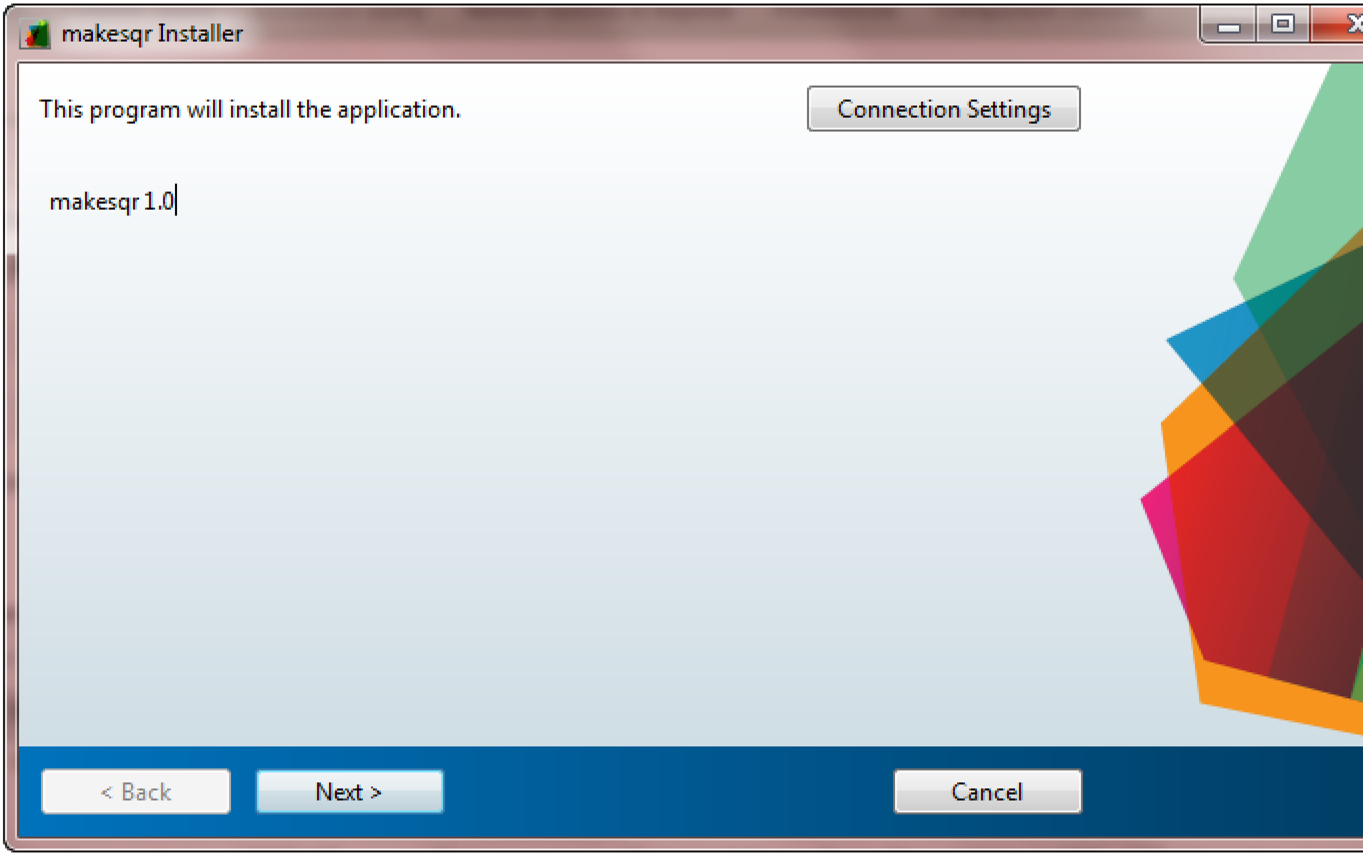
This example shows how to install the standalone application you created in "Create a Standalone Application in MATLAB" on page 1-12.

1 Locate the `MyAppInstaller_web` executable in the `for_redistribution` folder created by the MATLAB Compiler.

Note The file extension varies depend on which platform the installer was generated.

---

2 Double click the installer to run it.


makesqr Installer
Connection Settings
< Back
Next >
Cancel

This program will install the application.

makesqr 1.0

Note Any information entered in the MATLAB Compiler project window's **Application Information** appears on the this window.

---

3 If you connect to the internet using a proxy server, enter the server's settings.

  a Click **Connection Settings**.

  b Enter the proxy server settings in the provided window.

  c Click **OK**.

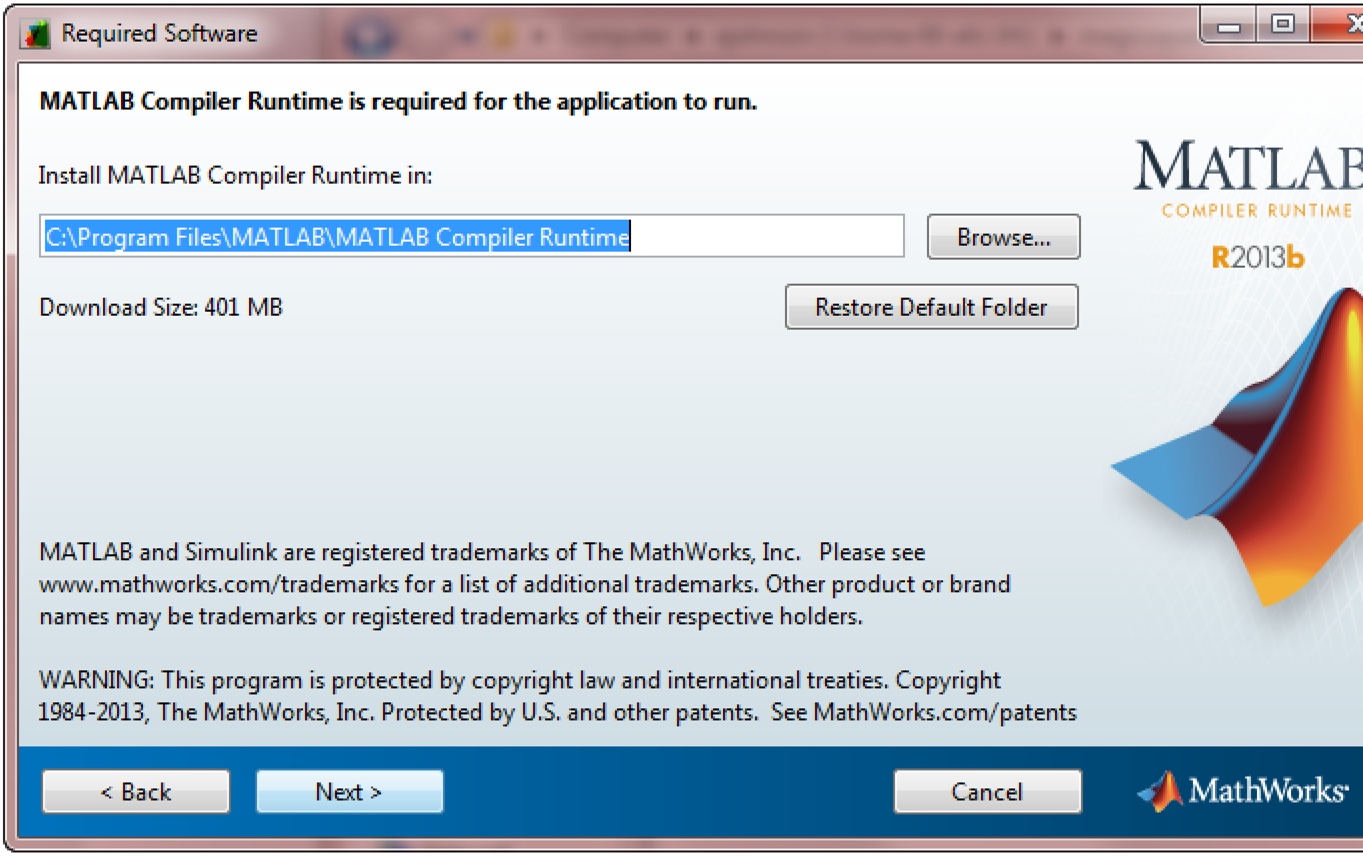
4 Click **Next** to advance to the Installation Options page.

Note On Linux® and Mac OS X you will not have the option of adding a desktop shortcut.

---

5 Click **Next** to advance to the Required Software page.

If asked about creating the destination folder, click **Yes**.



Required Software

MATLAB Compiler Runtime is required for the application to run.

Install MATLAB Compiler Runtime in:

C:\Program Files\MATLAB\MATLAB Compiler Runtime

Browse...

Download Size: 401 MB

Restore Default Folder

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. Please 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.

WARNING: This program is protected by copyright law and international treaties. Copyright 1984-2013, The MathWorks, Inc. Protected by U.S. and other patents. See MathWorks.com/patents

< Back
Next >
Cancel
MATLAB
COMPILER RUNTIME
R2013b
MathWorks

Note If you already have the correct version of the MATLAB Compiler Runtime (MCR) installed on the system, this page will have a message indicating that you do not have to install the MCR.

If you receive this message, skip to step 10 on page 1-19.

---

6. Click **Next** to advance to the License Agreement page.

   If asked about creating the destination folder, click **Yes**.

7. Read the license agreement.
8. Check **Yes** to accept the license.
9. Click **Next** to advance to the Confirmation page.
10. Click **Install**.

    The installer installs the MATLAB generated application. If needed, it also downloads and installs the MCR.

11. Click **Finish**.
12. Run your standalone application.
    - **a** Open a terminal window.
    - **b** Navigate to the folder into which you installed the application.

      If you accepted the default settings it will be located in one of the following location:

Windows®	`C:\Program Files\magicsquare`
Mac OS X	`/Applications/magicsquare`
Linux	`/usr/magicsquare`

    - **c** Run the application using the one of the following commands:

Windows	`application\magicsquare 5`
Mac OS X	
Linux	`./magicsquare 5`

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

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

Create a C/C++ Shared Library from MATLAB Code

This example shows how to create a C/C++ shared library using a MATLAB function. You can then hand the generated shared library off to the C/C++ developer who is responsible for integrating it into an application.

To create a C++ shared library:

1. In MATLAB, examine the MATLAB code that you want to deploy as a shared library.
   - **a** Open `addmatrix.m`.

     ```
     function a = addmatrix(a1, a2)

     a = a1 + a2;
     ```

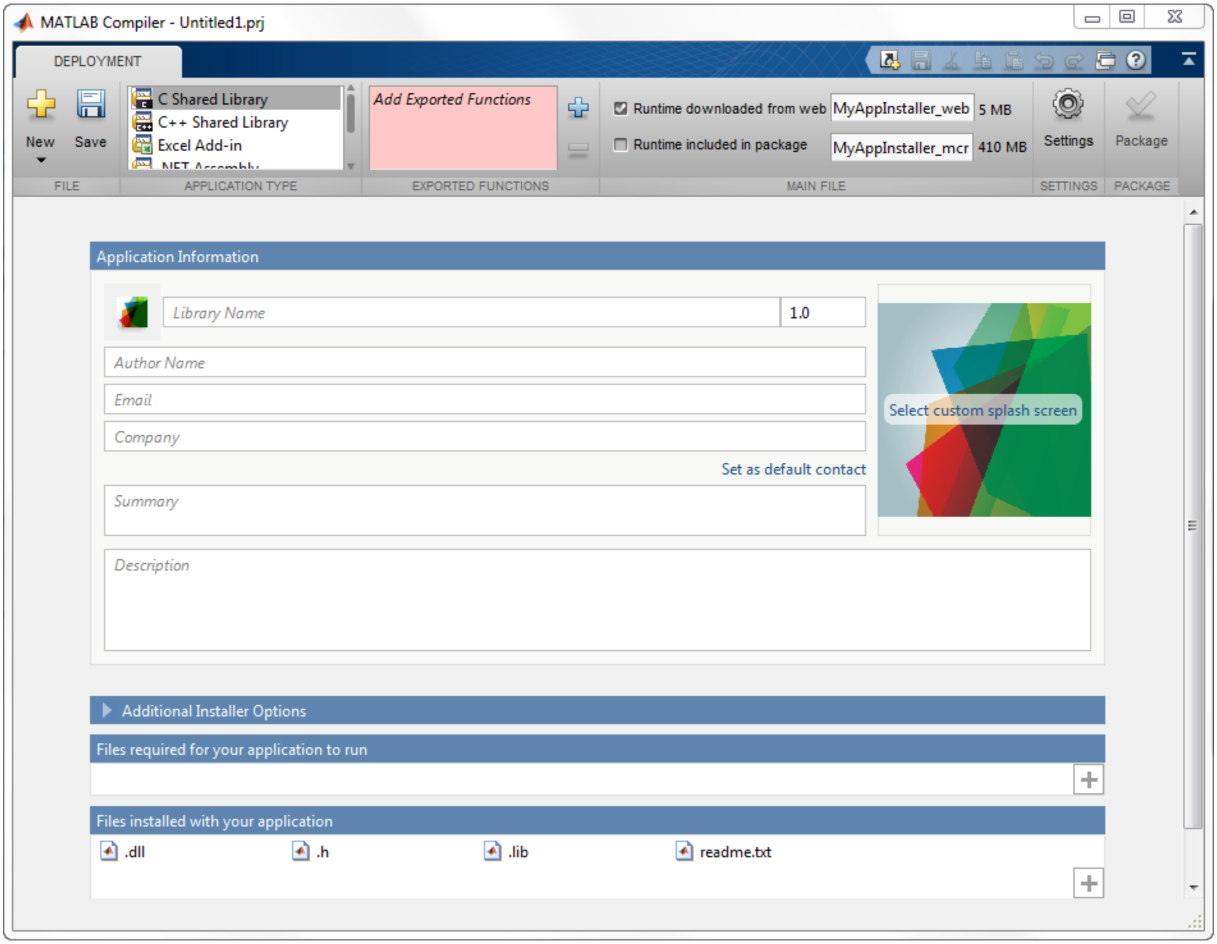
   - **b** At the MATLAB command prompt, enter `addmatrix(1,2)`.

     The output appears as follows:

     ```
     ans =

            3
     ```

2. Open the **Library Compiler**.
   - **a** On the toolstrip, select the **Apps** tab.
   - **b** Click the arrow at the far right of the tab to open the apps gallery.
   - **c** Click **Library Compiler** to open the **MATLAB Compiler** project window.


MATLAB Compiler - Untitled1.prj
DEPLOYMENT
New
Save
C Shared Library
C++ Shared Library
Excel Add-in
.NET Assembly
Add Exported Functions
Settings
Package
FILE
APPLICATION TYPE
EXPORTED FUNCTIONS
MAIN FILE
SETTINGS
PACKAGE
Application Information
Library Name
Author Name
Email
Company
Set as default contact
Summary
Description
Select custom splash screen
Additional Installer Options
Files required for your application to run
Files installed with your application
.dll
.h
.lib
readme.txt

☑ Runtime downloaded from web MyAppInstaller_web 5 MB

☐ Runtime included in package MyAppInstaller_mcr 410 MB

Version: 1.0

3 In the **Application Type** section of the toolstrip, select **C++ Shared Library** from the list.

Note If the **Application Type** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

---

4 Specify the MATLAB functions you want to deploy.

   a. In the **Exported Functions** section of the toolstrip, click the plus button.

   Note If the **Exported Functions** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

   ---

   b. In the file explorer that opens, locate and select the `addmatrix.m` file.

      `addmatrix.m` is located in *`matlabroot`*`\extern\examples\compiler`.

   c. Click **Open** to select the file and close the file explorer.

      addmatrix.m is added to the list of files and a minus button appears under the plus button.

5. In the **Packaging Options** section of the toolstrip, verify that the **Runtime downloaded from web** check box is selected.

   Note If the **Packaging Options** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

   This option creates an application installer that automatically downloads the MATLAB Compiler Runtime (MCR) and installs it along with the deployed shared library.

6. Explore the main body of the **MATLAB Compiler** project window.

   The project window is divided into the following areas:

   - **Application Information** — Editable information about the deployed application. This information is used by the generated installer to populate the installed application's metadata. See "Customize the Installer" on page 6-2.
   - **Additional Installer Options** — The default installation path for the generated installer. See "Customize the Installer" on page 6-2.
   - **Files required for your application** — Additional files required by the generated application. These files will be included in the generated

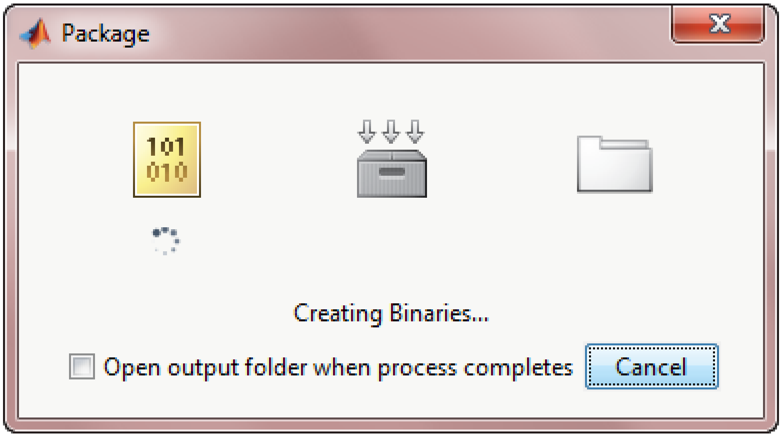
application installer. See “Manage Required Files in a Compiler Project” on page 6-6.

- **Files installed with your application** — Files that are installed with your application. These files include:
    - `readme.txt`
    - `.h` file
    - `.dll` file
    - `.lib` file

  See “Specify Files to Install with the Application” on page 6-8.

7 Click **Package**.

The Package window opens while the library is being generated.



Creating Binaries...

☐ Open output folder when process completes

8 Select the **Open output folder when process completes** check box.

When the deployment process is complete, a file explorer opens and displays the generated output.

It should contain:

- `for_redistribution` — A folder containing the installer to distribute the library
- `for_testing` — A folder containing the raw files generated by the compiler
- `for_redistribution_files_only` — A folder containing only the files needed to redistribute the library
- `PackagingLog.txt` — A log file generated by the compiler.

9. Click **Close** on the Package window.

10. Verify the contents of the generated output:

    - `for_redistribution` — A folder containing the installer to distribute the standalone application
    - `for_testing` — A folder containing the raw files generated by the compiler
    - `for_redistribution_files_only` — A folder containing only the files needed to redistribute the application
    - `PackagingLog.txt` — A log file generated by the compiler.

To follow up on this example:

- Try creating a shared library that consists of more than one function.
- Try “Integrate a C/C++ Shared Library into an Application” on page 1-26

Integrate a C/C++ Shared Library into an Application

This example shows how to call a C++ shared library built with MATLAB Compiler from a C++ application.

To create a C++ application that calls a MATLAB generated shared library:

1. Install the MATLAB Compiler Runtime (MCR) and shared library files in one of the following ways.

   - Running the installer generated by MATLAB. It is located in the `for_redistribution` folder of the deployment project.

     Doing so automatically installs the MCR from the Web and places the shared library folders onto your computer.

   - Manually installing the MCR and the generated shared libraries onto you development system.

     You can download the MCR installer from http://www.mathworks.com/products/compiler/mcr. The generated shared libraries and support files are located in the MATLAB deployment project's `for_testing` folder.

2. In the folder containing the generated shared libraries, create a new file called `addmatrix.cpp`.

3. Using a text editor, open `addmatrix.cpp`.

4. Place the following as the first line in the file.

   ```
   #include "addmatrix.h"
   ```

   Inserting this statement includes the generated header file for the MATLAB shared library.

5. Add the following `main()` function.

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

```
}
```

The `main()` function does the following:

- `mclmcrInitialize()` initializes the MCR so that it is ready to load the MATLAB code required to execute the deployed function.
- `mclRunMain()` creates a new thread and runs the MATLAB generated code in it.

6 Add a `run_main()` function to the application.

```
int run_main(int argc, char **argv)
{
}
```

7 Add the following code to the top of the `run_main()` function.

```
if (!mclInitializeApplication(NULL,0))
  {
    std::cerr << "could not initialize the application properly"
               << std::endl;
     return -1;
  }
```

The `mclInitializeApplication()` function sets up the application state for the MCR instance created in the application.

8 Add the following code below the code initializing the application.

```
if( !addmatrixInitialize() )
    {
        std::cerr << "could not initialize the library properly"
                     << std::endl;
     return -1;
 }
```

The `addmatrixInitialize()` function loads the required MATLAB code into the MCR.

9 Add a try/catch block after the block for `addmatrixInitialize()`.

10 In the try section of the try/catch block, add the following code.

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

// Create output array
mwArray out;
```

The code creates three instances of the mwArray class, `in1`, `in2`, and `out`. `in1` and `in2` passed as input parameters to the `addmatirx()` function generated by MATLAB. `out` is the value returned from the `addmatirx()` function. mwArray is a special class used by MATLAB generated code to facilitate the use of complex arrays.

11 After the code that initializes the parameters, add the following code to call the `addmatirx()` function and display the results.

```
addmatrix(1, out, in1, in2);

std::cout << "The value of added matrix is:" << std::endl;
std::cout << out << std::endl;
```

12 Add the following catch section to the try/catch block.

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

The first `catch` clause catches the MATLAB generated `mwException`. This exception is thrown by the MATLAB code running in the MCR.

The second `catch` clause catches any other exceptions that may be thrown.

13 Add the following after the try/catch block to terminate the MCR and clean up any resources it was using.

```
addmatrixTerminate();
mclTerminateApplication();
return 0;
```

`addmatrixTerminiate()` releases the resources used by the generated MATLAB code.

`mclTerminateApplication()` releases all state and resources used by the MCR for the application.

14 Save the C++ file.

The completed C++ file should resemble the following.

```
#include "addmatrix.h"

int run_main(int argc, char **argv)
{
  if (!mclInitializeApplication(NULL,0))
    {
      std::cerr << "could not initialize the application properly"
                << std::endl;
       return -1;
    }
  if( !addmatrixInitialize() )
    {
      std::cerr << "could not initialize the library properly"
                << std::endl;
      return -1;
    }

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

```
        mwArray in2(3, 3, mxDOUBLE_CLASS, mxREAL);
        in1.SetData(data, 9);
        in2.SetData(data, 9);

        // Create output array
        mwArray out;

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

        std::cout << "The value of added matrix is:" << 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;
      }

    addmatrixTerminate();
    mclTerminateApplication();
    return 0;
  }

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

15. Use the system's command line to navigate to the folder where you installed the C++ shared library.

16. Use `mbuild` to compile and link the application.

```
mbuild addmatrix.cpp addmatrix.lib
```

17 From the system's command prompt, run the application.

```
addmatrix
The value of added matrix is:
     2     8    14
     4    10    16
     6    12    18
```

To follow up on this example:

- Try installing the new application on a different computer.
- Try building an installer for the application.
- Try integrating a shared library that consists of more than one function.

Create a Deployable Archive for MATLAB Production Server

This example shows how to create a deployable archive for MATLAB Production Server using a MATLAB function. You can then hand the generated archive to a system administrator who will deploy it into MATLAB Production Server.

To create a deployable archive:

1. In MATLAB, examine the MATLAB code that you want to deploy.
   - a Open `addmatrix.m`.

     ```
     function a = addmatrix(a1, a2)

     a = a1 + a2;
     ```

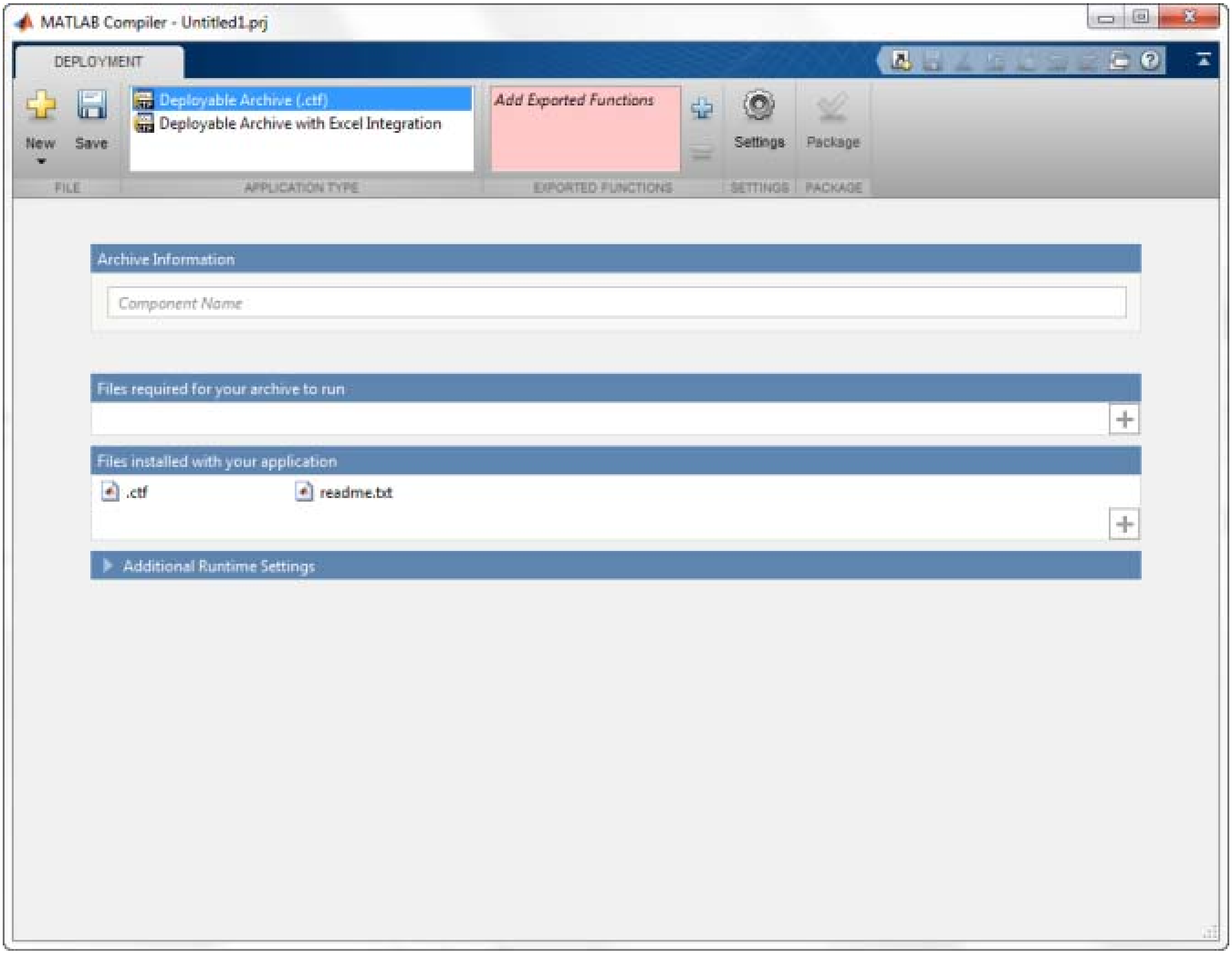
   - b At the MATLAB command prompt, enter `addmatrix(1,2)`.

     The output appears as follows:

     ```
     ans =

           3
     ```

2. Open the **Production Server Compiler**.
   - a On the toolstrip, select the **Apps** tab.
   - b Click the arrow on the far right of the tab to open the apps gallery.
   - c Click **Production Server Compiler**.


MATLAB Compiler - Untitled1.prj
DEPLOYMENT
New
Save
FILE
Deployable Archive (.ctf)
Deployable Archive with Excel Integration
APPLICATION TYPE
Add Exported Functions
EXPORTED FUNCTIONS
Settings
SETTINGS
Package
PACKAGE
Archive Information
Component Name
Files required for your archive to run
Files installed with your application
.ctf
readme.txt
Additional Runtime Settings

3 In the **Application Type** section of the toolstrip, select **Deployable Archive** from the list.

Note If the **Application Type** section of the toolstrip is collapsed, you can expand it by clicking the down arrow .

---

4. Specify the MATLAB functions you want to deploy.
   - a. In the **Exported Functions** section of the toolstrip, click the plus button.

     Note If the **Exported Functions** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

   - b. Using the file explorer, locate and select the `addmatrix.m` file.

     `addmatrix.m` is located in `*matlabroot*\extern\examples\compiler`.

   - c. Click **Open** to select the file and close the file explorer.

     addmatrix.m is added to the field. A minus button will appear below the plus button.

5. In the **Packaging Options** section of the toolstrip, verify that the **Runtime downloaded from web** check box is selected.

   Note If the **Packaging Options** section of the toolstrip is collapsed you can expand it by clicking the down arrow.

   This option creates an application installer that automatically downloads the MATLAB Compiler Runtime (MCR) and installs it.

6. Explore the main body of the project window.

   The project window is divided into the following areas:

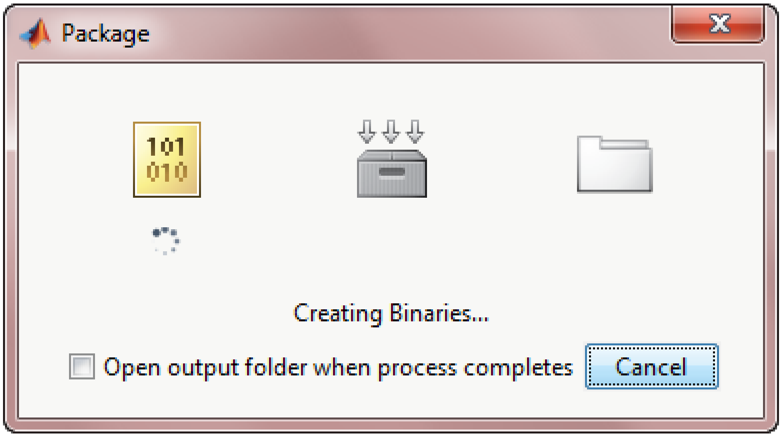
   - **Application Information** — Editable information about the deployed archive. This information is used by the generated installer to populate the installed application's metadata. See "Customize the Installer".

- **Additional Installer Options** — The default installation path for the generated installer. See “Customize the Installer”.
- **Files required for your application** — Additional files required by the archive. These files will be included in the generated archive. See “Manage Required Files in a Compiler Project”.
- **Files installed with your application** — Files that are installed with your archive. These files include:
  - `readme.txt`
  - `.ctf` file

  See “Specify Files to Install with the Application”.

7 Click **Package**.

The Package window opens while the library is being generated.



8 Select the **Open output folder when process completes** check box.

When the deployment process is complete, a file explorer opens and displays the generated output.

9 Verify the contents of the generated output:

- `for_redistribution` — A folder containing the installer to redistribute the archive to the system administrator responsible for the MATLAB Production Server
- `for_testing` — A folder containing the raw files generated by the compiler
- `for_redistribution_files_only` — A folder containing only the files needed to redistribute the archive
- `PackagingLog.txt` — A log file generated by the compiler.

10 Click **Close** on the Package window.

To lean more about MATLAB Production Server see “MATLAB Production Server”

For More Information

About This	Look Here
Detailed information on standalone applications	"Deploying Standalone Applications" on page 12-3
Creating libraries	"Integrate C Shared Libraries" on page 13-4 "Integrate C++ Shared Libraries" on page 13-18
Using the `mcc` command	"mcc Command Line Arguments Grouped by Task" on page A-8
Troubleshooting	"Common Issues" on page 14-4 "Failure Points and Possible Solutions" on page 14-5 "Troubleshooting mbuild" on page 14-16 "MATLAB® Compiler™" on page 14-18 "Deployed Applications" on page 14-22

2

Installation and Configuration

- “Install an ANSI C or C++ Compiler” on page 2-2
- “Configuring Your Options File with mbuild” on page 2-5
- “Solving Installation Problems” on page 2-8

Install an ANSI C or C++ Compiler

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

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

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

Supported ANSI C and C++ Windows Compilers

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

- Microsoft Visual C++® (MSVC).
  - The only compiler that supports the building of COM objects and Excel plug-ins is Microsoft Visual C++.
  - The only compiler that supports the building of .NET objects is Microsoft Visual C# Compiler for the Microsoft .NET Framework.
- Microsoft Windows SDK 7.1

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

Note For an up-to-date list of all the compilers supported by MATLAB and MATLAB Compiler, see the MathWorks Technical Support notes at http://www.mathworks.com/support/compilers/current_release/

Supported ANSI C and C++ UNIX Compilers

MATLAB Compiler software supports the native system compilers on:

- Linux

- Linux x86-64
- Mac OS X

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

Common Installation Issues and Parameters

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

Windows Operating System

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

UNIX Operating System

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

Configuring Your Options File with mbuild

In this section…
"What Is mbuild?" on page 2-5
"Locating and Customizing the Options File" on page 2-5

What Is mbuild?

Running the `mbuild` configuration script creates an option file that:

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

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

About mbuild and Linking

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

Locating and Customizing the Options File

- "Locating the Options File" on page 2-5
- "Changing the Options File" on page 2-6

Locating the Options File

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

- Current folder
- The `user profile` folder

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

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

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

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

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

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

Changing the Options File

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

```
mbuild -setup
```

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

To modify your options file on Windows:

1. Enter `mbuild -setup` to make a copy of the appropriate options file in your local area.
2. Edit your copy of the options file in your `user profile` folder to correspond to your specific needs, and save the modified file.

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

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

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

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

To modify your options file on the UNIX:

1. Use `mbuild -setup` to make a copy of the appropriate options file in your local area.
2. Edit your copy of the options file to correspond to your specific needs, and save the modified file.

Solving Installation Problems

You can contact MathWorks:

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

3

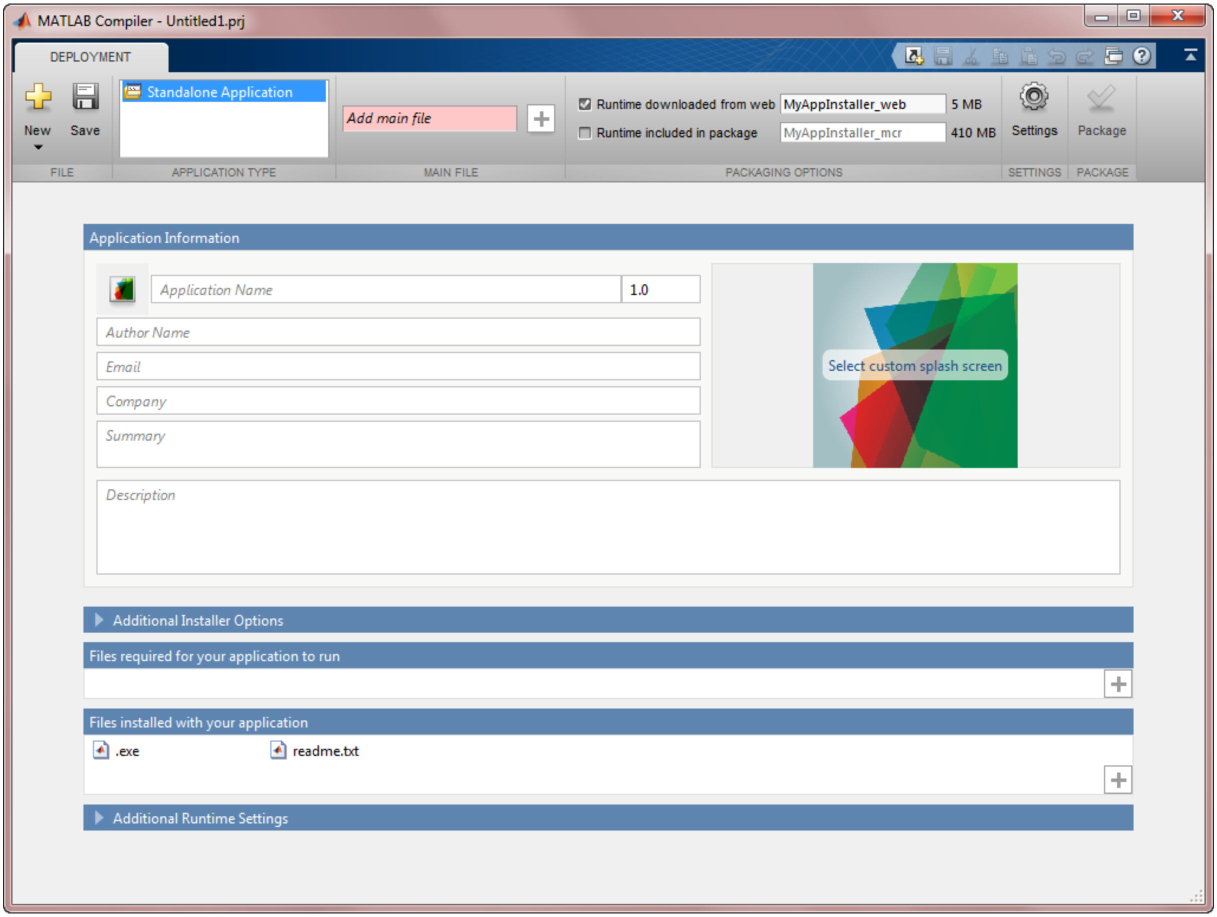
Deploying Standalone Applications

- "Compile a Standalone Application with the Standalone Compiler App" on page 3-2
- "Customize the Application's Run Time Behavior" on page 3-8
- "Compile a Standalone Application from the Command Line" on page 3-9
- "Working with Standalone Applications and Arguments" on page 3-11
- "Compile and Deploy Standalone Applications with the Parallel Computing Toolbox" on page 3-15
- "Run a Mac OS X Application" on page 3-18

Compile a Standalone Application with the Standalone Compiler App

To compile MATLAB code into a standalone application:

1. Open the **Application Compiler**.
   - a On the toolstrip select the **Apps** tab.
   - b Click the arrow at the far right of the tab to open the apps gallery.
   - c Click **Application Compiler** to open the **MATLAB Compiler** project window.



MATLAB Compiler - Untitled1.prj
DEPLOYMENT
New
Save
Standalone Application
Add main file
FILE
APPLICATION TYPE
MAIN FILE
PACKAGING OPTIONS
Settings
Package
SETTINGS
PACKAGE
Application Information
Application Name
Author Name
Email
Company
Summary
Select custom splash screen
Description
Additional Installer Options
Files required for your application to run
Files installed with your application
.exe
readme.txt
Additional Runtime Settings

☑ Runtime downloaded from web MyAppInstaller_web 5 MB

☐ Runtime included in package MyAppInstaller_mcr 410 MB

Version: 1.0

Note To open an existing project, select it from the MATLAB **Current Folder** panel.

Note You can also launch the standalone compiler using the `applicationCompiler` function.

2 Specify the main file of the MATLAB application you want to deploy.

   a In the **Main File** section of the toolstrip, click the plus button.

   Note If the **Main File** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

   ---

   b In the file explorer that opens, locate and select the MATLAB file.

   c Click **Open** to select the file and close the file explorer.

   The selected file's name is added to the list of main files and the plus button will be replaced by a minus button. The file name is used as the default application name.

3 In the **Packaging Options** section of the toolstrip, specify how the installer will deliver the MATLAB Compiler Runtime (MCR) with the application.

Note If the **Packaging Options** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

You can select one or both of the following options:

- **Runtime downloaded from web** — Generates an installer that downloads the MCR installer from the Web.
- **Runtime included in package** — Generates an installer that includes the MCR installer.

Note Selecting both options creates two installers.

Regardless of what options are selected the generated installer scans the target system to determine if there is an existing installation of the appropriate MCR. If there is not, the installer installs the MCR.

4 Specify the name of any generated installers.

5 In the **Application Information** and **Additional Installer Options** sections of the compiler, customize the look and feel of the generated installer.

   You can change the information used to identify the application data used by the installer:

   - Splash screen

   Note On Windows, the splash screen will be displayed when the compiled application starts in addition to when the installer runs.

   - Application icon
   - Application version
   - Name and contact information of the application's author
   - Brief summary of the application's purpose
   - Detailed description of the application

   You can also change the default location into which the application is installed and provide some notes to the installer.

   All of the provided information is displayed as the installer runs.

   For more information see "Customize the Installer" on page 6-2.

6 In the **Files required for your application to run** section of the compiler, verify that all of the files required to run the MATLAB application are listed.

   Note These files are compiled into the generated binaries along with the main file.

   In general the built-in dependency checker automatically populates this section with the appropriate files. However, you can manually add any files it missed.

For more information see “Manage Required Files in a Compiler Project” on page 6-6.

7. In the **Files installed with your application** section of the compiler, verify that any additional non-MATLAB files you want installed with the application are listed.

   Note These files are placed in the `applications` folder of the installed application.

   This section automatically lists:

   - Generated executable
   - (Linux) Shell script for launching the application
   - Readme file

   You can manually add files to the list. Additional files can include documentation, sample data files, and examples to accompany the application.

   For more information see “Specify Files to Install with the Application” on page 6-8.

8. In the **Additional Runtime Settings** section of the compiler, specify some of the advanced runtime behaviors for the application.

   These behaviors include:

   - (Windows) if a command window is required to run the application
   - if the application generates a log file

   For more information see “Customize the Application’s Run Time Behavior” on page 3-8.

9. Click **Settings** to customize the flags passed to the compiler and the folders where the generated files are written.

10. Click **Package** to compile the MATLAB code and generate the installers.

11 Verify the contents of the generated output:

- `for_redistribution` — A folder containing the installer to distribute the standalone application
- `for_testing` — A folder containing the raw files generated by the compiler
- `for_redistribution_files_only` — A folder containing only the files needed to redistribute the application
- `PackagingLog.txt` — A log file generated by the compiler

Customize the Application's Run Time Behavior

In **Additional Runtime Settings**, you can change the following run-time behaviors for the compiled application:

- On Windows if a command window is opened when you double-click the application from the file explorer

  Note If the application generates output to the console or requires command line input, you must unselect this option.

- If the application generates a MATLAB log file

By default, all of these behaviors are set to false. When you double-click a compiled application in the Windows file explorer, the application's window opens without a command prompt and will not generate a log file.

Compile a Standalone Application from the Command Line

In this section...
"Execute Compiler Projects with deploytool" on page 5-12
"Compile a Standalone Application with mcc" on page 3-9

You can compile standalone applications from both the MATLAB command line and the system terminal command line:

- `deploytool` invokes the compiler app to execute a presaved compiler project
- `mcc` invokes the raw compiler

Execute Compiler Projects with deploytool

The `deploytool` command has two flags to invoke the compiler without opening a window:

- `-build` *`project_name`* — Invoke the compiler to build the project and do not generate an installer.
- `-package` *`project_name`* — Invoke the compiler to build the project and generate an installer.

For example, `deploytool -package magicsqaure` generates of the binary files defined by the `magicaqure` project and packages them into an installer that you can distribute to others.

Compile a Standalone Application with mcc

The `mcc` command invokes the raw compiler and provides fine-level control over the compilation of the application. It, however, cannot package the results in an installer.

To invoke the compiler to generate an application use either the `-m` or the`-e` flag with `mcc`. Both compile a MATLAB function and generate a standalone executable. The `-m` flag creates a standard executable that can be run from a command line. The`-e` flag is a Windows-specific option that generates an executable that does not open a command prompt when double-clicked from the Windows file explorer.

The following `mcc` options can be used for compiling standalone applications.

Option	Description
`-W main -T link:exe`	Generate a standard executable. Equivalent to using `-m`.
`-W WinMain -T link:exe`	Generate an executable that does not open a command prompt when double-clicked from the Windows file explorer. Equivalent to using `-e`.
`-a` *`filePath`*	Add any files on the path to the generated binaries.
`-d` *`outFolder`*	Specify the folder where the results of compilation are written.
`-o` *`fileName`*	Specify the name of the generated executable file.

Working with Standalone Applications and Arguments

In this section…

"Overview" on page 3-11

"Passing File Names, Numbers or Letters, Matrices, and MATLAB Variables" on page 3-11

"Running Standalone Applications that Use Arguments" on page 3-12

Overview

You usually create a standalone to simply run the application without passing or retrieving any arguments to or from it.

However, arguments can be passed to standalone applications created using MATLAB Compiler in the same way that input arguments are passed to any console-based application.

The following are example commands used to execute an application called `filename` from a DOS or Linux command prompt with different types of input arguments.

Passing File Names, Numbers or Letters, Matrices, and MATLAB Variables

To Pass....	Use This Syntax....	Notes
A file named `helpfile`	`filename helpfile`	
Numbers or letters	`filename 1 2 3 a b c`	Do *not* use commas or other separators between the numbers and letters you pass.

To Pass....	Use This Syntax....	Notes
Matrices as input	`filename "[1 2 3]" "[4 5 6]"`	Place double quotes around input arguments to denote a blank space.

MATLAB variables	`for k=1:10`
	`cmd = ['filename ',num2str(k)];`
	`system(cmd);`
	`end`
		To pass a MATLAB variable to a program as input, you must first convert it to a string.

Running Standalone Applications that Use Arguments

You call a standalone application that uses arguments from MATLAB with any of the following commands:

- `SYSTEM`
- `DOS`
- `UNIX`
- `!`

To pass the contents of a MATLAB variable to the program as an input, the variable must first be converted to a string. For example:

Using SYSTEM, DOS, or UNIX

Specify the entire command to run the application as a string (including input arguments). For example, passing the numbers and letters `1 2 3 a b c` could be executed using the `SYSTEM` command, as follows:

```
system('filename 1 2 3 a b c')
```

Using the ! (bang) Operator

You can also use the `!` (bang) operator, from within MATLAB, as follows:

```
!filename 1 2 3 a b c
```

When you use the `!` (bang) operator, the remainder of the input line is interpreted as the `SYSTEM` command, so it is not possible to use MATLAB variables.

Using a Windows System

To run a standalone application by double clicking on it, you create a batch file that calls the standalone application with the specified input arguments. For example:

```
rem This is main.bat file which calls
rem filename.exe with input parameters

filename "[1 2 3]" "[4 5 6]"
@echo off
pause
```

The last two lines of code in `main.bat` are added so that the window displaying your output stays open until you press a key.

Once you save this file, you run your code with the arguments specified above by double clicking on the icon for `main.bat`.

Using a MATLAB File You Plan to Deploy

When running MATLAB files that use arguments that you also plan to deploy with MATLAB Compiler, keep the following in mind:

- The input arguments you pass to your executable from a system prompt will be received as string input. Thus, if you expect the data in a different format (for example, double), you must first convert the string input to the required format in your MATLAB code. For example, you can use `STR2NUM` to convert the string input to numerical data.
- You cannot return values from your standalone application to the user. The only way to return values from compiled code is to either display it on the screen or store it in a file.

  In order to have data displayed back to the screen, do one of the following:

- Unsuppress the commands that yield your return data. Do not use semicolons to unsuppress.
- Use the `DISP` command to display the variable value, then redirect the outputs to other applications using redirects (the > operator) or pipes (||) on non-Windows systems.

Taking Input Arguments and Displaying to a Screen Using a MATLAB File. Here are two ways to use a MATLAB file to take input arguments and display data to the screen:

Method 1

```
function [x,y]=foo(z);

if ischar(z)
z=str2num(z);
else
z=z;
end
x=2*z % Omit the semicolon after calculation to display the value on the sc
y=z^2;
disp(y) %Use DISP command to display the value of a variable explicitly
```

Method 2

```
function [x,y]=foo(z);

if isdeployed
z=str2num(z);
end
x=2*z % Omit the semicolon after calculation to display the value on the sc
y=z^2;
disp(y) % Use DISP command to display the value of a variable explicitly
```

Compile and Deploy Standalone Applications with the Parallel Computing Toolbox

In this section...

"Standalone Applications with a Profile Passed at Run-Time" on page 3-15

"Standalone Applications with an Embedded Profile" on page 3-16

Standalone Applications with a Profile Passed at Run-Time

When using the Parallel Computing Toolbox™, you can pass the cluster profile to the compiled application at runtime. The steps to do so are similar to using a standard compiled application. There are only a few extra steps:

1. Write Parallel Computing Toolbox code.
2. Use the **Cluster Profile Manager**'s **Export** button to export the desired profiles.

   The **Cluster Profile Manager** can be opened by clicking **Parallel > Manage Cluster Profiles**.
3. Compile the application.

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

   ---
4. Write a shell script that calls the application using the `-mcruserdata ParallelProfile:`*`profile`* flag.

   ```
   myApp -mcruserdata ParallelProfile:C:\work9b\pctdeploytool\myprofile.settin
   ```

   `profile` should be specified as the full path name for the cluster profile file.

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

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

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

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

---

5. Distribute the following files to people wishing to run the application:

   - the generated installer
   - the cluster profile
   - the script that starts the application using the cluster profile

---

Note Users of the application must have access to the cluster specified in the profile.

Standalone Applications with an Embedded Profile

When using the Parallel Computing Toolbox, you can include the cluster profile with the compiled application. The steps to do so are similar to using a standard compiled application. There are only a few extra steps:

1. Write Parallel Computing Toolbox code.

2. Write a second MATLAB function that uses `setmcruserdata` to load the cluster profile and pass it to the MCR.

   ```
   function run_parallel_funct
   setmcruserdata('ParallelProfile', 'profile');
   a = parallel_funct
   ```

```
end
```

3. Use the **Cluster Profile Manager**'s **Export** button to export the desired profile.

   The name used to save the cluster profile should match the *`profile`* value used in `setmcruserdata`.

   The **Cluster Profile Manager** can be opened by clicking **Parallel > Manage Cluster Profiles**.

4. Compile the application.

   a. Use the `run_`*`parallel_funct`* as the main file for the application.

   b. Include the cluster profile in the **Files required for your application** field of the compiler app.

   c. Include the `.m` file for *`parallel_funct`* in the **Files required for your application** field of the compiler app.

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

---

5. Distribute the generated installer to anyone interested in using the application.

Note Users of the application must have access to the cluster specified in the profile.

Run a Mac OS X Application

In this section…

"Overview" on page 3-18

"Installing the Macintosh Application Launcher Preference Pane" on page 3-18

"Configuring the Installation Area" on page 3-18

"Launching the Application" on page 3-21

Overview

Macintosh graphical applications, launched through the Mac OS X finder utility, require additional configuration if MATLAB software or the MCR were not installed in default locations.

Installing the Macintosh Application Launcher Preference Pane

Install the Macintosh Application Launcher preference pane, which gives you the ability to specify your installation area.

1. In the Mac OS X Finder, navigate to *`install_area`*`/toolbox/compiler/maci64`.

2. Double-click on **MW_App_Launch.prefPane**.

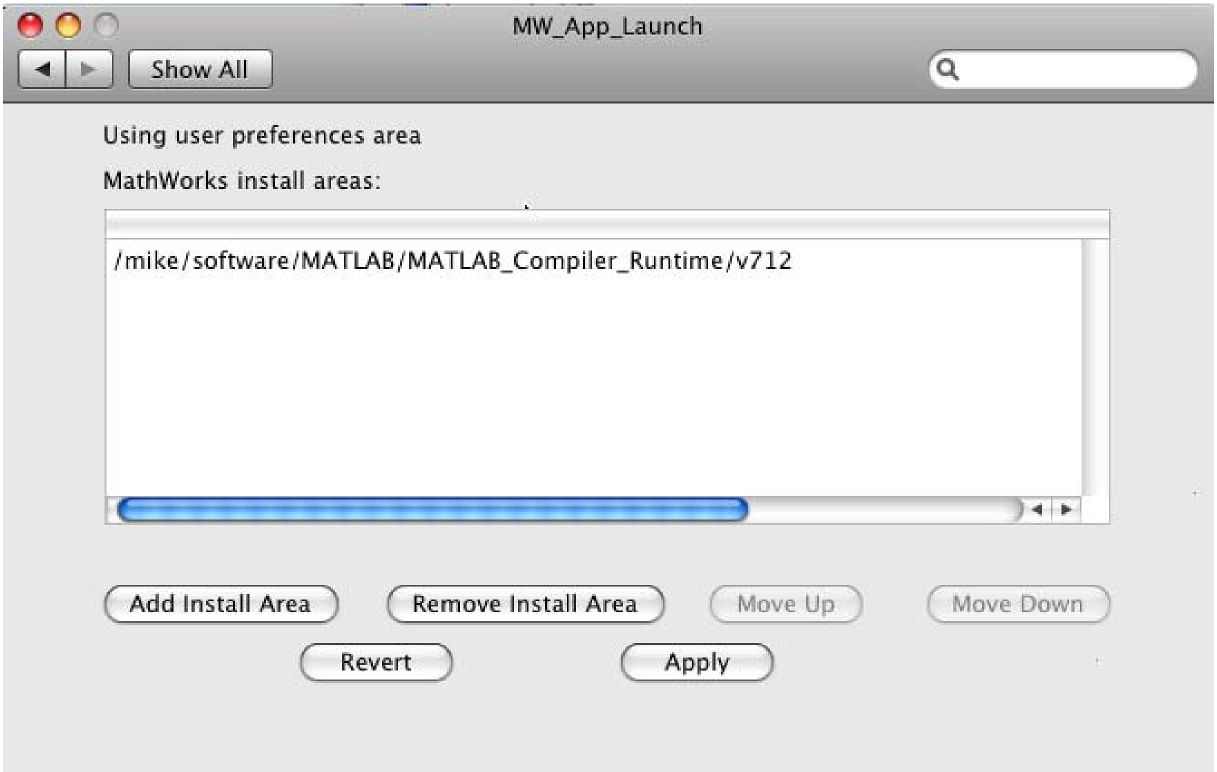
Note The Macintosh Application Launcher manages only *user* preference settings. If you copy the preferences defined in the launcher to the Macintosh System Preferences area, the preferences are still manipulated in the User Preferences area.

Configuring the Installation Area

Once the preference pane is installed, you configure the installation area.

1 Launch the preference pane by clicking on the apple logo in the upper left corner of the desktop.

2 Click on **System Preferences**. The **MW_App_Launch** preference pane appears in the **Other** area.


MW_App_Launch
Show All

Using user preferences area

MathWorks install areas:

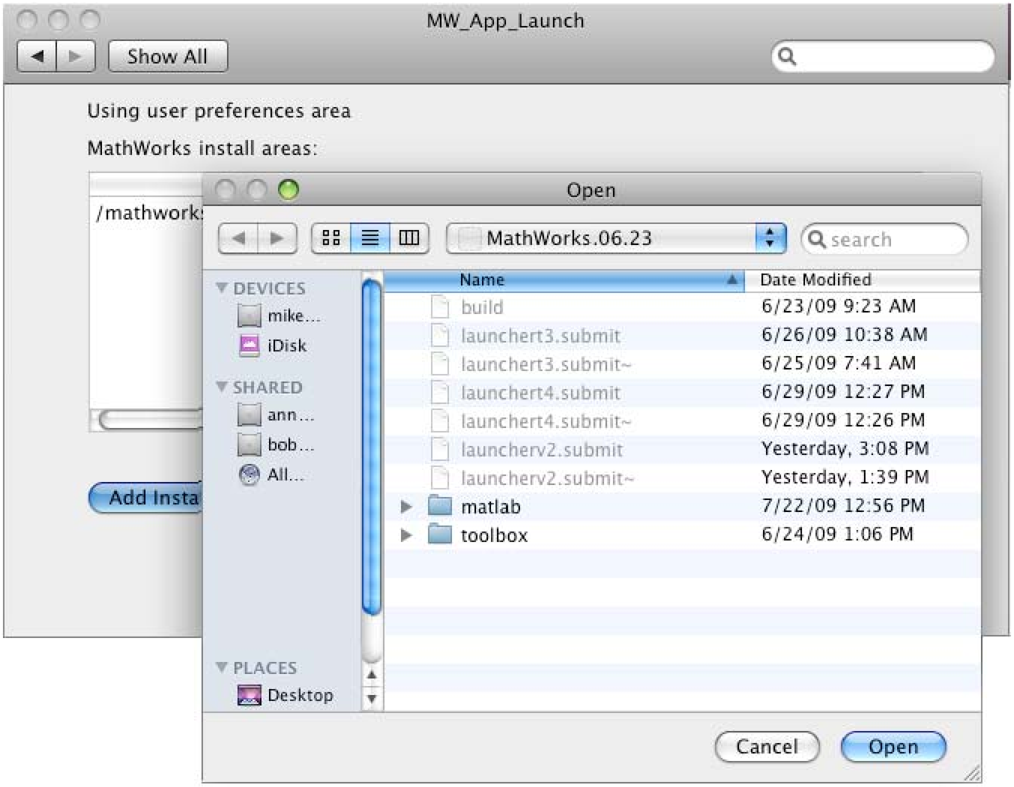
/mike/software/MATLAB/MATLAB_Compiler_Runtime/v712

Add Install Area
Remove Install Area
Move Up
Move Down
Revert
Apply

3 Click **Add Install Area** to define an installation area on your system.

4 Define the default installation path by browsing to it.

5 Click **Open**.


MW_App_Launch
Show All
Open
MathWorks.06.23
search
DEVICES
mike...
iDisk
SHARED
ann...
bob...
All...
PLACES
Desktop
Add Insta
Cancel
Open

Using user preferences area

MathWorks install areas:

/mathwork

Name	Date Modified
build	6/23/09 9:23 AM
launchert3.submit	6/26/09 10:38 AM
launchert3.submit~	6/25/09 7:41 AM
launchert4.submit	6/29/09 12:27 PM
launchert4.submit~	6/29/09 12:26 PM
launcherv2.submit	Yesterday, 3:08 PM
launcherv2.submit~	Yesterday, 1:39 PM
matlab	7/22/09 12:56 PM
toolbox	6/24/09 1:06 PM

Modifying Your Installation Area

Occasionally, you remove an installation area, define additional areas or change the order of installation area precedence.

You can use the following options in MathWorks® Application Launcher to modify your installation area:

- **Add Install Area** — Defines the path on your system where your applications install by default.
- **Remove Install Area** — Removes a previously defined installation area.

- **Move Up** — After selecting an installation area, click this button to move the defined path up the list. Binaries defined in installation areas at the top of the list have precedence over all succeeding entries.
- **Move Down** — After selecting an installation area, click this button to move the defined path down the list. Binaries defined in installation areas at the top of the list have precedence over all succeeding entries.
- **Apply** — Saves changes and exits MathWorks Application Launcher.
- **Revert** — Exits MathWorks Application Launcher without saving any changes.

Launching the Application

When you create a Macintosh application, a Macintosh bundle is created. If the application does not require standard input and output, launch the application by clicking on the bundle in the Mac OS X Finder utility.

The location of the bundle is determined by whether you use `mcc` or `applicationCompiler` to build the application:

- If you use `applicationCompiler`, the application bundle is placed in the compiled application's `for_redistribution` folder.
- If you use `mcc`, the application bundle is placed in the current working directory or in the output directory as specified by the `mcc` "-o Specify Output Name" on page 17-29 switch.

4

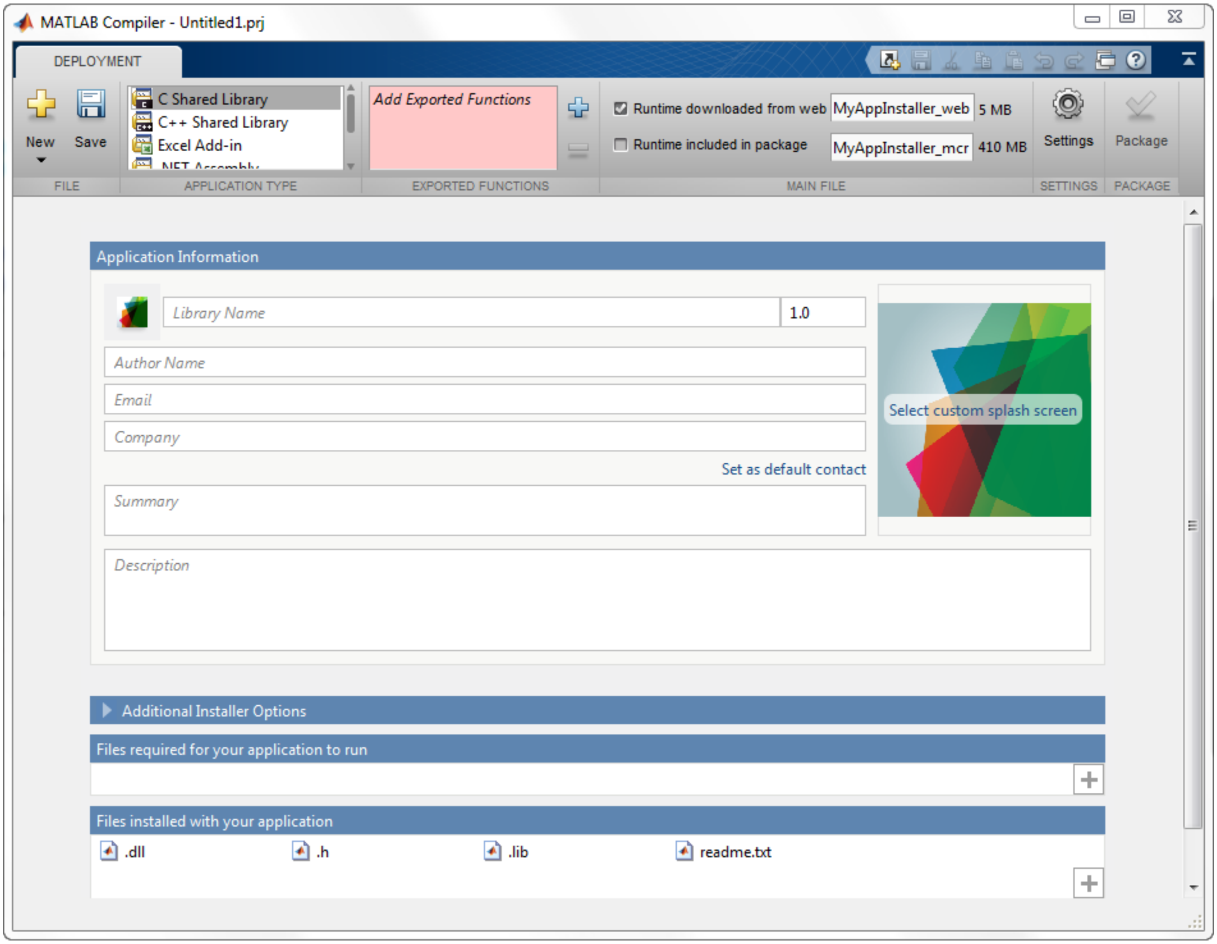
Deploying C/C++ Shared Libraries

- "Compile a C/C++ Shared Library with the Library Compiler App" on page 4-2
- "Compile a C/C++ Shared Library from the Command Line" on page 4-7
- "What Are Wrapper Files?" on page 4-9

Compile a C/C++ Shared Library with the Library Compiler App

To compile MATLAB code into a shared library:

1. Open the **Library Compiler**.
   - a On the toolstrip select the **Apps** tab.
   - b Click the arrow at the far right of the tab to open the apps gallery.
   - c Click **Library Compiler** to open the **MATLAB Compiler** project window.



MATLAB Compiler - Untitled1.prj

C Shared Library

☑ Runtime downloaded from web MyAppInstaller_web 5 MB

☐ Runtime included in package MyAppInstaller_mcr 410 MB

Library Name 1.0

Files installed with your application: .dll .h .lib readme.txt

DEPLOYMENT
New
Save
FILE
C Shared Library
C++ Shared Library
Excel Add-in
.NET Assembly
APPLICATION TYPE
Add Exported Functions
EXPORTED FUNCTIONS
MAIN FILE
Settings
SETTINGS
Package
PACKAGE
Application Information
Library Name
Author Name
Email
Company
Set as default contact
Select custom splash screen
Summary
Description
Additional Installer Options
Files required for your application to run
Files installed with your application

Note To open an existing project, select it from the MATLAB **Current Folder** panel.

Note You can also launch the library compiler using the `libraryCompiler` function.

---

2. In the **Application Type** section of the toolstrip, select either **C Shared Library** or **C++ Shared Library**.

   Note If the **Application Type** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

   ---

3. Specify the MATLAB files you want deployed in the shared library.
   1. In the **Exported Functions** section of the toolstrip, click the plus button.

      Note If the **Exported Functions** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

      ---

   2. In the file explorer that opens, locate and select one or more MATLAB files.
   3. Click **Open** to select the file and close the file explorer.

      The names of the selected files are added to the list and a minus button appears below the plus button. The name of the first file listed is used as the default application name.

4. In the **Packaging Options** section of the toolstrip, specify how the installer will deliver the MATLAB Compiler Runtime (MCR) with the shared library.

   Note If the **Packaging Options** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

   You can select one or both of the following options:

   - **Runtime downloaded from web** — Generates an installer that downloads the MCR installer from the Web.
   - **Runtime included in package** — Generates an installer that includes the MCR installer.

Note Selecting both options creates two installers.

Regardless of the options selected the generated installer scans the target system to determine if there is an existing installation of the appropriate MCR. If there is not, the installer installs the MCR.

5 Specify the name of any generated installers.

6 In the **Application Information** and **Additional Installer Options** sections of the compiler, customize the look and feel of the generated installer.

   You can change the information used to identify the application data used by the installer:

   - Splash screen
   - Installer icon
   - Library version
   - Name and contact information of the library's author
   - Brief summary of the library's purpose
   - Detailed description of the library

   You can also change the default location into which the library is installed and provide some notes to the installer.

   All of the provided information is displayed as the installer runs.

   For more information see "Customize the Installer" on page 6-2.

7 In the **Files required for your application to run** section of the compiler, verify that all of the files required by the deployed MATLAB functions are listed.

Note These files are compiled into the generated binaries along with the exported files.

In general the built-in dependency checker will automatically populate this section with the appropriate files. However, if needed you can manually add any files it missed.

For more information see "Manage Required Files in a Compiler Project" on page 6-6.

8. In the **Files installed with your application** section of the compiler, verify that any additional non-MATLAB files you want installed with the application are listed.

Note These files are placed in the `applications` folder of the installed application.

This section automatically lists:

- Generated shared library
- Header file
- Dynamically linked library
- Readme file

You can manually add files to the list. Additional files can include documentation, sample data files, and examples to accompany the application.

For more information see "Specify Files to Install with the Application" on page 6-8.

9. Click the **Settings** button to customize the flags passed to the compiler and the folders to which the generated files are written.

10. Click the **Package** button to compile the MATLAB code and generate any installers.

Compile a C/C++ Shared Library from the Command Line

In this section...

"Execute Compiler Projects with deploytool" on page 5-12

"Compile a Shared Library with mcc" on page 4-7

You can compile shared libraries from both the MATLAB command line and the system terminal command line:

- `deploytool` invokes the compiler app to execute presaved compiler projects
- `mcc` invokes the raw compiler

Execute Compiler Projects with deploytool

The `deploytool` command has two flags to invoke the compiler without opening a window:

- `-build` *`project_name`* — Invoke the compiler to build the project and do not generate an installer.
- `-package` *`project_name`* — Invoke the compiler to build the project and generate an installer.

For example, `deploytool -package magicsqaure` generates of the binary files defined by the `magicaqure` project and packages them into an installer that you can distribute to others.

Compile a Shared Library with mcc

The `mcc` command invokes the raw compiler and provides fine-level control over the compilation of the shared library. It, however, cannot package the results in an installer.

To invoke the compiler to generate a library use the `-l` flag with `mcc`. The `-l` flag creates a C shared library that you can integrate into applications developed in C or C++.

For compiling shared libraries, you can also use the following options.

Compiler Shared Library Options

Option	Description
`-W lib:`*`name`*` -T link:lib`	Generate a C shared library. Equivalent to using `-l`.
`-W cpplib:`*`name`*` -T link:lib`	Generate a C++ shared library.
`-a `*`filePath`*	Add the file, or files, on the path to the generated binary.
`-d `*`outFolder`*	Specify the folder into which the results of compilation are written.

What Are Wrapper Files?

In this section...
"C Library Wrapper" on page 4-9
"C++ Library Wrapper" on page 4-9

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

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

- Performs wrapper-specific initialization and termination
- Provides the dispatching of function calls to the MCR

C Library Wrapper

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

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

C++ Library Wrapper

The `-W cpplib:libname` option produces the C++ library wrapper file. This option allows the inclusion of an arbitrary set of MATLAB files into a library. The generated header file contains all of the entry points for all of the compiled MATLAB functions.

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

5

Compiling Deployable Archives for MATLAB Production Server

- “State-Dependent Functions” on page 5-2
- “Unsupported MATLAB Data Types for Client and Server Marshaling” on page 5-5
- “Compile a Deployable Archive with the Production Server Compiler App” on page 5-6
- “Compile a Deployable Archive from the Command Line” on page 5-12

State-Dependent Functions

MATLAB code that you want to deploy often carries *state*—a specific data value in a program or program variable.

Does My MATLAB Function Carry State?

Example of carrying state in a MATLAB program include, but are not limited to:

- Modifying or relying on the MATLAB path and the Java class path
- Accessing MATLAB state that is inherently persistent or global. Some example of this include:
  - Random number seeds
  - Handle Graphics® root objects that retain data
  - MATLAB or MATLAB toolbox settings and preferences
- Creating global and persistent variables.
- Loading MATLAB objects (MATLAB classes) into MATLAB. If you access a MATLAB object in any way, it loads into MATLAB.
- Calling MEX files, Java methods, or C# methods containing static variables.

Defensive Coding Practices

If your MATLAB function not only carries state, but *relies on it* for your function to properly execute, you must take additional steps (listed in this section) to ensure state retention.

When you deploy your application, consider cases where you carry state, and safeguard against that state's corruption if needed. *Assume* that your state may be changed and code defensively against that condition.

The following are examples of "defensive coding" practices:

Reset System-Generated Values in the Deployed Application

If you are using a random number seed, for example, reset it in your deployed application program to ensure the integrity of your original MATLAB function.

Validate Global or Persistent Variable Values

If you must use global or persistent variables, always validate their value in your deployed application and reset if needed.

Ensure Access to Data Caches

If your function relies on cached transaction replies, for instance, ensure your deployed system and application has access to that cache outside of the MATLAB environment.

Use Simple Data Types When Possible

Simple data types are usually not tied to a specific application and means of storing state. Your options for choosing an appropriate state-preserving tool increase as your data types become less complicated and specific.

Avoid Using MATLAB Callback Functions

Avoid using MATLAB callbacks, such as `timer`. Callback functions have the ability to interrupt and override the current state of the MATLAB Production Server worker and may yield unpredictable results in multiuser environments.

Techniques for Preserving State

The most appropriate method for preserving state depends largely on the type of data you need to save.

- Databases provide the most versatile and scalable means for retaining stateful data. The database acts as a generic repository and can generally work with any application in an enterprise development environment. It does not impose requirements or restrictions on the data structure or layout. Another related technique is to use comma-delimited files, in applications such as Microsoft Excel.
- Data that is specific to a third-party programming language, such as Java and C#, can be retained using a number of techniques. Consult the online documentation for the appropriate third-party vendor for best practices on preserving state.

Caution Using MATLAB `LOAD` and `SAVE` functions is often used to preserve state in MATLAB applications and workspaces. While this may be successful in some circumstances, it is highly recommended that the data be validated and reset if needed, if not stored in a generic repository such as a database.

Unsupported MATLAB Data Types for Client and Server Marshaling

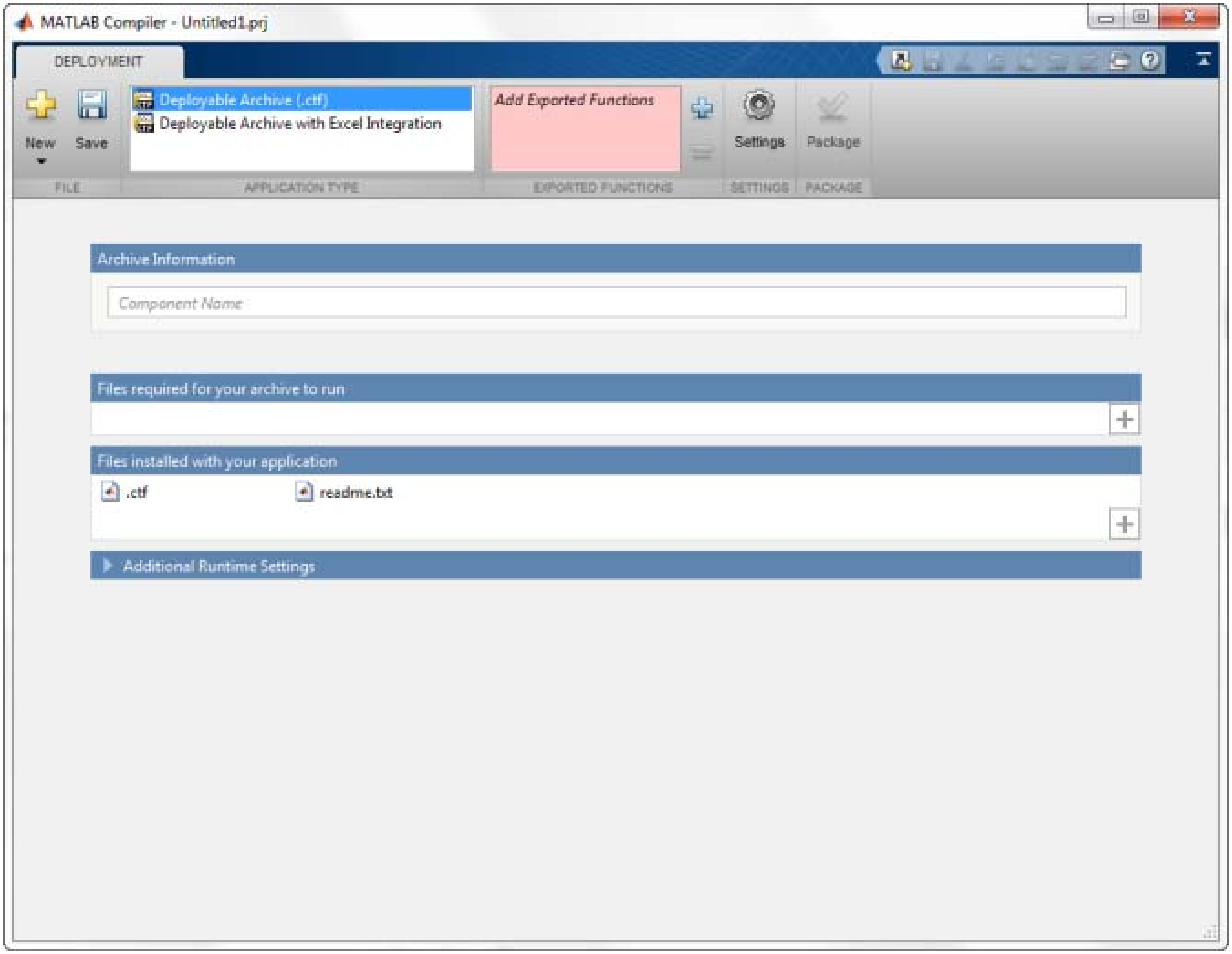
These data types are not supported for marshaling between MATLAB Production Server server instances and clients:

- MATLAB function handles
- Complex (imaginary) data
- Sparse arrays

Compile a Deployable Archive with the Production Server Compiler App

To compile MATLAB code into a deployable archive:

1. Open the **Production Server Compiler**.
    a. On the toolstrip select the **Apps** tab on the toolstrip.
    b. Click the arrow at the far right of the tab to open the apps gallery.
    c. Click **Production Server Compiler**.


MATLAB Compiler - Untitled1.prj
DEPLOYMENT
New
Save
FILE
Deployable Archive (.ctf)
Deployable Archive with Excel Integration
APPLICATION TYPE
Add Exported Functions
EXPORTED FUNCTIONS
Settings
SETTINGS
Package
PACKAGE
Archive Information
Component Name
Files required for your archive to run
Files installed with your application
.ctf
readme.txt
Additional Runtime Settings

Note To open an existing project, select it from the MATLAB **Current Folder** panel.

Note You can also launch the compiler using the `productionServerCompiler` function.

---

2 In the **Application Type** section of the toolstrip, select **Deployable Archive**.

Note If the **Application Type** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

---

3 Specify the MATLAB files you want deployed in the package.

   a In the **Exported Functions** section of the toolstrip, click the plus button.

Note If the **Exported Functions** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

---

   b In the file explorer that opens, locate and select one or more the MATLAB files.

   c Click **Open** to select the file and close the file explorer.

   The names of the selected files are added to the list and a minus button appears below the plus button. The name of the first file listed is used as the default application name.

4 In the **Packaging Options** section of the toolstrip, specify how the installer will deliver the MATLAB Compiler Runtime (MCR) with the archive.

Note If the **Packaging Options** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

You can select one or both of the following options:

- **Runtime downloaded from web** — Generates an installer that downloads the MCR installer from the Web.
- **Runtime included in package** — Generates an installer that includes the MCR installer.

Note Selecting both options creates two installers.

Regardless of the options selected the generated installer scans the target system to determine if there is an existing installation of the appropriate MCR. If there is not, the installer installs the MCR.

5. Specify the name of any generated installers.
6. In the **Application Information** and **Additional Installer Options** sections of the compiler, customize the look and feel of the generated installer.

   You can change the information used to identify the application data used by the installer:

   - Splash screen
   - Installer icon
   - Version
   - Name and contact information of the archive's author
   - Brief summary of the archive's purpose
   - Detailed description of the archive

   You can also change the default location into which the archive is installed and provide some notes to the installer.

   All of the provided information is displayed as the installer runs.

   For more information see "Customize the Installer" on page 6-2.

7. In the **Files required for your application to run** section of the compiler, verify that all of the files required by the deployed MATLAB functions are listed.

   Note These files are compiled into the generated binaries along with the exported files.

   The built-in dependency checker will automatically populate this section with the appropriate files. However, if needed you can manually add any files it missed.

   For more information see "Manage Required Files in a Compiler Project" on page 6-6.

8. In the **Files installed with your application** section of the compiler, verify that any additional non-MATLAB files you want installed with the application are listed.

   Note These files are placed in the `applications` folder of the installation.

   This section automatically lists:

   - Generated deployable archive
   - Readme file

   You can manually add files to the list. Additional files can include documentation, sample data files, and examples to accompany the application.

   For more information see "Specify Files to Install with the Application" on page 6-8.

9. Click the **Settings** button to customize the flags passed to the compiler and the folders to which the generated files are written.

10. Click the **Package** button to compile the MATLAB code and generate any installers.

11 Verify that the generated output contains:

- `for_redistribution` — A folder containing the installer to distribute the archive
- `for_testing` — A folder containing the raw generated files to create the installer
- `for_redistribution_files_only` — A folder containing only the files needed to redistribute the archive
- `PackagingLog.txt` — A log file generated by the compiler

Compile a Deployable Archive from the Command Line

In this section...

"Execute Compiler Projects with deploytool" on page 5-12

"Compile a Deployable Archive with mcc" on page 5-12

You can compile deployable archives from both the MATLAB command line and the system terminal command line:

- `deploytool` invokes the compiler app to execute a pre-saved compiler project
- `mcc` invokes the raw compiler

Execute Compiler Projects with deploytool

The `deploytool` command has two flags to invoke the compiler without opening a window:

- `-build` *`project_name`* — Invoke the compiler to build the project and do not generate an installer.
- `-package` *`project_name`* — Invoke the compiler to build the project and generate an installer.

For example, `deploytool -package magicsqaure` generates of the binary files defined by the `magicaqure` project and packages them into an installer that you can distribute to others.

Compile a Deployable Archive with mcc

The `mcc` command invokes the raw compiler and provides fine-level control over the compilation of the deployable archive. It, however, cannot package the results in an installer.

To invoke the compiler to generate a deployable arcive use the `-W CTF:`*`component_name`* flag with `mcc`. The `-W CTF:`*`component_name`* flag creates a deployable archive called *`component_name`*`.ctf`.

For compiling deployable archives, you can also use the following options.

Compiler Java Options

Option	Description
`-a` *`filePath`*	Add any files on the path to the generated binary.
`-d` *`outFolder`*	Specify the folder into which the results of compilation are written.
`class{`*`className`*`:`*`mfilename...`*`}`	Specify that an additional class is generated that includes methods for the listed MATLAB files.

6

Customizing a Compiler Project

- “Customize the Installer” on page 6-2
- “Manage Required Files in a Compiler Project” on page 6-6
- “Specify Files to Install with the Application” on page 6-8
- “Manage Support Packages” on page 6-9

Customize the Installer

In this section...
"Change the Application Icon" on page 6-2 "Add Application Information" on page 6-3 "Change the Splash Screen" on page 6-4 "Change the Installation Path" on page 6-4 "Change the Logo" on page 6-5 "Edit the Installation Notes" on page 6-5

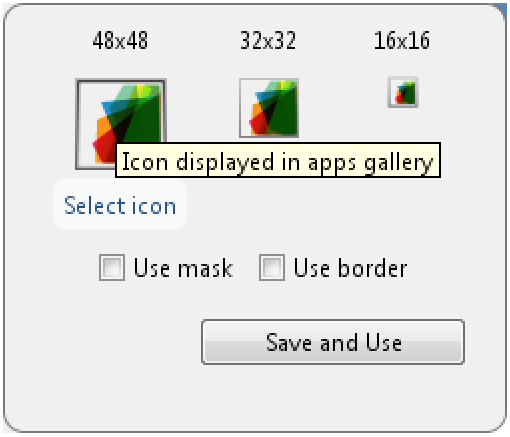
Change the Application Icon

The application icon is used for the generated installer. For standalone applications, it is also the application's icon.

You can change the default icon in **Application Information**. To set a custom icon:

1 Click the graphic to the left of the **Application name** field.

A window previewing the icon opens.


48x48
32x32
16x16
Icon displayed in apps gallery
Select icon
☐ Use mask
☐ Use border
Save and Use

2. Click **Select icon**.
3. Using the file explorer, locate the graphic file to use as the application icon.
4. Select the graphic file.
5. Click **OK** to return to the icon preview.
6. Select **Use mask** to fill any blank spaces around the icon with white.
7. Select **Use border** to add a border around the icon.
8. Click **Save and Use** to return to the main compiler window.

Add Application Information

The **Application Information** section of the compiler app allows you to provide these values:

- Name

  Determines the name of the installed MATLAB components. For example, if the name is `foo`, the installed executable would be `foo.exe`, the Windows start menu entry would be **foo**. The folder created for the application would be *`InstallRoot`*`/foo`.

  The default value is the name of the first function listed in the **Main File(s)** field of the compiler.
- Version

  The default value is 1.0.
- Author name
- Support e-mail address
- Company name

  Determines the full installation path for the installed MATLAB components. For example, if the company name is `bar`, the full installation path would be *`InstallRoot`*`/bar/`*`ApplicationName`*.
- Summary
- Description

This information is all optional and, unless otherwise stated, is only used for display purposes. It appears on the first page of the installer. On Windows systems, this information is also displayed in the Windows **Add/Remove Programs** control panel.

Change the Splash Screen

The installer's splash screen displays after the installer is started. It is displayed, along with a status bar, while the installer initializes.

You can change the default image by clicking the **Select custom splash screen** link in **Application Information**. When the file explorer opens, locate and select a new image.

Note You can drag and drop a custom image onto the default splash screen.

Change the Installation Path

Default Installation Paths on page 6-4 lists the default path the installer will use when installing the compiled binaries onto a target system.

Default Installation Paths

Windows	`C:\Program Files\`*`companyName`*`\`*`appName`*
Mac OS X	`/Applications/`*`companyName`*`/`*`appName`*
Linux	`/usr/`*`companyName`*`/`*`appName`*

You can change the default installation path by editing the **Default installation folder** field under **Additional Installer Options**.

The **Default installation folder** field has two parts:

- root folder — A drop down list that offers options for where the install folder is installed. Custom Installation Roots on page 6-5 lists the optional root folders for each platform.

Custom Installation Roots

Windows	`C:\Users\userName\AppData`
Linux	`/usr/local`

- install folder — A text field specifying the path appended to the root folder.

Change the Logo

The logo displays after the installer is started. It is displayed on the right side of the installer.

You change the default image by clicking the **Select custom logo** link in **Additional Installer Options**. When the file explorer opens, locate and select a new image.

Note You can drag and drop a custom image onto the default logo.

Edit the Installation Notes

Installation notes are displayed once the installer has successfully installed the packaged files on the target system. They can provide useful information concerning any additional set up that is required to use the installed binaries or simply provide instructions for how to run the application.

The field for editing the installation notes is in **Additional Installer Options**.

Manage Required Files in a Compiler Project

In this section...

"Dependency Analysis" on page 6-6

"Using the Compiler Apps" on page 6-6

"Using mcc" on page 6-7

Dependency Analysis

The compiler uses a dependency analysis function to automatically determine what additional MATLAB files are required for the application to compile and run. These files are automatically compiled into the generated binary. The compiler does not generate any wrapper code allowing direct access to the functions defined by the required files.

Using the Compiler Apps

If you are using one of the compiler apps, the required files discovered by the dependency analysis function are listed in the **Files required by your application to run** field.

To add files:

1. Click the plus button in the field.
2. Select the desired file from the file explorer.
3. Click **OK**.

To remove files:

1. Select the desired file.
2. Press the **Delete** key.

Caution Removing files from the list of required files may cause your application to not compile or to not run properly when deployed.

Using mcc

If you are using `mcc` to compile your MATLAB code, the compiler does not display a list of required files before running. Instead, it compiles all of the required files that are discovered by the dependency analysis function and adds them to the generated binary file.

You can add files to the list by passing one, or more, `-a` arguments to `mcc`. The `-a` arguments add the specified files to the list of files to be added into the generated binary. For example, `-a hello.m` adds the file `hello.m` to the list of required files and `-a ./foo` adds all of the files in `foo`, and its subfolders, to the list of required files.

Specify Files to Install with the Application

The compiler apps package files to install along with the ones it generates. By default the installer includes a readme file with instructions on installing the MATLAB Compiler Runtine(MCR) and configuring it.

These files are listed in the **Files installed with your application** section of the app.

to add files to the list:

1. Click the plus button in the field.
2. Select the desired file from the file explorer.
3. Click **OK** to close the file explorer.

To remove files from the list:

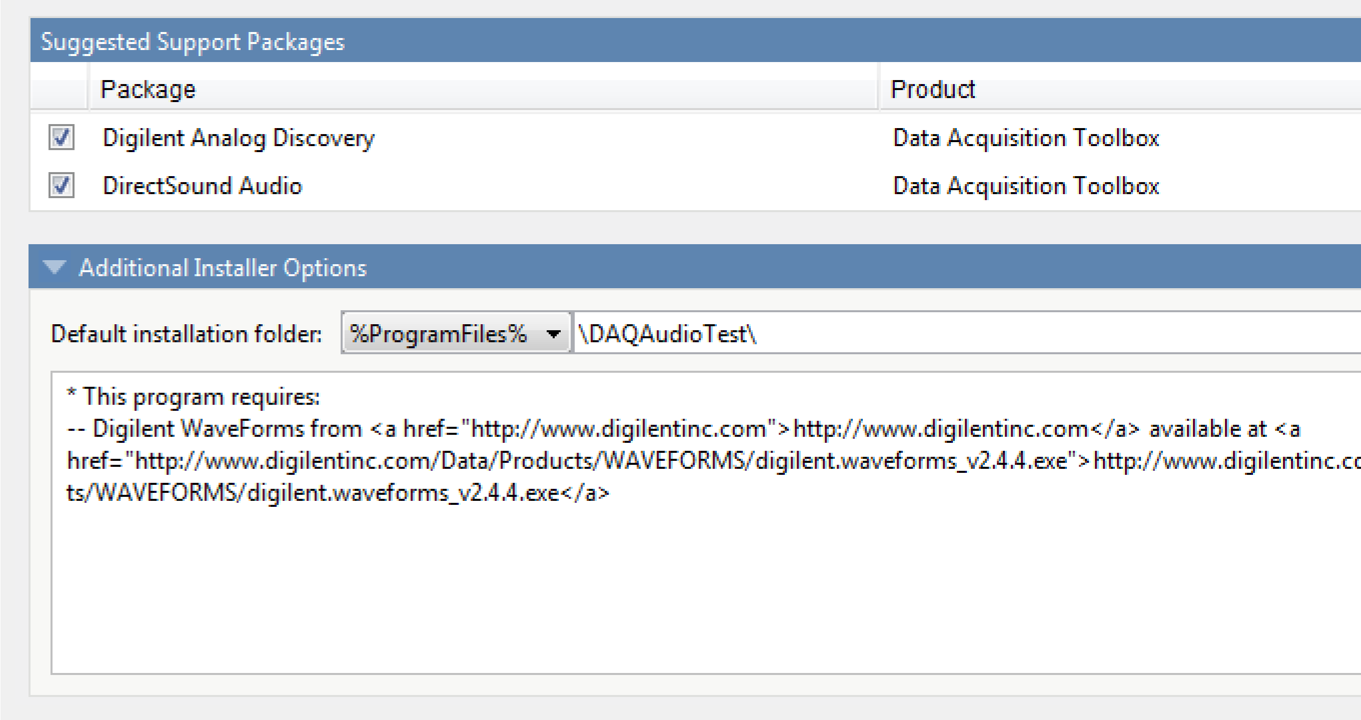
1. Select the desired file.
2. Press the **Delete** key.

Caution Removing the binary targets from the list results in an installer that does not install the intended functionality.

When installed on a target computer, the files listed in the **Files installed with your application** are placed in the `application` folder.

Manage Support Packages

Many MATLAB toolboxes use support packages to interact with hardware or to provide additional processing capabilities. If your MATLAB code uses a toolbox with an installed support package, MATLAB Compiler displays a **Suggested Support Packages** section.



	Package	Product
☑	Digilent Analog Discovery	Data Acquisition Toolbox
☑	DirectSound Audio	Data Acquisition Toolbox

Default installation folder: %ProgramFiles% \DAQAudioTest\

* This program requires:
-- Digilent WaveForms from http://www.digilentinc.com available at <a
href="http://www.digilentinc.com/Data/Products/WAVEFORMS/digilent.waveforms_v2.4.4.exe">http://www.digilentinc.co
ts/WAVEFORMS/digilent.waveforms_v2.4.4.exe

The list displays all installed support packages that your MATLAB code requires. The list is determined using these criteria:

- the support package is installed
- your code has a direct dependency on the support package
- your code is dependent on the base product of the support package

- your code is dependent on at least one of the files listed as a dependency in the `mcc.xml` file of the support package, and the base product of the support package is MATLAB

Deselect support packages that are not required by your application.

Some support packages require third-party drivers that MATLAB Compiler cannot package. In this case, the compiler adds the information to the installation notes. You can edit installation notes in the **Additional Installer Options** section of the app. To remove the installation note text, deselect the support package with the third-party dependency.

Caution Any text you enter beneath the generated text will be lost if you deselect the support package.

7

MATLAB Code Deployment

- “Application Deployment Products and the Compiler Apps” on page 7-2
- “Write Deployable MATLAB Code” on page 7-10
- “How the Deployment Products Process MATLAB Function Signatures” on page 7-14
- “Load MATLAB Libraries using loadlibrary” on page 7-16
- “Use MATLAB Data Files (MAT Files) in Compiled Applications” on page 7-18

Application Deployment Products and the Compiler Apps

In this section...

"What Is the Difference Between the Compiler Apps and the mcc Command Line?" on page 7-2

"How Does MATLAB® Compiler™ Software Build My Application?" on page 7-2

"Dependency Analysis Function" on page 7-5

"MEX-Files, DLLs, or Shared Libraries" on page 7-6

"Component Technology File (CTF Archive)" on page 7-6

What Is the Difference Between the Compiler Apps and the mcc Command Line?

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

Compiler app advantages include:

- You perform related deployment tasks with a single intuitive interface.
- You maintain related information in a convenient project file.
- Your project state persists between sessions.
- You load previously stored compiler projects from a prepopulated menu.
- Package applications for distribution.

How Does MATLAB Compiler Software Build My Application?

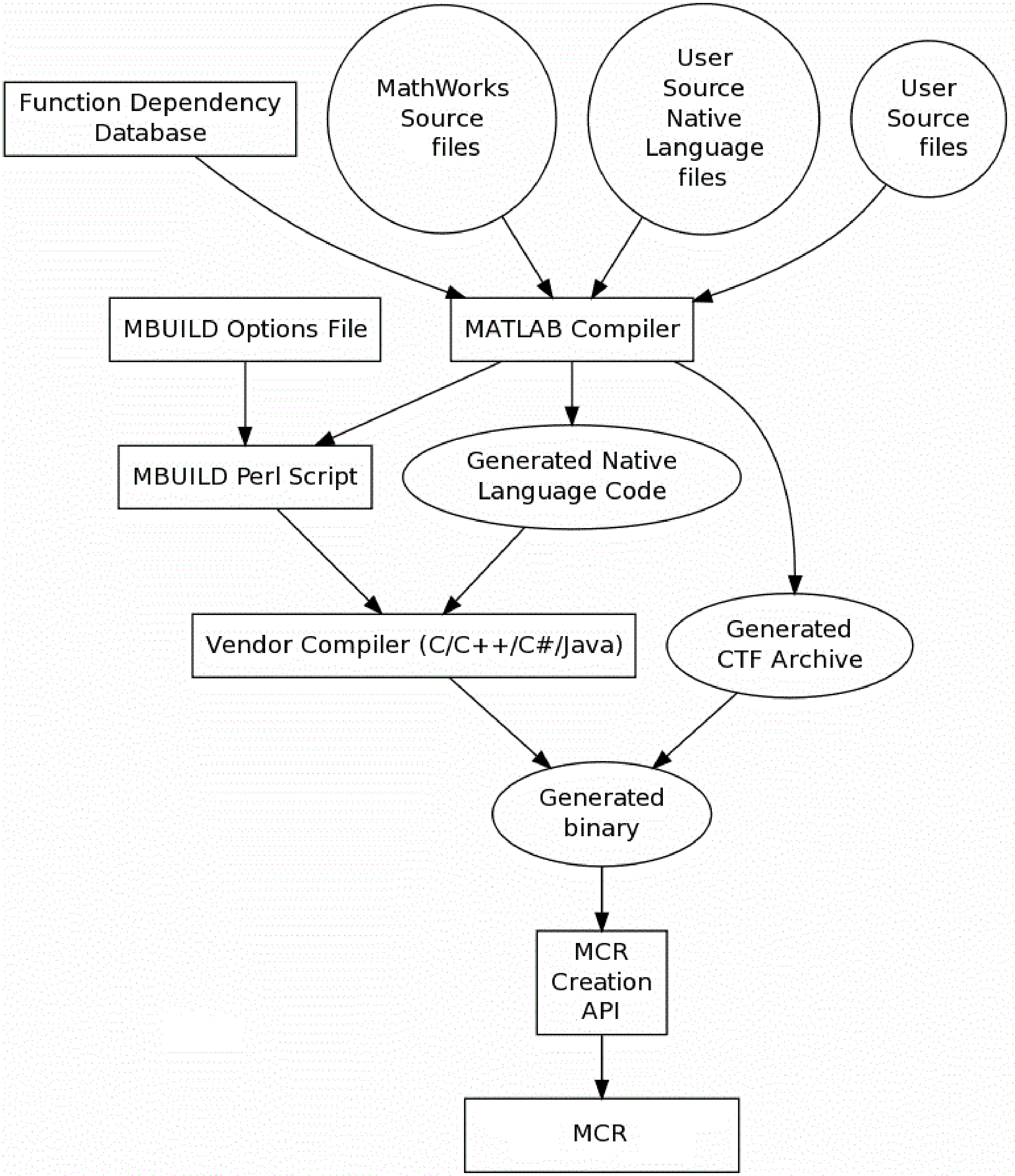
To build an application, MATLAB Compiler software performs these tasks:

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

2. Analyzes files for dependencies using a dependency analysis function. Dependencies affect deployability and originate from functions called by the file. Deployability is affected by:

   - File type — MATLAB, Java, MEX, and so on.
   - File location — MATLAB, MATLAB toolbox, user code, and so on.

   For more information about how the compiler does dependency analysis, see "Dependency Analysis Function" on page 7-5.


Function Dependency Database
MathWorks Source files
User Source Native Language files
User Source files
MBUILD Options File
MATLAB Compiler
MBUILD Perl Script
Generated Native Language Code
Vendor Compiler (C/C++/C#/Java)
Generated CTF Archive
Generated binary
MCR Creation API
MCR

- Circles: Input Files
- Boxes: MathWorks-supplied components

4. Creates a CTF archive from the input files and their dependencies. For more details about CTF archives see "Component Technology File (CTF Archive)" on page 7-6.
5. Generates target-specific wrapper code. For example, a C main function requires a very different wrapper than the wrapper for a Java interface class.
6. Generates target-specific binary package. For library targets such as C++ shared libraries, Java packages, or .NET assemblies, the compiler will invoke the required third-party compiler.

Dependency Analysis Function

MATLAB Compiler uses a dependency analysis function to determine the list of necessary files to include in the generated package. Sometimes, this process generates a large list of files, particularly when MATLAB object classes exist in the compilation and the dependency analyzer cannot resolve overloaded methods at compile time. Dependency analysis also processes `include`/`exclude` files on each pass.

Tip To improve compile time performance and lessen application size, prune the path with the `mcc` command's `-N` and `-p` flags. You can also specify **Files required for your application** in the compiler app.

The dependency analyzer searches for executable content such as:

- MATLAB files
- P-files

  Note If the MATLAB file corresponding to the p-file is not available, the dependency analysis will not be able to determine the p-file's dependencies.

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

- MEX-files

The dependency analyzer does not search for data files of any kind. You must manually include data files in the search.

MEX-Files, DLLs, or Shared Libraries

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

- Since the dependency analyzer cannot examine MEX-files, DLLs, or shared libraries to determine their dependencies, explicitly include all executable files these files require. To do so, use either the `mcc -a` option or the **Files required for your application to run** field in the compiler app.
- If you have any doubts that the dependency analyzer can find a MATLAB function called by a MEX-file, DLL, or shared library, then manually include that function. To do so, use either the `mcc -a` option or the **Files required for your application to run** field in the compiler app.
- Not all functions are compatible with MATLAB Compiler. Check the file `mccExcludedFiles.log` after your build completes. This file lists all functions called from your application that you cannot deploy.

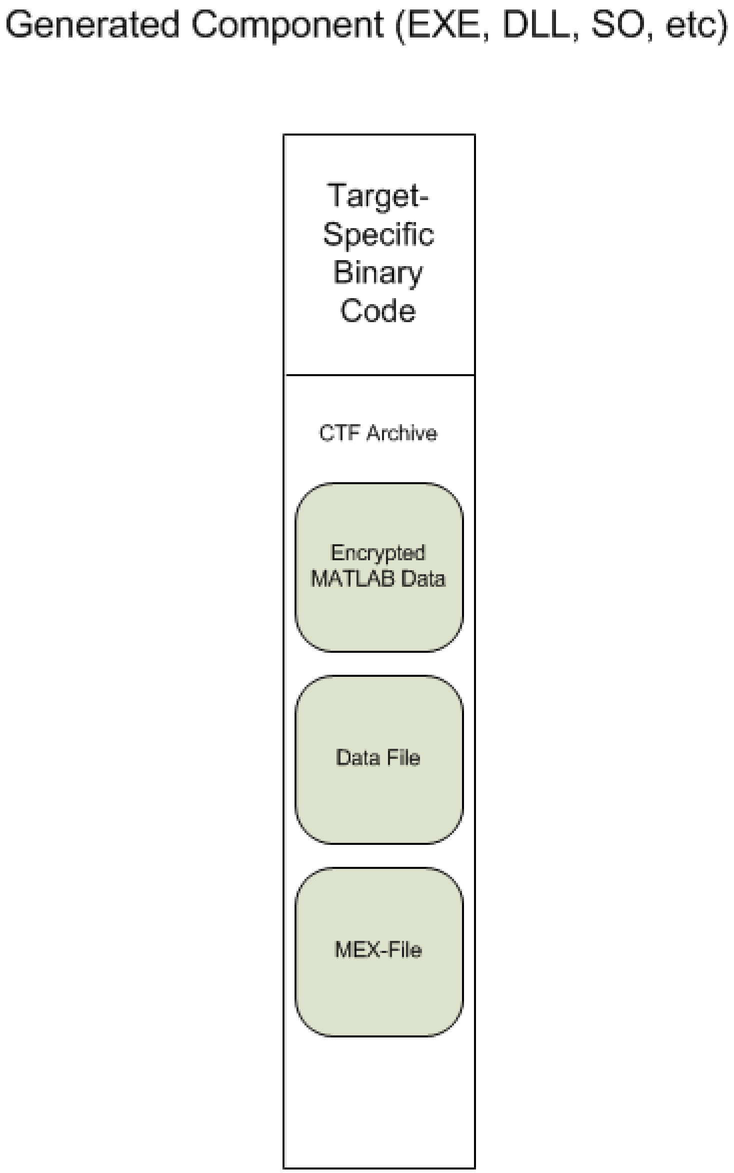
Component Technology File (CTF Archive)

Each application or shared library you produce using MATLAB Compiler has an embedded Component Technology File (CTF) archive. The archive contains all the MATLAB based content (MATLAB files, MEX-files, and so on) associated with the component. All MATLAB files in the CTF archive are encrypted using the Advanced Encryption Standard (AES) cryptosystem.

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

Information on CTF Archive Embedding/Extraction and Component Cache

Product	Refer to
MATLAB Compiler	"MCR Component Cache and CTF Archive Embedding" on page 11-11
MATLAB Builder NE	"MCR Component Cache and CTF Archive Embedding"
MATLAB Builder JA	"CTF Archive Embedding and Extraction"
MATLAB Builder EX	Using MCR Component Cache and CTF Archive Embedding


Generated Component (EXE, DLL, SO, etc)
Target-Specific Binary Code
CTF Archive
Encrypted MATLAB Data
Data File
MEX-File

Additional Details

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

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

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

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

Write Deployable MATLAB Code

In this section...

"Compiled Applications Do Not Process MATLAB Files at Runtime" on page 7-10

"Do Not Rely on Changing Directory or Path to Control the Execution of MATLAB Files" on page 7-11

"Use ismcc and isdeployed Functions To Execute Deployment-Specific Code Paths" on page 7-12

"Gradually Refactor Applications That Depend on Noncompilable Functions" on page 7-12

"Do Not Create or Use Nonconstant Static State Variables" on page 7-12

"Get Proper Licenses for Toolbox Functionality You Want to Deploy" on page 7-13

Compiled Applications Do Not Process MATLAB Files at Runtime

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

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

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

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

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

1. Run the code once in MATLAB to obtain your generated function.
2. Compile the MATLAB code with MATLAB Compiler, including the generated function.

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

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

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

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

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

If you find you cannot avoid placing `addpath` calls in your MATLAB code, use `ismcc` and `isdeployed`. See "Use ismcc and isdeployed Functions To Execute Deployment-Specific Code Paths" on page 7-12 for details.

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

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

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

Gradually Refactor Applications That Depend on Noncompilable Functions

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

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

- *Design-time code* is code that is currently evolving. Almost all code goes through a phase of perpetual rewriting, debugging, and optimization. In some toolboxes, such as the Neural Network Toolbox product, the code goes through a period of self-training as it reacts to various data permutations and patterns. Such code is almost never designed to be deployed.
- *Run-time code*, on the other hand, has solidified or become stable—it is in a finished state and is ready to be deployed by the end user.

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

Do Not Create or Use Nonconstant Static State Variables

Avoid using the following:

- Global variables in MATLAB code

- Static variables in MEX-files
- Static variables in Java code

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

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

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

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

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

Get Proper Licenses for Toolbox Functionality You Want to Deploy

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

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

How the Deployment Products Process MATLAB Function Signatures

In this section...
"MATLAB Function Signature" on page 7-14
"MATLAB Programming Basics" on page 7-14

MATLAB Function Signature

MATLAB supports multiple signatures for function calls.

The generic MATLAB function has the following structure:

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

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

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

All arguments represent a specific MATLAB type.

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

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

MATLAB Programming Basics

Creating a Deployable MATLAB Function

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

```
>> 1 + 1
```

cannot be deployed.

However, the following calculation:

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

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

Taking Inputs into a Function

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

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

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

Load MATLAB Libraries using loadlibrary

Note It is important to understand the difference between the following:

- MATLAB `loadlibrary` function — Loads shared library into MATLAB.
- Operating system `loadlibrary` function — Loads specified Windows or UNIX operating system module into the address space of the calling process.

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

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

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

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

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

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

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

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

Once the prototype file is generated, add the file to the CTF archive of the application being compiled. You can do this with the `-a` option (if using the `mcc` command) or by dragging it under **Other/Additional Files** (as a helper file) if using the Deployment Tool.

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

Restrictions on Using MATLAB Function loadlibrary with MATLAB Compiler

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

- You can not use `loadlibrary` inside of MATLAB to load a shared library built with MATLAB Compiler.
- With MATLAB Compiler Version 3.0 (R13SP1) and earlier, you cannot compile calls to `loadlibrary` because of general restrictions and limitations of the product.

Use MATLAB Data Files (MAT Files) in Compiled Applications

In this section...

"Explicitly Including MAT files Using the %#function Pragma" on page 7-18

"Load and Save Functions" on page 7-18

"MATLAB Objects" on page 7-21

Explicitly Including MAT files Using the %#function Pragma

MATLAB Compiler excludes MAT files from "Dependency Analysis Function" on page 7-5 by default.

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

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

Load and Save Functions

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

- Use `isdeployed` to determine if your code is running in or out of the MATLAB workspace.
- Specify the data file by either using `WHICH` (to locate its full path name) define it relative to the location of `ctfroot`.
- All MAT-files are unchanged after `mcc` runs. These files are not encrypted when written to the CTF archive.

For more information about CTF archives, see "Component Technology File (CTF Archive)" on page 7-6.

See the `ctfroot` reference page for more information about `ctfroot`.

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

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

The following example specifies three MATLAB data files:

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

1. Navigate to *`matlab_root`*`\extern\examples\compiler\Data_Handling`.
2. Compile `ex_loadsave.m` with the following `mcc` command:

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

ex_loadsave.m.

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

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

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

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

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

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

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

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

MATLAB Objects

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

```
%#function class_constructor
```

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

8

C and C++ Standalone Executable and Shared Library Creation

- "Input and Output Files" on page 8-2
- "Dependency Analysis Function and User Interaction with the Compilation Path" on page 8-8

Input and Output Files

In this section...

"Standalone Executable" on page 8-2
"C Shared Library" on page 8-3
"C++ Shared Library" on page 8-5
"Macintosh 64 (Maci64)" on page 8-7

Standalone Executable

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

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

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

C Shared Library

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

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

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

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

C++ Shared Library

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

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

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

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

Macintosh 64 (Maci64)

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

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

Dependency Analysis Function and User Interaction with the Compilation Path

addpath and rmpath in MATLAB

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

- The path is modified for the current MATLAB session only.
- If MATLAB Compiler is run outside of MATLAB, this doesn't work unless a `savepath` is done in MATLAB.

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

Passing -I <directory> on the Command Line

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

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

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

- *`matlabroot`*`\toolbox\matlab`
- *`matlabroot`*`\toolbox\local`
- *`matlabroot`*`\toolbox\compiler\deploy`
- *`matlabroot`*`\toolbox\compiler`

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

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

```
p <directory>
```

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

- If a folder is included with `-p` that is on the original MATLAB path, the folder and all its subfolders that appear on the original path are added to the compilation path in an order-sensitive context.
- If a folder is included with `-p` that is not on the original MATLAB path, that folder is not included in the compilation. (You can use `-I` to add it.)
- If a path is added with the `-I` option while this feature is active (`-N` has been passed) and it is already on the MATLAB path, it is added in the order-sensitive context as if it were included with `-p`. Otherwise, the folder is added to the head of the path, as it normally would be with `-I`.

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

Deployment Process

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

- “Overview” on page 9-2
- “Deploying to Developers” on page 9-3
- “Deploying to End Users” on page 9-6
- “Working with the MCR” on page 9-14
- “Deploy Applications Created Using Parallel Computing Toolbox” on page 9-37
- “Deploying a Standalone Application on a Network Drive (Windows Only)” on page 9-38
- “MATLAB® Compiler™ Deployment Messages” on page 9-40
- “Using MATLAB® Compiler™ Generated DLLs in Windows Services” on page 9-41
- “Reserving Memory for Deployed Applications with MATLAB Memory Shielding” on page 9-42

Overview

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

- “Deploying to Developers” on page 9-3
- “Deploying to End Users” on page 9-6

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

Watch a Video

Watch a video about deploying applications using MATLAB Compiler.

Deploying to Developers

In this section…

"Procedure" on page 9-3

"What Software Does a Developer Need?" on page 9-3

"Ensuring Memory for Deployed Applications" on page 9-5

Procedure

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

---

1. Create a package that contains the software necessary to support the compiled MATLAB code. It is frequently helpful to install the MCR on development machines, for testing purposes. See "What Software Does a Developer Need?" on page 9-3
2. Write instructions for how to use the package.
   a. If your package was created with the compiler app, developers can just run the installer generated by the compiler.
   b. All developers must set path environment variables properly. See "MCR Path Settings for Development and Testing" on page 16-2.
3. Distribute the package and instructions.

What Software Does a Developer Need?

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

- "Standalone Application" on page 9-4
- "C or C++ Shared Library" on page 9-4

Standalone Application

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

Software Module	**Description**
MCR Installer (Windows)	The MCR Installer is a self-extracting executable that installs the necessary components to develop your application. This file is included with MATLAB Compiler. Run `mcrinstaller` function to obtain name of executable.
MCR Installer (Linux)	The MCR Installer is a self-extracting executable that installs the necessary components to develop your application on UNIX machines (other than Mac). This file is included with MATLAB Compiler. Run `mcrinstaller` function to obtain name of binary.
MCR Installer (Mac)	Run `mcrinstaller` function to obtain name of binary.
`*application_name*.exe` (Windows)	Application created by MATLAB Compiler. Maci64 must include the bundle directory hierarchy.
`*application_name*` (UNIX)	
`*application_name*.app` (Maci64)	

C or C++ Shared Library

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

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

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 9-42 for more information.

Deploying to End Users

In this section...

"Steps by the Developer to Deploy to End Users" on page 9-6

"What Software Does the End User Need?" on page 9-9

"Using Relative Paths with Project Files" on page 9-12

"Porting Generated Code to a Different Platform" on page 9-12

"Extracting a CTF Archive Without Executing the Component" on page 9-12

"Ensuring Memory for Deployed Applications" on page 9-13

Steps by the Developer to Deploy to End Users

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

1. Create a package that contains the software needed at run time. See "What Software Does a Developer Need?" on page 9-3 for more details.

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

- MATLAB preferences set at compile time are inherited by the compiled application. Therefore, include no files in your preferences folder that you do not want exposed to end users.
- Preferences set by a compiled application do not affect the MATLAB preferences, and preferences set in MATLAB do not affect a compiled application until that application is recompiled. MATLAB does not save your preferences folder until you exit MATLAB. Therefore, if you change your MATLAB preferences, stop and restart MATLAB before attempting to recompile using your new preferences.

.

The preferences folder is as follows:

- `$HOME/.matlab/`*`current_release`* on UNIX
- *`system root`*`\profiles\`*`user`*`\application data\mathworks\matlab\`*`current_release`* on Windows

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

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

---

2 Write instructions for the end user. See “Steps by the End User” on page 9-7.

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 9-8.

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

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

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

Using the MCR Installer GUI

1. When the MCR Installer wizard appears, click **Next** to begin the installation. Click **Next** to continue.
2. In the Select Installation Folder dialog box, specify where you want to install the MCR and whether you want to install the MCR for just yourself or others. Click **Next** to continue.

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

---

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

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

   The MCR Installer automatically:

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

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

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

What Software Does the End User Need?

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

- "Standalone Compiled Application That Accesses Shared Library" on page 9-9
- ".NET Application" on page 9-10
- "COM Application" on page 9-10
- "Java Application" on page 9-11
- "Microsoft® Excel® Add-in" on page 9-11

Standalone Compiled Application That Accesses Shared Library

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

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

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

.NET Application

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

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

COM Application

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

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

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

Java Application

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

Microsoft Excel Add-in

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

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

Using Relative Paths with Project Files

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

Porting Generated Code to a Different Platform

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

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

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

Extracting a CTF Archive Without Executing the Component

CTF archives contain content (MATLAB files and MEX-files) that need to be extracted from the archive before they can be executed. In order to extract the archive you must override the default CTF embedding option (see “MCR Component Cache and CTF Archive Embedding” on page 11-11). To do this, ensure that you compile your component with the “-C Do Not Embed CTF Archive by Default” on page 17-27 option.

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

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

```
extractCTF hello.ctf
```

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

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

Ensuring Memory for Deployed Applications

If you are having trouble obtaining memory for your deployed application, use MATLAB Memory Shielding for deployed applications to ensure a maximum amount of contiguous allocated memory. See "Reserving Memory for Deployed Applications with MATLAB Memory Shielding" on page 9-42 for more information.

Working with the MCR

In this section…

"About the MATLAB® Compiler™ Runtime (MCR)" on page 9-14

"The MCR Installer" on page 9-15

"Installing the MCR Non-Interactively" on page 9-23

"Uninstalling the MCR" on page 9-26

"MCR Startup Options" on page 9-28

"Using the MCR User Data Interface" on page 9-32

"Displaying MCR Initialization Start-Up and Completion Messages For Users" on page 9-34

About the MATLAB Compiler Runtime (MCR)

The MATLAB Compiler Runtime (MCR) is a standalone set of shared libraries that enables the execution of MATLAB files on computers without an installed version of MATLAB. Applications that use components built with MATLAB Compiler require access to an appropriate version of the MCR to run.

End-users of compiled components without access to MATLAB must install the MATLAB Compiler Runtime (MCR) on their computers or know the location of a network-installed MCR. The installers generated by the compiler app include the MCR installer. If you compiled your component using `mcc`, you should direct your end-users to download the MCR installer from the Web at `http://www.mathworks.com/products/compiler/mcr`.

See "The MCR Installer" on page 9-15 for more information.

How is the MCR Different from MATLAB?

This MCR differs from MATLAB in several important ways:

- In the MCR, MATLAB files are securely encrypted for portability and integrity.
- MATLAB has a desktop graphical interface. The MCR has all the MATLAB functionality without the graphical interface.

- The MCR is version-specific. You must run your applications with the version of the MCR associated with the version of MATLAB Compiler with which it was created. For example, if you compiled an application using version 4.10 (R2009a) of MATLAB Compiler, users who do not have MATLAB installed must have version 7.10 of the MCR installed. Use `mcrversion` to return the version number of the MCR.
- The MATLAB and Java paths in an MCR instance are fixed and cannot be changed. To change them, you must first customize them within MATLAB.

Performance Considerations and the MCR

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

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

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

The MCR Installer

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

Installing the MCR

To install the MCR, users of your component must run the MCR Installer.

Note When packaging compiled components for distribution, MATLAB Compiler can include the Web-based or the local MCR installer in the distribution package.

To install the MCR:

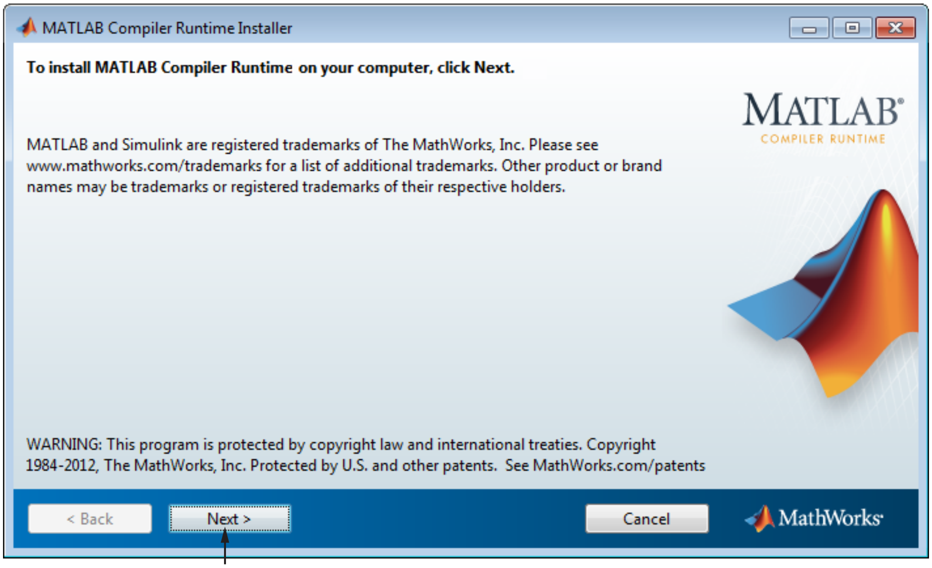
1 Start the MCR Installer.

Computer	**Steps**
Windows	Double-click the compiled component package self-extracting archive file, typically named `my_program_pkg.exe`, where `my_program` is the name of the compiled component. This extracts the MCR Installer from the archive, along with all the files that make up the MCR. Once all the files have been extracted, the MCR Installer starts automatically.

Linux	Extract the contents of the compiled component package, which is a Zip file on Linux systems, typically named, `my_program_pkg.zip`, where `my_program` is the name of the compiled component. Use the `unzip` command to extract the files from the package.
	`unzip MCRInstaller.zip`
	Run the MCR Installer script, from the directory where you unzipped the package file, by entering:
	`./install`
	For example, if you unzipped the package and MCR Installer in `\home\USER`, you run the `./install` from `\home\USER`.
	Note On Mac systems, you may need to enter an administrator username and password after you run `./install`.

Mac	

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



To install MATLAB Compiler Runtime on your computer, click Next.

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. Please 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.

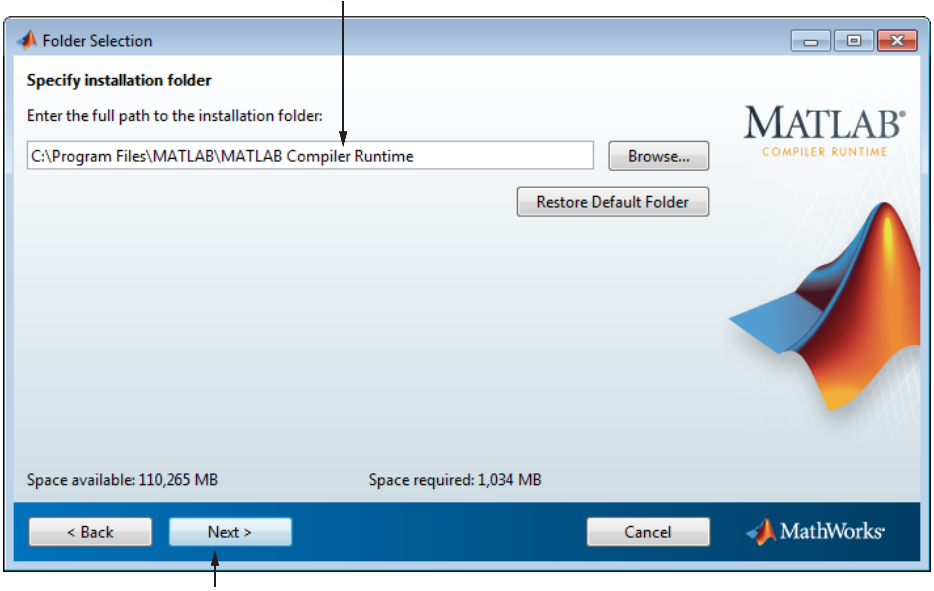
WARNING: This program is protected by copyright law and international treaties. Copyright 1984-2012, The MathWorks, Inc. Protected by U.S. and other patents. See MathWorks.com/patents

Click Next.

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

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

Specify installation folder.


Folder Selection
MATLAB
COMPILER RUNTIME
Browse...
Restore Default Folder
< Back
Next >
Cancel
MathWorks

Specify installation folder

Enter the full path to the installation folder:

C:\Program Files\MATLAB\MATLAB Compiler Runtime

Space available: 110,265 MB

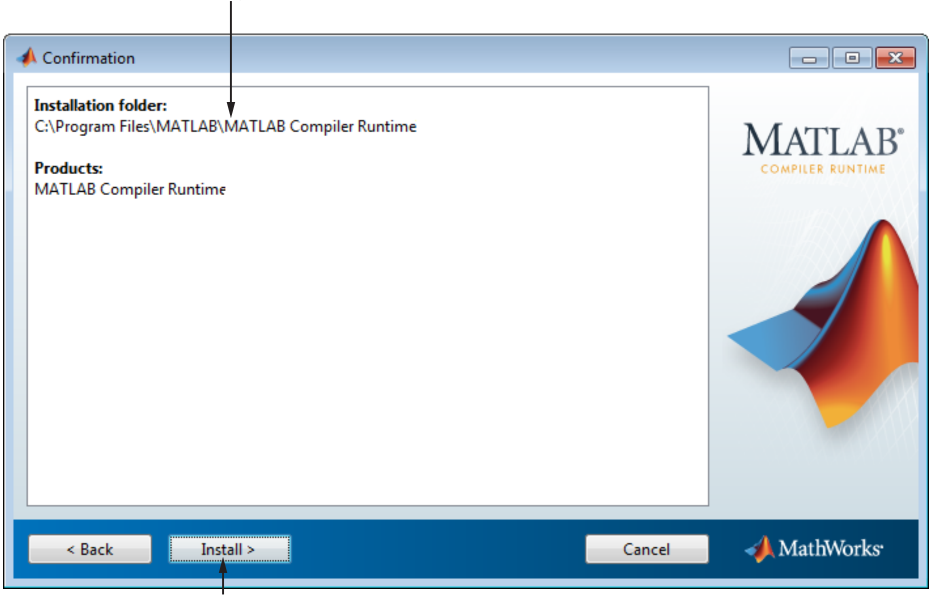
Space required: 1,034 MB

Click Next.

4 Confirm your choices and click **Next**.

The MCR Installer starts copying files into the installation folder.

Con firm your choices.

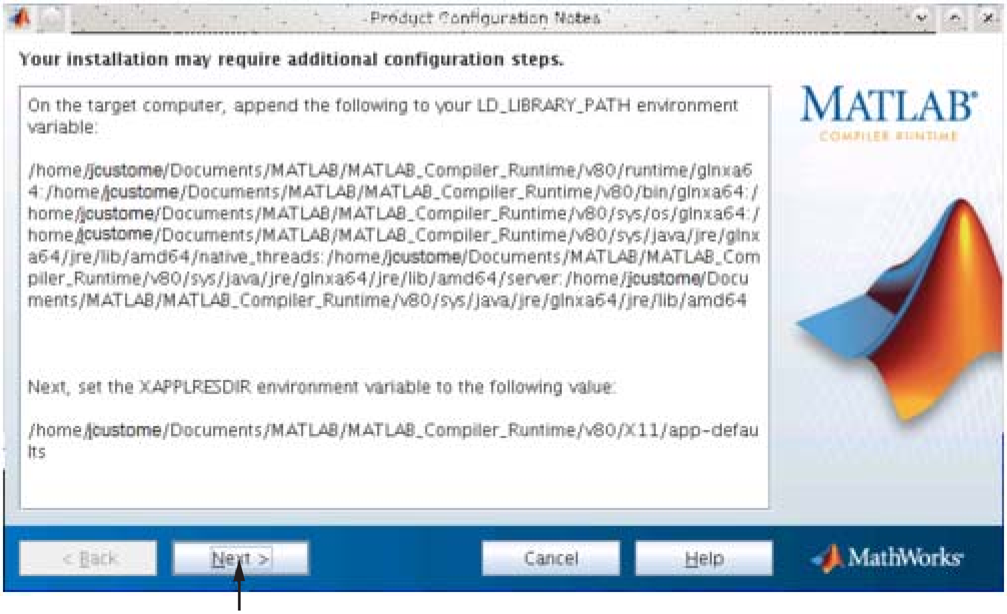

Confirmation
MATLAB
COMPILER RUNTIME
< Back
Install >
Cancel
MathWorks

Installation folder:
C:\Program Files\MATLAB\MATLAB Compiler Runtime

Products:
MATLAB Compiler Runtime

Click Install.

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



Your installation may require additional configuration steps.

On the target computer, append the following to your LD_LIBRARY_PATH environment variable:

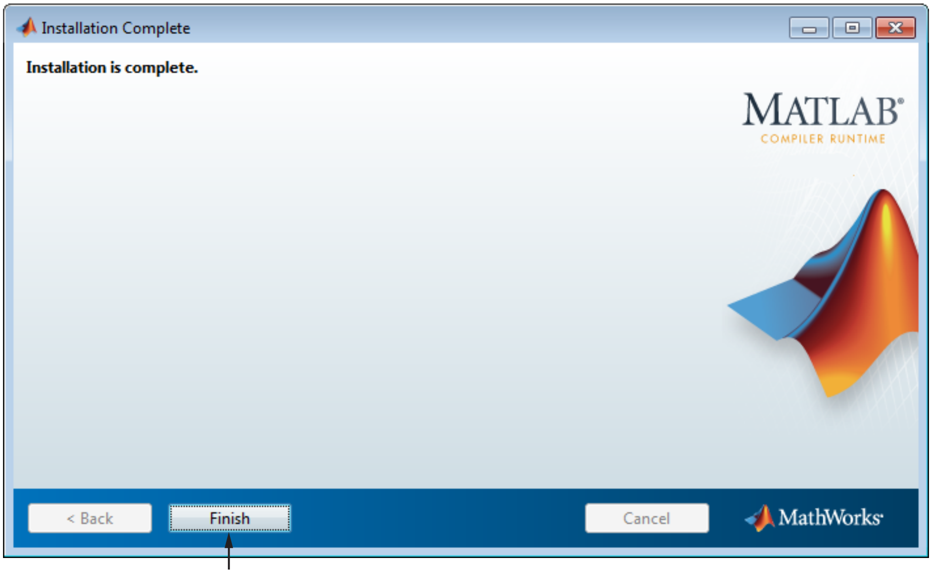
```
/home/jcustome/Documents/MATLAB/MATLAB_Compiler_Runtime/v80/runtime/glnxa6
4:/home/jcustome/Documents/MATLAB/MATLAB_Compiler_Runtime/v80/bin/glnxa64:/
home/jcustome/Documents/MATLAB/MATLAB_Compiler_Runtime/v80/sys/os/glnxa64:/
home/jcustome/Documents/MATLAB/MATLAB_Compiler_Runtime/v80/sys/java/jre/glnx
a64/jre/lib/amd64/native_threads:/home/jcustome/Documents/MATLAB/MATLAB_Com
piler_Runtime/v80/sys/java/jre/glnxa64/jre/lib/amd64/server:/home/jcustome/Docu
ments/MATLAB/MATLAB_Compiler_Runtime/v80/sys/java/jre/glnxa64/jre/lib/amd64
```

Next, set the XAPPLRESDIR environment variable to the following value:

```
/home/jcustome/Documents/MATLAB/MATLAB_Compiler_Runtime/v80/X11/app-defau
lts
```

Click Next.

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



Installation is complete.

Click Finish.

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

Installing the MCR and MATLAB on the Same Machine

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

You can, however, install the MCR for debugging purposes. See “Modifying the Path” on page 9-22.

Caution 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.

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

- **Windows**

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

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

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

- **UNIX**

  To run deployed components against the MCR install, on Linux, Linux x86-64, or the `<mcr_root>/runtime/<arch>` folder must appear on your `LD_LIBRARY_PATH` before `*matlabroot*/runtime/<arch>`. See "MCR Path Settings for Run-time Deployment" on page 16-4 for the platform-specific commands.

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

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

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

Installing Multiple MCRs on One Machine

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

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

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

Deploying a Recompiled Application. Always run your compiled applications with the version of the MCR that corresponds to the MATLAB version with which your application was built. If you upgrade your MATLAB Compiler software on your development machine and distribute the recompiled application to your users, you should also distribute the corresponding version of the MCR. Users should upgrade their MCR to the new version. If users need to maintain multiple versions of the MCR on their systems, refer to “Installing Multiple MCRs on One Machine” on page 9-22 for more information.

Installing the MCR Non-Interactively

To install the MCR without having to interact with the installer dialog boxes, use one of the MCR installer’s non-interactive modes:

- silent—the installer runs as a background task and does not display any dialog boxes
- automated—the installer displays the dialog boxes but does not wait for user interaction

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

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

Running the Installer in Silent Mode

To install the MCR in silent mode:

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

Note On Windows systems, **manually** extract the contents of the installer file.

---

2. Run the MCR installer, specifying the `-mode` option and `-argreeToLicense yes` on the command line.

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

Platform	Command
Windows	`setup -mode silent -argreeToLicense yes`
Linux	`./install -mode silent -argreeToLicense yes`
Mac OS X	`./install -mode silent -argreeToLicense yes`

Note If you do not include the `-argreeToLicense yes` the installer will not install the MCR.

---

3 View a log of the installation.

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

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

Customizing a Non-Interactive Installation

When run in one of the non-interactive modes, the installer will use the default values unless told to do otherwise. Like the MATLAB installer, the MCR installer accepts a number of command line options that modify the default installation properties.

Option	Description
`-destinationFolder`	Specifies where the MCR will be installed.
`-outputFile`	Specifies where the installation log file is written.
`-automatedModeTimeout`	Specifies how long, in milliseconds, that the dialog boxes are displayed when run in automatic mode.
`-inputFile`	Specifies an installer control file with the values for all of the above options.

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

Uninstalling the MCR

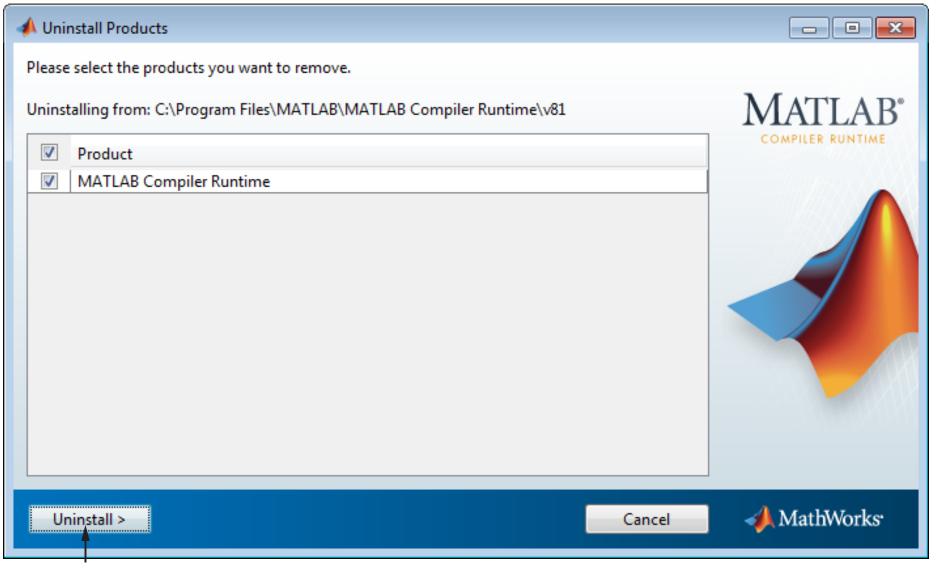
The method you use to uninstall the MCR from your computer varies depending on the type of computer.

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

Windows

1. Start the uninstaller. From the Windows Start menu, search for the **Add or Remove Programs** control panel, and double-click MATLAB Compiler Runtime in the list. You can also launch the MCR Uninstaller from the *`mcr_root`*`\uninstall\bin\`*`arch`* folder, where *`mcr_root`* is your MCR installation folder and *`arch`* is an architecture-specific folder, such as `win64`.

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



Uninstall Products

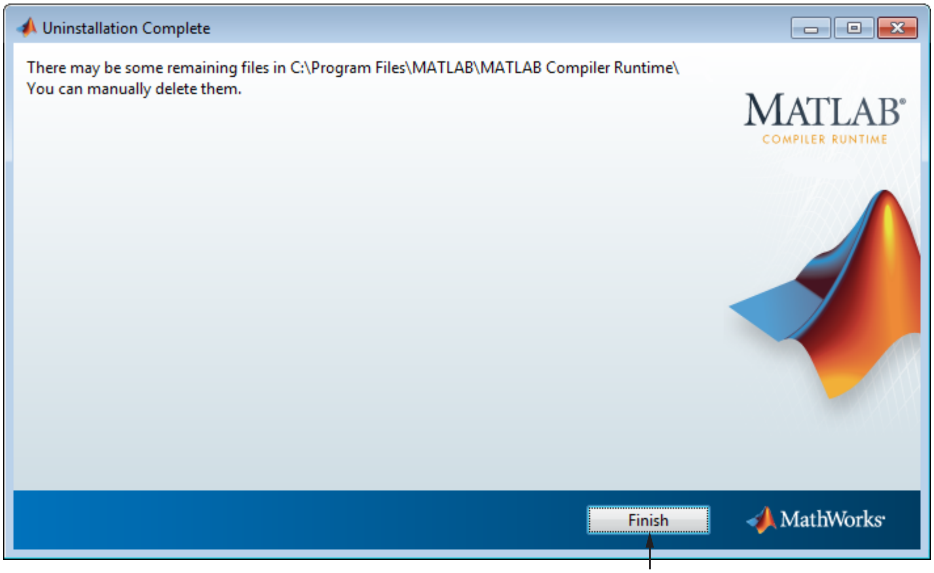
Please select the products you want to remove.

Uninstalling from: C:\Program Files\MATLAB\MATLAB Compiler Runtime\v81

☑	Product
☑	MATLAB Compiler Runtime

Click Uninstall.

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


Uninstallation Complete
MATLAB
COMPILER RUNTIME
Finish
MathWorks

There may be some remaining files in C:\Program Files\MATLAB\MATLAB Compiler Runtime\
You can manually delete them.

Click Finish.

Linux

1 Exit the application.

2 Enter this command at the Linux prompt:

```
rm -rf mcr_root
```

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

Mac

- Exit the application.

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

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

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

MCR Startup Options

Setting MCR Options

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

- Using the **Additional Runtime Settings** area of the compiler apps.
- Using the `mcc` command, specify the `-R` switch.

Using a Compiler App. In the **Additional Runtime Settings** area of the compiler apps, you can set the following options.

Note Not all options are available for all compilation targets.

Setting MCR Startup Options Using the Deployment Tool GUI

MCR Startup Option	This option...	Set the options by...
`-nojvm`	Disables the Java Virtual Machine, which is enabled by default.	Select the **No JVM** checkbox.

Setting MCR Startup Options Using the Deployment Tool GUI (Continued)

MCR Startup Option	This option...	Set the options by...
	This can help improve MCR performance.	
`-nodisplay`	On Linux, launches the MCR without display functionality.	In the **Settings** box, enter `-R -nodisplay`.
`-logfile`	Writes information about MCR startup to a logfile.	Select the **Create log file** checkbox. Enter the path to the logfile, including the logfile name, in the **Log File** box.

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

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

Setting -nojvm

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

Setting -nodisplay (Linux Only)

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

Setting -logfile

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

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

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

Retrieving MCR Startup Options (Shared Libraries Only)

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

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

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

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

Retrieving Information About MCR Startup Options.

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

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

Using the MCR User Data Interface

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

- You need to supply run-time profile information to a client running an application created with the Parallel Computing Toolbox. You supply and change profile information on a per-execution basis. For example, two instances of the same application may run simultaneously with different profiles. See "Deploy Applications Created Using Parallel Computing Toolbox" on page 9-37 for more information.
- You want to set up a global workspace, a global variable or variables that MATLAB and your client can access.
- You want to store the state of any variable or group of variables.

The API consists of:

- Two MATLAB functions callable from within deployed application MATLAB code
- Four external C functions callable from within deployed application wrapper code

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

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

MATLAB Functions

Use the MATLAB functions `getmcruserdata` and `setmcruserdata` from deployed MATLAB applications. They are loaded by default only in applications created with the MATLAB Compiler or builder products. See "Using the MCR User Data Interface" on page 9-32 for more information.

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

Setting MCR Data for Standalone Executables

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

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

```
parallelapp.exe -mcruserdata
                 ParallelProfile:config.settings
```

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

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

Setting and Retrieving MCR Data for Shared Libraries

As mentioned in "Using the MCR User Data Interface" on page 9-32, there are many possible scenarios for working with MCR Data. The most general scenario involves setting the MCR with specific data for later retrieval, as follows:

1. In your code, Include the MCR header file and the library header generated by MATLAB Compiler.
2. Properly initialize your application using `mclInitializeApplication`.
3. After creating your input data, write or "set" it to the MCR with `setmcruserdata` .
4. After calling functions or performing other processing, retrieve the new MCR data with `getmcruserdata`.
5. Free up storage memory in work areas by disposing of unneeded arrays with `mxDestroyArray`.
6. Shut down your application properly with `mclTerminateApplication`.

Displaying MCR Initialization Start-Up and Completion Messages For Users

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

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

You have the following options:

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

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

Some examples of different ways to invoke this option follow:

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

Best Practices

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

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

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

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

Deploy Applications Created Using Parallel Computing Toolbox

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

Compile and Deploy a Shared Library with the Parallel Computing Toolbox

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

1. Compile the shared library using the Deployment Tool.

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

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

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

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

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

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

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

---

1. On any Windows machine, run `mcrinstaller` function to obtain name of the MCR Installer executable.
2. Copy the entire MCR folder (the folder where MCR is installed) onto a network drive.
3. Copy the compiled application into a separate folder in the network drive and add the path `<mcr_root>\<ver>\runtime\<arch>` to all client machines. All network users can then execute the application.
4. Run `vcredist_x86.exe` on for 32-bit clients; run `vcredist_x64.exe` for 64-bit clients.
5. If you are using MATLAB Builder EX, register `mwcomutil.dll` and `mwcommgr.dll` on every client machine.

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

   To register the DLLs, at the DOS prompt enter

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

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

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

MATLAB Compiler Deployment Messages

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

Using MATLAB Compiler Generated DLLs in Windows Services

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

1. Create a file named `java.opts`.

2. Add the following line to the file:

   ```
   -Xrs
   ```

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

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

Reserving Memory for Deployed Applications with MATLAB Memory Shielding

In this section...

"What Is MATLAB Memory Shielding and When Should You Use It?" on page 9-42

"Requirements for Using MATLAB Memory Shielding" on page 9-43

"Invoking MATLAB Memory Shielding for Your Deployed Application" on page 9-43

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

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

- Your data set is large
- You are trying to compensate for the memory limitations inherent in a 32-bit Windows system
- The computer available to you has limited resources
- Network resources are restrictive

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

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

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

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

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

Requirements for Using MATLAB Memory Shielding

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

- Your deployed application is failing because it cannot find the proper amount of memory and not for another unrelated reason. As a best practice, let the operating system attempt to satisfy runtime memory requests, if possible. See "What Is MATLAB Memory Shielding and When Should You Use It?" on page 9-42 for examples of cases where you can benefit by using MATLAB Memory Shielding
- Your application runs on a Windows 32-bit system. While MATLAB Memory Shielding runs on 64-bit Windows systems without failing, it has no effect on your application.
- You are running with a standalone application or Windows executable. MATLAB Memory Shielding does not work with shared libraries, .NET components or Java components.
- You have run the MCR Installer on your system to get the MATLAB Compiler Runtime (MCR). The memory shielding feature is installed with the MCR.

Invoking MATLAB Memory Shielding for Your Deployed Application

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

Using the Command Line

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

The base command-line syntax is:

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

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

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

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

   - A higher level of protection does not always provide a larger size block and can occasionally cause start-up problems. Therefore, start with a lower level of protection and be conservative when raising the level of protection.
   - Use only memory shielding levels that guarantee a successful execution of your application. See the table MemShieldStarter Options on page 9-45 for more details on which shield options to choose.
   - Contact your system administrator for further advice on successfully running your application.

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

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

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

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

MemShieldStarter Options

Option	Description
`-help`	Invokes help for `MemShieldStarter`
`-gui`	Starts the Windows graphical interface for `MemShieldStarter.exe`. See "Using the GUI" on page 9-46 for more details.
`-shield` *`shield_level`*	See "Shield Level Options" on page 9-45.
`fully-qualified_application_path`	The fully qualified path to your user application
`user-defined_application_arguments`	Arguments passed to your user application. `MemShieldStarter.exe` only passes user arguments. It does not alter them.

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

- `none` — This value completely disables memory shielding. Use this value if your application fails to start successfully with the default (`-shield minimum`) option.
- `minimum` — The option defaults to this setting. Minimum shielding protects the range `0x50000000` to `0x70000000` during startup until just before processing `matlabrc`. This value ensures at least approximately 500 MB of contiguous memory available up to this point.

  When experimenting with a shielding level. start with `minimum`. To use the default, do not specify a shield option upon startup. If your application fails to start successfully using `minimum`, use `-shield none`. If your application starts successfully with the default value for *`shield_level`*, try using the `-shield medium` option to guarantee more memory.
- `medium` — This value protects the same range as `minimum`, `0x50000000` to `0x70000000`, but protects the range until just after startup processes `matlabrc`. It ensures that there is at least approximately 500 MB of

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

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

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

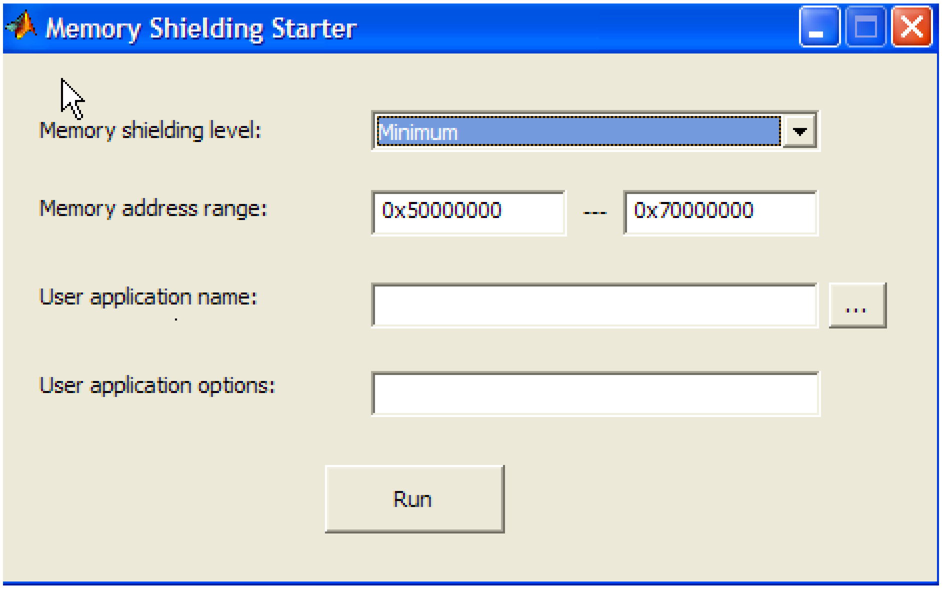
Using the GUI

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

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

   ```
   MemShieldStarter -gui
   ```

   The Memory Shielding Starter dialog box opens:



Memory shielding level: Minimum

Memory address range: 0x50000000 --- 0x70000000

User application name:

User application options:

2. Enter the appropriate values as described in MemShieldStarter Options on page 9-45. 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 9-45 for more details on appropriate shield options for various situations.

10

Distributing Code to an End User

Share MATLAB Code Using the MATLAB Compiler Runtime (MCR)

Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)

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

Install MATLAB Compiler Runtime (MCR)

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

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

The MCR installer does the following:

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

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

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

MCR Prerequisites

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

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

3. Do not install the MCR in MATLAB installation directories.
4. The MCR installer requires approximately 2 GB of disk space.

Add the MCR Installer to the Installer

This example shows how to include the MCR in the generated installer, using one of the compiler apps. The generated installer contains all files needed to run the standalone application or shared library built with MATLAB Compiler and properly lays they out on a target system.

1. On the **Packaging Options** section of the compiler interface, select one or both of the following options:
   - **Runtime downloaded from web** — This option builds an installer that invokes the MCR installer from the MathWorks Web site.
   - **Runtime included in package** — The option includes the MCR installer into the generated installer.
2. Click **Package**.
3. Distribute the installer as needed.

Install the MCR

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

If you are given an installer containing the compiled artifacts, then the MCR is installed along with the application or shared library. If you are given just the raw binary files, download the MCR installer from the Web and run the installer.

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

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

Compiler Commands

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

- "Command Overview" on page 11-2
- "Simplify Compilation Using Macros" on page 11-5
- "Invoke MATLAB Build Options" on page 11-8
- "MCR Component Cache and CTF Archive Embedding" on page 11-11
- "Explicitly Including a File for Compilation Using the %#function Pragma" on page 11-14
- "Use the mxArray API to Work with MATLAB Types" on page 11-16
- "Script Files" on page 11-17
- "Compiler Tips" on page 11-20

Command Overview

In this section...

"Compiler Options" on page 11-2

"Combining Options" on page 11-2

"Conflicting Options on the Command Line" on page 11-3

"Using File Extensions" on page 11-3

"Interfacing MATLAB Code to C/C++ Code" on page 11-4

Compiler Options

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

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

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

Macros are MathWorks supplied MATLAB Compiler options that simplify the more common compilation tasks. Instead of manually grouping several options together to perform a particular type of compilation, you can use a simple macro option. You can always use individual options to customize the compilation process to satisfy your particular needs. For more information on macros, see "Simplify Compilation Using Macros" on page 11-5.

Combining Options

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

```
mcc -mv myfun
```

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

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

This format is *not* valid:

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

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

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

Conflicting Options on the Command Line

If you use conflicting options, MATLAB Compiler resolves them from left to right, with the rightmost option taking precedence. For example, using the equivalencies in "Macro Options" on page 11-5,

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

is equivalent to:

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

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

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

Using File Extensions

The valid, recommended file extension for a file submitted to MATLAB Compiler is `.m`. Always specify the complete file name, including the `.m` extension, when compiling with `mcc` or you may encounter unpredictable results.

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

Interfacing MATLAB Code to C/C++ Code

To designate code to be compiled with C or C++, rewrite the C or C++ function as a MEX-file and call it from your application.

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

Code Proper Return Types From C and C++ Methods

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

Simplify Compilation Using Macros

In this section...

"Macro Options" on page 11-5

"Working With Macro Options" on page 11-5

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.

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

Working With Macro Options

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

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

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

-m Macro

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

Changing Macro Options

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

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

Specifying Default Macro Options

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

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

```
 mcc foo.m
```

to execute as though it were:

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

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

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

to behave as though the command were:

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

Invoke MATLAB Build Options

In this section…

"Specifying Full Path Names to Build MATLAB Code" on page 11-8

"Using Bundle Files to Build MATLAB Code" on page 11-9

Specifying Full Path Names to Build MATLAB Code

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

1. Breaks the full name into the corresponding path name and file names (`<path>` and `<file>`).
2. Replaces the full path name in the argument list with "`-I <path> <file>`".

Specifying Full Paths Names

For example:

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

would be treated as

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

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

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

would be equivalent to

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

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

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

Using Bundle Files to Build MATLAB Code

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

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

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

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

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

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

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

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

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

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

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

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

Bundle Files Available with MATLAB Compiler

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

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

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

MCR Component Cache and CTF Archive Embedding

In this section...

"Overriding Default Behavior" on page 11-12
"For More Information" on page 11-13

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

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

These variables allow you to specify the following:

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

Use the following environment variables to change these settings.

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

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

You can override this automatic embedding and extraction behavior by compiling with the `-C` option. See "Overriding Default Behavior" on page 11-12 for more information.

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

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

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

Overriding Default Behavior

To extract the CTF archive in a manner prior to R2008a, alongside the compiled shared library or executable, compile using the option "-C Do Not Embed CTF Archive by Default" on page 17-27.

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

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

For More Information

For more information about the CTF archive, see "Component Technology File (CTF Archive)" on page 7-6.

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

In this section...
"Using feval" on page 11-14
"Using %#function" on page 11-14

Using feval

In standalone mode, the pragma

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

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

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

Using %#function

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

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

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

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

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

Use the mxArray API to Work with MATLAB Types

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

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

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

Script Files

In this section...

"Converting Script MATLAB Files to Function MATLAB Files" on page 11-17

"Including Script Files in Deployed Applications" on page 11-18

Converting Script MATLAB Files to Function MATLAB Files

MATLAB provides two ways to package sequences of MATLAB commands:

- Function MATLAB files
- Script MATLAB files

Some things to remember about script and function MATLAB files:

- Variables used inside function MATLAB files are local to that function; you cannot access these variables from the MATLAB interpreter's workspace unless they are passed back by the function. By contrast, variables used inside script MATLAB files are shared with the caller's workspace; you can access these variables from the MATLAB interpreter command line.
- Variables that are declared as persistent in a MEX-file may not retain their values through multiple calls from MATLAB.

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

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

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

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

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

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

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

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

Including Script Files in Deployed Applications

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

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

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

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

```
function houdini_fcn
 houdini;
```

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

```
mcc -m houdini_fcn
```

Compiler Tips

In this section...

“Calling a Function from the Command Line” on page 11-20

“Using winopen in a Deployed Application” on page 11-21

“Using MAT-Files in Deployed Applications” on page 11-21

“Compiling a GUI That Contains an ActiveX Control” on page 11-21

“Debugging MATLAB® Compiler™ Generated Executables” on page 11-22

“Deploying Applications That Call the Java Native Libraries” on page 11-22

“Locating .fig Files in Deployed Applications” on page 11-22

“Terminating Figures by Force In a Standalone Application” on page 11-23

“Passing Arguments to and from a Standalone Application” on page 11-23

“Using Graphical Applications in Shared Library Targets” on page 11-25

“Using the VER Function in a Compiled MATLAB Application” on page 11-25

Calling a Function from the Command Line

You can make a MATLAB function into a standalone that is directly callable from the system command line. All the arguments passed to the MATLAB function from the system command line are strings. Two techniques to work with these functions are:

- Modify the original MATLAB function to test each argument and convert the strings to numbers.
- Write a wrapper MATLAB function that does this test and then calls the original MATLAB function.

  For example:

  ```
  function x=foo(a, b)
   if (ischar(a)), a = str2num(a), end;
   if (ischar(b)), b = str2num(b), end;
  ```

```
% The rest of your MATLAB code here...
```

You only do this if your function expects numeric input. If your function expects strings, there is nothing to do because that's the default from the command line.

Using winopen in a Deployed Application

`winopen` is a function that depends closely on a computer's underlying file system. You need to specify a path to the file you want to open, either absolute or relative.

When using `winopen` in deployed mode:

1. Verify that the file being passed to the command exists on the MATLAB path.

2. Use the `which` command to return an absolute path to the file.

3. Pass the path to `winopen`.

Using MAT-Files in Deployed Applications

To use a MAT-file in a deployed application, use the MATLAB Compiler `-a` option to include the file in the CTF archive. For more information on the `-a` option, see "-a Add to Archive" on page 17-25.

Compiling a GUI That Contains an ActiveX Control

When you save a GUI that contains ActiveX components, GUIDE creates a file in the current folder for each such component. The file name consists of the name of the GUI followed by an underscore (_) and `activex`*`n`*, where *`n`* is a sequence number. For example, if the GUI is named `ActiveXcontrol` then the file name would be `ActiveXcontrol_activex1`. The file name does not have an extension.

If you use MATLAB Compiler `mcc` command to compile a GUIDE-created GUI that contains an ActiveX component, you must use the `-a` option to add the ActiveX control files that GUIDE saved in the current folder to the CTF archive. Your command should be similar to

```
mcc -m mygui -a mygui_activex1
```

where `mygui_activex1` is the name of the file. If you have more than one such file, use a separate `-a` option for each file.

Debugging MATLAB Compiler Generated Executables

As of MATLAB Compiler 4, it is no longer possible to debug your entire program using a C/C++ debugger; most of the application is MATLAB code, which can only be debugged in MATLAB. Instead, run your code in MATLAB and verify that it produces the desired results. Then you can compile it. The compiled code will produce the same results.

Deploying Applications That Call the Java Native Libraries

If your application interacts with Java, you need to specify the search path for native method libraries by editing `librarypath.txt` and deploying it.

1. Copy `librarypath.txt` from *`matlabroot`*`/toolbox/local/librarypath.txt`.

2. Place `librarypath.txt` in `<mcr_root>/<ver>/toolbox/local`.

   `<mcr_root>` refers to the complete path where the MCR library archive files are installed on your machine.

3. Edit `librarypath.txt` by adding the folder that contains the native library that your application's Java code needs to load.

Locating .fig Files in Deployed Applications

MATLAB Compiler locates `.fig` files automatically when there is a MATLAB file with the same name as the `.fig` file in the same folder. If the `.fig` file does not follow this rule, it must be added with the `-a` option.

Terminating Figures by Force In a Standalone Application

The purpose of `mclWaitForFiguresToDie` is to block execution of a calling program as long as figures created in encapsulated MATLAB code are displayed. `mclWaitForFiguresToDie` takes no arguments. Your application can call `mclWaitForFiguresToDie` any time during execution. Typically you use `mclWaitForFiguresToDie` when:

- There are one or more figures you want to remain open.
- The function that displays the graphics requires user input before continuing.
- The function that calls the figures was called from `main()` in a console program.

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

Both MATLAB Builder NE and MATLAB Builder JA use `mclWaitForFiguresToDie` through the use of wrapper methods. See "Blocking Execution of a Console Application That Creates Figures" in the MATLAB Builder NE User's Guide and "Blocking Execution of a Console Application that Creates Figures" in the MATLAB Builder JA User's Guide for more details and code fragment examples.

Caution Use caution when calling the `mclWaitForFiguresToDie` function. Calling this function from an interactive program like Excel can hang the application. This function should be called *only* from console-based programs.

Passing Arguments to and from a Standalone Application

To pass input arguments to a MATLAB Compiler generated standalone application, you pass them just as you would to any console-based application. For example, to pass a file called `helpfile` to the compiled function called `filename`, use

```
filename helpfile
```

To pass numbers or letters (e.g., 1, 2, and 3), use

```
filename 1 2 3
```

Do not separate the arguments with commas.

To pass matrices as input, use

```
filename "[1 2 3]" "[4 5 6]"
```

You have to use the double quotes around the input arguments if there is a space in it. The calling syntax is similar to the `dos` command. For more information, see the MATLAB `dos` command.

The things you should keep in mind for your MATLAB file before you compile are:

- The input arguments you pass to your application from a system prompt are considered as string input. If, in your MATLAB code before compilation, you are expecting the data in different format, say double, you will need to convert the string input to the required format. For example, you can use `str2num` to convert the string input to numerical data. You can determine at run time whether or not to do this by using the `isdeployed` function. If your MATLAB file expects numeric inputs in MATLAB, the code can check whether it is being run as a standalone application. For example:

  ```
  function myfun (n1, n2)
  if (isdeployed)
   n1 = str2num(n1);
   n2 = str2num(n2);
  end
  ```

- You cannot return back values from your standalone application to the user. The only way to return values from compiled code is to either display it on the screen or store it in a file. To display your data on the screen, you either need to unsuppress (do not use semicolons) the commands whose results yield data you want to return to the screen or, use the `disp` command to display the value. You can then redirect these outputs to other applications using output redirection (> operator) or pipes (only on UNIX systems).

Passing Arguments to a Double-Clickable Application

On Windows, if you want to run the standalone application by double-clicking it, you can create a batch file that calls this standalone application with the specified input arguments. Here is an example of the batch file:

```
rem main.bat file that calls sub.exe with input parameters
sub "[1 2 3]" "[4 5 6]"
@echo off
pause
```

The last two lines of code keep your output on the screen until you press a key. If you save this file as `main.bat`, you can run your code with the specified arguments by double-clicking the `main.bat` icon.

Using Graphical Applications in Shared Library Targets

When deploying a GUI as a shared library to a C/C++ application, use `mclWaitForFiguresToDie` to display the GUI until it is explicitly terminated.

Using the VER Function in a Compiled MATLAB Application

When you use the `VER` function in a compiled MATLAB application, it will perform with the same functionality as if you had called it from MATLAB. However, be aware that when using `VER` in a compiled MATLAB application, only version information for toolboxes which the compiled application uses will be displayed.

12

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 12-2
- "Deploying Standalone Applications" on page 12-3

Introduction

Suppose you want to create an application that calculates the rank of a large magic square. One way to create this application is to code the whole application in C or C++; however, this would require writing your own magic square, rank, and singular value routines. An easier way to create this application is to write it as one or more MATLAB files, taking advantage of the power of MATLAB and its tools.

You can create MATLAB applications that take advantage of the mathematical functions of MATLAB, yet do not require that end users own MATLAB. Standalone applications are a convenient way to package the power of MATLAB and to distribute a customized application to your users.

The source code for standalone applications consists either entirely of MATLAB files or some combination of MATLAB files and MEX-files.

MATLAB Compiler takes your MATLAB files and generates a standalone executable that allows your MATLAB application to be invoked from outside of interactive MATLAB.

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

Deploying Standalone Applications

In this section…

"Compiling the Application" on page 12-3

"Testing the Application" on page 12-3

"Deploying the Application" on page 12-4

"Running the Application" on page 12-6

Compiling the Application

This example takes a MATLAB file, `magicsquare.m`, and creates a standalone application, `magicsquare`.

1. Copy the file `magicsquare.m` from

   ```
   matlabroot\extern\examples\compiler
   ```

   to your work folder.

2. To compile the MATLAB code, use

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

   The `-m` option tells MATLAB Compiler (`mcc`) to generate a standalone application. The `-v` option (verbose) displays the compilation steps throughout the process and helps identify other useful information such as which third-party compiler is used and what environment variables are referenced.

   This command creates the standalone application called `magicsquare` and additional files. The Windows platform appends the `.exe` extension to the name. See the table in "Standalone Executable" on page 8-2 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 "MCR Path Settings for Run-time Deployment" on page 16-4

2. Run the standalone application from the system prompt (shell prompt on UNIX or DOS prompt on Windows) by typing the application name.

   ```
   magicsquare.exe 4                               (On Windows)
   magicsquare 4                                   (On UNIX)
   magicsquare.app/Contents/MacOS/magicsquare      (On Maci64)
   ```

   The results are:

   ```
   ans =
       16     2     3    13
        5    11    10     8
        9     7     6    12
        4    14    15     1
   ```

Deploying the Application

You can distribute a MATLAB Compiler generated standalone application to any target machine that has the same operating system as the machine on which the application was compiled.

For example, if you want to deploy an application to a Windows machine, you must use MATLAB Compiler to build the application on a Windows machine. If you want to deploy the same application to a UNIX machine, you must use MATLAB Compiler on the same UNIX platform and completely rebuild the application. To deploy an application to multiple platforms requires MATLAB and MATLAB Compiler licenses on all the desired platforms.

Windows

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

Component	Description
MCR Installer	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform. Run the `mcrinstaller` command to obtain name of executable.
`magicsquare`	Application; `magicsquare.exe` for Windows

UNIX

Distribute and package your standalone application on UNIX by packaging the following files and distributing them to the deployment machine.

Component	Description
MCR Installer	MATLAB Compiler Runtime library archive; platform-dependent file that must correspond to the end user's platform. Run the `mcrinstaller` command to obtain name of the binary.
`magicsquare`	Application

Maci64

Distribute and package your standalone application on 64-bit Macintosh by copying, tarring, or zipping as described in the following table.

Component	Description
MCR Installer	MATLAB Compiler Runtime library archive; platform-dependent file that must correspond to the end user's platform. Run the `mcrinstaller` command to obtain name of the binary.
`magicsquare`	Application
`magicsquare.app`	Application bundle Assuming `foo` is a folder within your current folder:

`magicsquare.app`

- Distribute by copying:

```
cp -R myapp.app foo
```

- Distribute by tarring:

```
tar -cvf myapp.tar myapp.app
  cd foo
  tar -xvf../ myapp.tar
```

- Distribute by zipping:

```
zip -ry myapp myapp.app
  cd foo
  unzip ..\myapp.zip
```

Running the Application

These steps describe the process that end users must follow to install and run the application on their machines.

Preparing Your Machines

Install the MCR by running the `mcrinstaller` command to obtain name of the executable or binary. For more information on running the MCR Installer utility and modifying your system paths, see "Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)" on page 10-2.

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 11-23.

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
```

13

Libraries

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

- "Introduction" on page 13-2
- "Addressing mwArrays Above the 2 GB Limit" on page 13-3
- "Integrate C Shared Libraries" on page 13-4
- "Integrate C++ Shared Libraries" on page 13-18
- "Call MATLAB® Compiler™ API Functions (mcl*) from C/C++ Code" on page 13-23
- "About Memory Management and Cleanup" on page 13-35

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 `int`s. When `MX_COMPAT_32` is not defined, `mwSize` and `mwIndex` are defined to `size_t`. This can lead to compiler warnings and errors with respect to signed and unsigned mismatches.

In R2008b, all support for MX_COMPAT_32 was removed.

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

Integrate C Shared Libraries

In this section…

"C Shared Library Wrapper" on page 13-4

"C Shared Library Example" on page 13-4

"Calling a Shared Library" on page 13-12

"Using C Shared Libraries On a Mac OS X System" on page 13-17

C Shared Library Wrapper

The C library wrapper option allows you to create a shared library from an arbitrary set of MATLAB files on both Microsoft Windows and UNIX operating systems. MATLAB Compiler generates a wrapper file, a header file, and an export list. The header file contains all of the entry points for all of the compiled MATLAB functions. The export list contains the set of symbols that are exported from a C shared library.

Note Even if you are not producing a shared library, you must use `-W lib` or `-W cpplib` when including any MATLAB Compiler generated code into a larger application.

C Shared Library Example

This example takes several MATLAB files and creates a C shared library. It also includes a standalone driver application to call the shared library.

Building the Shared Library

1. Copy the following files from *matlabroot*`\extern\examples\compiler` to your work directory:

```
matlabroot\extern\examples\compiler\addmatrix.m
matlabroot\extern\examples\compiler\multiplymatrix.m
matlabroot\extern\examples\compiler\eigmatrix.m
matlabroot\extern\examples\compiler\matrixdriver.c
```

Note `matrixdriver.c` contains the standalone application's `main` function.

---

2 To create the shared library, enter the following command on a single line:

```
mcc -B csharedlib:libmatrix addmatrix.m multiplymatrix.m
eigmatrix.m -v
```

The `-B csharedlib` option is a bundle option that expands into

```
-W lib:<libname> -T link:lib
```

The `-W lib:<libname>` option tells MATLAB Compiler to generate a function wrapper for a shared library and call it `libname`. The `-T link:lib` option specifies the target output as a shared library. Note the directory where the product puts the shared library because you will need it later on.

Writing a Driver Application for a Shared Library

Note You must call `mclInitializeApplication` once at the beginning of your driver application, before calling any other MathWorks functions (for example, before accessing an `MWArray`). See "Calling a Shared Library" on page 13-12 for complete details on using a MATLAB Compiler generated library in your application.

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

All programs that call MATLAB Compiler generated shared libraries have roughly the same structure:

1. Declare variables and process/validate input arguments.
2. Call `mclInitializeApplication`, and test for success. This function sets up the global MCR state and enables the construction of MCR instances.

Caution Avoid issuing `cd` commands from the driver application prior to calling `mclInitializeApplication`. Failure to do so can cause a failure in MCR initialization.

---

3. Call, once for each library, `<libraryname>Initialize`, to create the MCR instance required by the library.
4. Invoke functions in the library, and process the results. (This is the main body of the program.)

Note If your driver application displays MATLAB figure windows, you should include a call to `mclWaitForFiguresToDie(NULL)` before calling the `Terminate` functions and `mclTerminateApplication` in the following two steps.

---

5. Call, once for each library, `<lib>Terminate`, to destroy the associated MCR.

Caution `<lib>Terminate` will bring down enough of the MCR address space that the same library (or any other library) cannot be initialized. Issuing a `<lib>Initialize` call after a `<lib>Terminate` call causes unpredictable results. Instead, use the following structure:

```
...code...
mclInitializeApplication();
lib1Initialize();
lib2Initialize();

lib1Terminate();
lib2Terminate();
mclTerminateApplication();
...code...
```

---

6. Call `mclTerminateApplication` to free resources associated with the global MCR state.

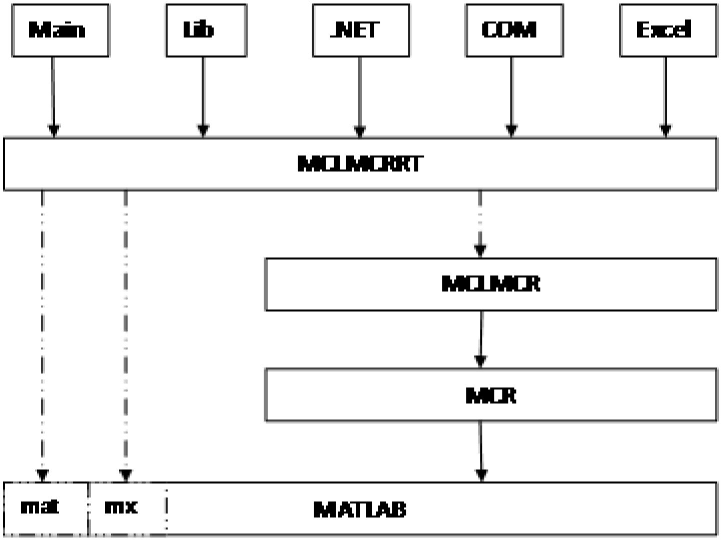
7. Clean up variables, close files, etc., and exit.

This example uses `matrixdriver.c` as the driver application.

How the `mclmcrrt` Proxy Layer Handles Loading of Libraries. All application and software components generated by MATLAB Compiler and the associated builder products need to link against only one MathWorks library, `mclmcrrt`. This library provides a proxy API for all the public functions in MATLAB libraries used for matrix operations, MAT-file access, utility and memory management, and application runtime. The `mclmcrrt` library lies between deployed components and these other version-dependent libraries, providing the following functionality:

- Ensures that multiple versions of the MATLAB Compiler Runtime can coexist
- Provides a layer of indirection
- Ensures applications are thread-safe
- Loads the dependent (re-exported) libraries dynamically

The relationship between `mclmcrrt` and other MATLAB libraries is shown in the following figure.



The MCLMCRRT Proxy Layer

In the figure, solid arrows designate static linking and dotted arrows designate dynamic linking. The figure illustrates how the `mclmcrrt` library layer sits above the `mclmcr` and `mcr` libraries. The `mclmcr` library contains the run-time functionality of the deployed components. The `mcr` module ensures each deployed component runs in its own context at runtime. The `mclmcrrt` proxy layer, in addition to loading the `mclmcr`, also dynamically loads the `MX` and `MAT` modules, primarily for `mxArray` manipulation. For more information, see the MathWorks Support database and search for information on the `MSVC` shared library.

Caution Deployed applications must only link to the `mclmcrrt` proxy layer library (`mclmcrrt.lib` on Windows, `mclmcrrt.so` on Linux, and `mclmcrrt.dylib` on Macintosh). Do not link to the other libraries shown in the figure, such as `mclmcr`, `libmx`, and so on.

Compiling the Driver Application

To compile the driver code, `matrixdriver.c`, you use your C/C++ compiler. Execute the following `mbuild` command that corresponds to your development platform. This command uses your C/C++ compiler to compile the code.

```
mbuild matrixdriver.c libmatrix.lib    (Windows)
mbuild matrixdriver.c -L. -lmatrix -I. (UNIX)
```

Note This command assumes that the shared library and the corresponding header file created from step 2 are in the current working directory.

On UNIX, if this is not the case, replace the "`.`" (dot) following the `-L` and `-I` options with the name of the directory that contains these files, respectively.

On Windows, if this is not the case, specify the full path to `libmatrix.lib`, and use a `-I` option to specify the directory containing the header file.

This generates a standalone application, `matrixdriver.exe`, on Windows, and `matrixdriver`, on UNIX.

Difference in the Exported Function Signature. The interface to the `mlf` functions generated by MATLAB Compiler from your MATLAB file routines has changed from earlier versions of the product. The generic signature of the exported `mlf` functions is

- MATLAB functions with no return values

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

- MATLAB functions with at least one return value

  ```
  bool MW_CALL_CONV
        mlf<function-name>(int number_of_return_values,
  <list_of_pointers_to_return_variables>,
  <list_of_input_variables>);
  ```

Refer to the header file generated for your library for the exact signature of the exported function. For example, in the library created in the previous section, the signature of the exported addmatrix function is

```
void mlfAddmatrix(int nlhs,mxArray **a,mxArray *a1,mxArray *a2);
```

Testing the Driver Application

These steps test your standalone driver application and shared library on your development machine.

Note Testing your application on your development machine is an important step to help ensure that your application is compilable. To verify that your application compiled properly, you must test all functionality that is available with the application. If you receive an error message similar to `Undefined function` or `Attempt to execute script` *`script_name`* `as a function`, it is likely that the application will not run properly on deployment machines. Most likely, your CTF archive is missing some necessary functions. Use `-a` to add the missing functions to the archive and recompile your code.

---

1. To run the standalone application, add the directory containing the shared library that was created in step 2 in "Building the Shared Library" on page 13-4 to your dynamic library path.
2. Update the path for your platform by following the instructions in "MCR Path Settings for Development and Testing" on page 16-2.
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
```

Deploying Standalone Applications That Call MATLAB Compiler Based Shared Libraries

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

Component	**Description**
MCR Installer	Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform. Run the `mcrinstaller` command to obtain name of executable.
`matrixdriver`	Application; `matrixdriver.exe` for Windows `matrixdriver.app` for Mac OS X (bundle directory structure must be deployed)
`libmatrix`	Shared library; extension varies by platform. Extensions are: • Windows — `.dll` • Linux, Linux x86-64 — `.so` • Mac OS X — `.dylib`

Note You can distribute a MATLAB Compiler generated standalone application to any target machine that has the same operating system as the machine on which the application was compiled. If you want to deploy the same application to a different platform, you must use MATLAB Compiler on the different platform and completely rebuild the application.

Deploying Shared Libraries to Be Used with Other Projects

To distribute the shared library for use with an external application, you need to distribute the following.

Component	Description
MCR Installer	(Windows) Self-extracting MATLAB Compiler Runtime library utility; platform-dependent file that must correspond to the end user's platform. Run the `mcrinstaller` command to obtain name of executable.
`libmatrix`	Shared library; extension varies by platform, for example, `DLL` on Windows
`libmatrix.h`	Library header file

Calling a Shared Library

At runtime, there is an MCR instance associated with each individual shared library. Consequently, if an application links against two MATLAB Compiler generated shared libraries, there will be two MCR instances created at runtime.

You can control the behavior of each MCR instance by using MCR options. The two classes of MCR options are global and local. Global MCR options are identical for each MCR instance in an application. Local MCR options may differ for MCR instances.

To use a shared library, you must use these functions:

- `mclInitializeApplication`

- `mclTerminateApplication`

Initializing and Terminating Your Application with mclInitializeApplication and mclTerminateApplication

`mclInitializeApplication` allows you to set the global MCR options. They apply equally to all MCR instances. You must set these options before creating your first MCR instance.

These functions are necessary because some MCR options such as whether or not to start Java, whether or not to use the MATLAB JIT feature, and so on, are set when the first MCR instance starts and cannot be changed by subsequent instances of the MCR.

Caution You must call `mclInitializeApplication` once at the beginning of your driver application. You must make this call before calling any other MathWorks functions. This also applies to shared libraries. Avoid calling `mclInitializeApplication` multiple times in an application as it will cause the application to hang.

After you call `mclTerminateApplication`, you may not call `mclInitializeApplication` again. No MathWorks functions may be called after `mclTerminateApplication`.

Avoid issuing `cd` commands from the driver application prior to calling `mclInitializeApplication`. Failure to do so can cause a failure in MCR initialization.

The function signatures are

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

mclInitializeApplication. Takes an array of strings (options) that you set (the same options that can be provided to `mcc` via the `-R` option) and a count of the number of options (the length of the option array). Returns `true` for success and `false` for failure.

mclTerminateApplication. Takes no arguments and can *only* be called after all MCR instances have been destroyed. Returns `true` for success and `false` for failure.

The following code example is from `matrixdriver.c`:

```
int main(){

    mxArray *in1, *in2; /* Define input parameters */
    mxArray *out = NULL;/* and output parameters to pass to
                           the library functions */

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

    /* Call library initialization routine and make sure that
       the library was initialized properly */
    mclInitializeApplication(NULL,0);
    if (!libmatrixInitialize()){
        fprintf(stderr,"could not initialize the library
                        properly\n");
        return -1;
    }

    /* Create the input data */
    in1 = mxCreateDoubleMatrix(3,3,mxREAL);
    in2 = mxCreateDoubleMatrix(3,3,mxREAL);
    memcpy(mxGetPr(in1), data, 9*sizeof(double));
    memcpy(mxGetPr(in2), data, 9*sizeof(double));

    /* Call the library function */
    mlfAddmatrix(1, &out, in1, in2);
    /* Display the return value of the library function */
    printf("The value of added matrix is:\n");
    display(out);
    /* Destroy return value since this variable will be reused
       in next function call. Since we are going to reuse the
       variable, we have to set it to NULL. Refer to MATLAB
       Compiler documentation for more information on this. */
    mxDestroyArray(out); out=0;
    mlfMultiplymatrix(1, &out, in1, in2);
    printf("The value of the multiplied matrix is:\n");
    display(out);
    mxDestroyArray(out); out=0;
    mlfEigmatrix(1, &out, in1);
    printf("The Eigen value of the first matrix is:\n");
    display(out);
    mxDestroyArray(out); out=0;

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

    /* Free the memory created */
    mxDestroyArray(in1); in1=0;
    mxDestroyArray(in2); in2 = 0;
    mclTerminateApplication();
    return 0;
}
```

Caution `mclInitializeApplication` can only be called *once* per application. Calling it a second time generates an error, and will cause the function to return `false`. This function must be called before calling any C MEX function or MAT-file API function.

Using a Shared Library

To use a MATLAB Compiler generated shared library in your application, you must perform the following steps:

1. Include the generated header file for each library in your application. Each MATLAB Compiler generated shared library has an associated header file named *`libname`*`.h`, where *`libname`* is the library's name that was passed in on the command line when the library was compiled.

2. Initialize the MATLAB libraries by calling the `mclInitializeApplication` API function. You must call this function once per application, and it must be called before calling any other MATLAB API functions, such as C-MEX functions or C MAT-file functions. `mclInitializeApplication` must be called before calling any functions in a MATLAB Compiler generated

shared library. You may optionally pass in application-level options to this function. `mclInitializeApplication` returns a Boolean status code. A return value of `true` indicates successful initialization, and `false` indicates failure.

3. For each MATLAB Compiler generated shared library that you include in your application, call the library's initialization function. This function performs several library-local initializations, such as unpacking the CTF archive, and starting an MCR instance with the necessary information to execute the code in that archive. The library initialization function will be named *`libname`*`Initialize()`, where *`libname`* is the library's name that was passed in on the command line when the library was compiled. This function returns a Boolean status code. A return value of `true` indicates successful initialization, and `false` indicates failure.

Note On Windows, if you want to have your shared library call a MATLAB shared library (as generated by MATLAB Compiler), the MATLAB library initialization function (e.g., `<libname>Initialize, <libname>Terminate, mclInitialize, mclTerminate`) cannot be called from your shared library during the `DllMain(DLL_ATTACH_PROCESS)` call. This applies whether the intermediate shared library is implicitly or explicitly loaded. You must place the call somewhere after `DllMain()`.

---

4. Call the exported functions of each library as needed. Use the C MEX API to process input and output arguments for these functions.

5. When your application no longer needs a given library, call the library's termination function. This function frees the resources associated with its MCR instance. The library termination function will be named `<libname>Terminate()`, where `<libname>` is the library's name that was passed in on the command line when the library was compiled. Once a library has been terminated, that library's exported functions should not be called again in the application.

6. When your application no longer needs to call any MATLAB Compiler generated libraries, call the `mclTerminateApplication` API function. This function frees application-level resources used by the MCR. Once you call

this function, no further calls can be made to MATLAB Compiler generated libraries in the application.

Restrictions When using MATLAB Function loadlibrary

You can not use the MATLAB function `loadlibrary` inside of MATLAB to load a C shared library built with MATLAB Compiler.

For more information about using `loadlibrary`, see “Load MATLAB Libraries using loadlibrary” on page 7-16.

Using C Shared Libraries On a Mac OS X System

For information on using C shared libraries on a Macintosh system, see “Using C/C++ Shared Libraries on a Mac OS X System” on page 13-21.

Integrate C++ Shared Libraries

In this section...
"C++ Shared Library Wrapper" on page 13-18
"C++ Shared Library Example" on page 13-18

C++ Shared Library Wrapper

The C++ library wrapper option allows you to create a shared library from an arbitrary set of MATLAB files. MATLAB Compiler generates a wrapper file and a header file. The header file contains all of the entry points for all of the compiled MATLAB functions.

C++ Shared Library Example

This example rewrites the previous C shared library example using C++. The procedure for creating a C++ shared library from MATLAB files is identical to the procedure for creating a C shared library, except you use the `cpplib` wrapper. Enter the following command on a single line:

```
mcc -W cpplib:libmatrixp -T link:lib addmatrix.m multiplymatrix.m eigmatrix.m -v
```

The `-W cpplib:<libname>` option tells MATLAB Compiler to generate a function wrapper for a shared library and call it `<libname>`. The `-T link:lib` option specifies the target output as a shared library. Note the directory where the product puts the shared library because you will need it later.

Writing the Driver Application

Note Due to name mangling in C++, you must compile your driver application with the same version of your third-party compiler that you use to compile your C++ shared library.

In the C++ version of the `matrixdriver` application `matrixdriver.cpp`, arrays are represented by objects of the class `mwArray`. Every `mwArray` class object contains a pointer to a MATLAB array structure. For this reason, the attributes of an `mwArray` object are a superset of the attributes of a MATLAB array. Every MATLAB array contains information about the size and shape of the array (i.e., the number of rows, columns, and pages) and either one or two arrays of data. The first array stores the real part of the array data and the second array stores the imaginary part. For arrays with no imaginary part, the second array is not present. The data in the array is arranged in column-major, rather than row-major, order.

Caution Avoid issuing `cd` commands from the driver application prior to calling `mclInitializeApplication`. Failure to do so can cause a failure in MCR initialization.

Caution to Mac users: when running the `matrixdriver` example, invoke `mclInitializeApplication` prior to `mclRunMain`.

For information about how the compiler uses a proxy layer for the libraries that an application must link, see "How the `mclmcrrt` Proxy Layer Handles Loading of Libraries" on page 13-7.

Compiling the Driver Application

To compile the `matrixdriver.cpp` driver code, you use your C++ compiler. By executing the following `mbuild` command that corresponds to your development platform, you will use your C++ compiler to compile the code.

```
mbuild matrixdriver.cpp libmatrixp.lib          (Windows)
mbuild matrixdriver.cpp -L. -lmatrixp -I.       (UNIX)
```

Note This command assumes that the shared library and the corresponding header file are in the current working directory.

On Windows, if this is not the case, specify the full path to `libmatrixp.lib`, and use a `-I` option to specify the directory containing the header file.

On UNIX, if this is not the case, replace the "`.`" (dot) following the `-L` and `-I` options with the name of the directory that contains these files, respectively.

Incorporating a C++ Shared Library into an Application

To incorporate a C++ shared library into your application, you will, in general, follow the steps in "Using a Shared Library" on page 13-15. There are two main differences to note when using a C++ shared library:

- Interface functions use the `mwArray` type to pass arguments, rather than the `mxArray` type used with C shared libraries.
- C++ exceptions are used to report errors to the caller. Therefore, all calls must be wrapped in a `try-catch` block.

Exported Function Signature

The C++ shared library target generates two sets of interfaces for each MATLAB function. The first set of exported interfaces is identical to the `mlx` signatures that are generated in C shared libraries. The second set of interfaces is the C++ function interfaces. The generic signature of the exported C++ functions is as follows:

MATLAB Functions with No Return Values.

```
bool MW_CALL_CONV <function-name>(<mwArray_lists>);
```

MATLAB Functions with at Least One Return Value.

```
bool MW_CALL_CONV <function-name>(int <number_of_return_values>,
     <mxArray_pointers>, <mwArray_lists>);
```

In this case, *`mwArray_lists`* represents a comma-separated list of type `const mwArray&` and *`mxArray_pointers`* represents a comma-separated list of pointers of type `mwArray&`. For example, in the `libmatrix` library, the C++ interfaces to the `addmatrix` MATLAB function is generated as:

```
void addmatrix(int nargout, mwArray& a , const mwArray& a1,
               const mwArray& a2);
```

Error Handling

C++ interface functions handle errors during execution by throwing a C++ exception. Use the `mwException` class for this purpose. Your application can catch `mwExceptions` and query the `what()` method to get the error message. To correctly handle errors when calling the C++ interface functions, wrap each call inside a `try-catch` block.

```
 try
{
  ...
  (call function)
  ...
}
catch (const mwException& e)
{
  ...
  (handle error)
  ...
}
```

The `matrixdriver.cpp` application illustrates the typical way to handle errors when calling the C++ interface functions.

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

To use a MATLAB Compiler generated library on a Mac OS X system, a separate thread needs to be created.

The initialization of the shared library and subsequent calls to that library's functions is performed by this thread. The function `mclRunMain`, provided by MATLAB Compiler, takes care of the thread creation process.

The main thread of the application is the thread that calls your driver program's `main()` function. The body of your `main()` function should call the `mclRunMain` function, passing to it the address of another function. This function should contain the library initialization routines and necessary calls to the shared library generated by MATLAB Compiler.

The `matrixdriver.c` example illustrates this procedure. This example rewrites the C shared library example from this chapter for use on Mac OS X. Follow the same procedure as in "C Shared Library Example" on page 13-4 to build and run this application.

The Mac version of the `matrixdriver` application differs from the version on other platforms. The `run_main()` function performs the basic tasks of initialization, calling the library's functions, and termination. Compare this function with the `matrixdriver` `main()` function on other platforms, listed in the earlier example.

Working with C++ Shared Libraries and Sparse Arrays

The MATLAB Compiler API includes static factory methods for working with sparse arrays.

For a complete list of the methods, see "C++ Utility Classes" on page D-4.

Call MATLAB Compiler API Functions (mcl*) from C/C++ Code

In this section...

"Functions in the Shared Library" on page 13-23

"Type of Application" on page 13-23

"Structure of Programs That Call Shared Libraries" on page 13-25

"Library Initialization and Termination Functions" on page 13-26

"Print and Error Handling Functions" on page 13-27

"Functions Generated from MATLAB Files" on page 13-29

"Retrieving MCR State Information While Using Shared Libraries" on page 13-34

Functions in the Shared Library

A shared library generated by MATLAB Compiler contains at least seven functions. There are three generated functions to manage library initialization and termination, one each for printed output and error messages, and two generated functions for each MATLAB file compiled into the library.

To generate the functions described in this section, first copy `sierpinski.m`, `main_for_lib.c`, `main_for_lib.h`, and `triangle.c` from *`matlabroot`*`\extern\examples\compiler` into your directory, and then execute the appropriate MATLAB Compiler command.

Type of Application

For a C Application on Windows

```
mcc -W lib:libtriangle -T link:lib sierpinski.m
mbuild triangle.c main_for_lib.c libtriangle.lib
```

For a C Application on UNIX

```
mcc -W lib:libtriangle -T link:lib sierpinski.m
mbuild triangle.c main_for_lib.c -L. -ltriangle -I.
```

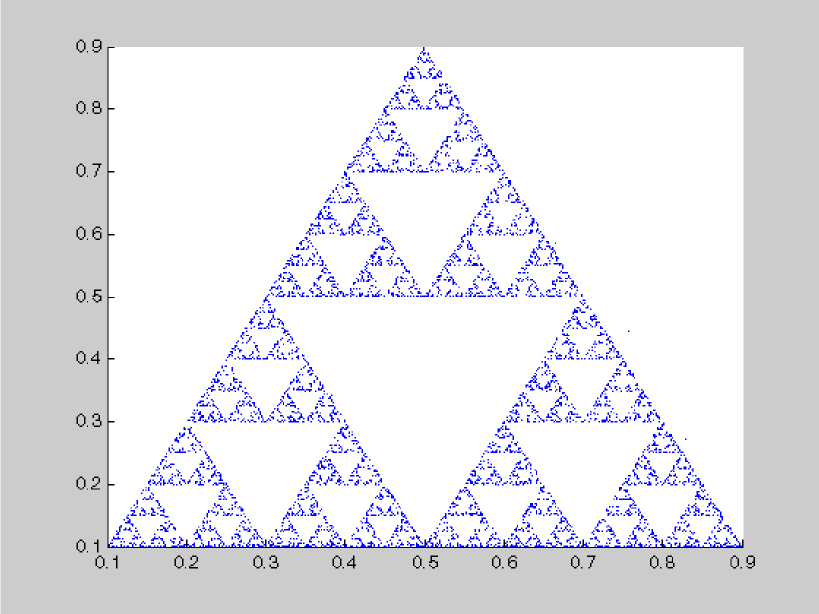
For a C++ Application on Windows

```
mcc -W cpplib:libtrianglep -T link:lib sierpinski.m
mbuild triangle.cpp main_for_lib.c libtrianglep.lib
```

For a C++ Application on UNIX

```
mcc -W cpplib:libtriangle -T link:lib sierpinski.m
mbuild triangle.cpp main_for_lib.c -L. -ltriangle -I.
```

These commands create a main program named `triangle`, and a shared library named `libtriangle`. The library exports a single function that uses a simple iterative algorithm (contained in `sierpinski.m`) to generate the fractal known as Sierpinski's Triangle. The main program in `triangle.c` or `triangle.cpp` can optionally take a single numeric argument, which, if present, specifies the number of points used to generate the fractal. For example, `triangle 8000` generates a diagram with 8,000 points.



In this example, MATLAB Compiler places all of the generated functions into the generated file `libtriangle.c` or `libtriangle.cpp`.

Structure of Programs That Call Shared Libraries

All programs that call MATLAB Compiler generated shared libraries have roughly the same structure:

1. Declare variables and process/validate input arguments.
2. Call `mclInitializeApplication`, and test for success. This function sets up the global MCR state and enables the construction of MCR instances.
3. Call, once for each library, `<libraryname>Initialize`, to create the MCR instance required by the library.
4. Invoke functions in the library, and process the results. (This is the main body of the program.)

5. Call, once for each library, `<libraryname>Terminate`, to destroy the associated MCR.

6. Call `mclTerminateApplication` to free resources associated with the global MCR state.

7. Clean up variables, close files, etc., and exit.

To see these steps in an actual example, review the main program in this example, `triangle.c`.

Library Initialization and Termination Functions

The library initialization and termination functions create and destroy, respectively, the MCR instance required by the shared library. You must call the initialization function before you invoke any of the other functions in the shared library, and you should call the termination function after you are finished making calls into the shared library (or you risk leaking memory).

There are two forms of the initialization function and one type of termination function. The simpler of the two initialization functions takes no arguments; most likely this is the version your application will call. In this example, this form of the initialization function is called `libtriangleInitialize`.

```
bool libtriangleInitialize(void)
```

This function creates an MCR instance using the default print and error handlers, and other information generated during the compilation process.

However, if you want more control over how printed output and error messages are handled, you may call the second form of the function, which takes two arguments.

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

By calling this function, you can provide your own versions of the print and error handling routines called by the MCR. Each of these routines has the same signature (for complete details, see “Print and Error Handling

Functions" on page 13-27). By overriding the defaults, you can control how output is displayed and, for example, whether or not it goes into a log file.

Note Before calling either form of the library initialization routine, you must first call `mclInitializeApplication` to set up the global MCR state. See "Calling a Shared Library" on page 13-12 for more information.

On Microsoft Windows platforms, MATLAB Compiler generates an additional initialization function, the standard Microsoft DLL initialization function `DllMain`.

```
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason,
                    void *pv)
```

The generated `DllMain` performs a very important service; it locates the directory in which the shared library is stored on disk. This information is used to find the CTF archive, without which the application will not run. If you modify the generated `DllMain` (not recommended), make sure you preserve this part of its functionality.

Library termination is simple.

```
void libtriangleTerminate(void)
```

Call this function (once for each library) before calling `mclTerminateApplication`.

Print and Error Handling Functions

By default, MATLAB Compiler generated applications and shared libraries send printed output to standard output and error messages to standard error. MATLAB Compiler generates a default print handler and a default error handler that implement this policy. If you'd like to change this behavior, you must write your own error and print handlers and pass them in to the appropriate generated initialization function.

You may replace either, both, or neither of these two functions. The MCR sends all regular output through the print handler and all error output through the error handler. Therefore, if you redefine either of these functions, the MCR will use your version of the function for all the output that falls into class for which it invokes that handler.

The default print handler takes the following form.

```
static int mclDefaultPrintHandler(const char *s)
```

The implementation is straightforward; it takes a string, prints it on standard output, and returns the number of characters printed. If you override or replace this function, your version must also take a string and return the number of characters “handled.” The MCR calls the print handler when an executing MATLAB file makes a request for printed output, e.g., via the MATLAB function `disp`. The print handler does not terminate the output with a carriage return or line feed.

The default error handler has the same form as the print handler.

```
static int mclDefaultErrorHandler(const char *s)
```

However, the default implementation of the print handler is slightly different. It sends the output to the standard error output stream, but if the string does not end with carriage return, the error handler adds one. If you replace the default error handler with one of your own, you should perform this check as well, or some of the error messages printed by the MCR will not be properly formatted.

Caution The error handler, despite its name, does not handle the actual errors, but rather the message produced after the errors have been caught and handled inside the MCR. You cannot use this function to modify the error handling behavior of the MCR -- use the `try` and `catch` statements in your MATLAB files if you want to control how a MATLAB Compiler generated application responds to an error condition.

Note If you provide alternate C++ implementations of either `mclDefaultPrintHandler` or `mclDefaultErrorHandler`, then functions must be declared `extern "C"`. For example:

```
extern "C" int myPrintHandler(const char *s);
```

Functions Generated from MATLAB Files

For each MATLAB file specified on the MATLAB Compiler command line, the product generates two functions, the `mlx` function and the `mlf` function. Each of these generated functions performs the same action (calls your MATLAB file function). The two functions have different names and present different interfaces. The name of each function is based on the name of the first function in the MATLAB file (`sierpinski`, in this example); each function begins with a different three-letter prefix.

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

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

---

mlx Interface Function

The function that begins with the prefix `mlx` takes the same type and number of arguments as a MATLAB MEX-function. (See the External Interfaces documentation for more details on MEX-functions.) The first argument, `nlhs`, is the number of output arguments, and the second argument, `plhs`, is a pointer to an array that the function will fill with the requested number of return values. (The “`lhs`” in these argument names is short for “left-hand side” -- the output variables in a MATLAB expression are those on the left-hand side of the assignment operator.) The third and fourth parameters are the number of inputs and an array containing the input variables.

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

mlf Interface Function

The second of the generated functions begins with the prefix `mlf`. This function expects its input and output arguments to be passed in as individual variables rather than packed into arrays. If the function is capable of producing one or more outputs, the first argument is the number of outputs requested by the caller.

```
void mlfSierpinski(int nargout, mxArray** x, mxArray** y,
                   mxArray* iterations, mxArray* draw)
```

In both cases, the generated functions allocate memory for their return values. If you do not delete this memory (via `mxDestroyArray`) when you are done with the output variables, your program will leak memory.

Your program may call whichever of these functions is more convenient, as they both invoke your MATLAB file function in an identical fashion. Most programs will likely call the `mlf` form of the function to avoid managing the extra arrays required by the `mlx` form. The example program in `triangle.c` calls `mlfSierpinski`.

```
mlfSierpinski(2, &x, &y, iterations, draw);
```

In this call, the caller requests two output arguments, `x` and `y`, and provides two inputs, `iterations` and `draw`.

If the output variables you pass in to an `mlf` function are not NULL, the `mlf` function will attempt to free them using `mxDestroyArray`. This means that you can reuse output variables in consecutive calls to `mlf` functions without worrying about memory leaks. It also implies that you must pass either `NULL` or a valid MATLAB array for all output variables or your program will fail because the memory manager cannot distinguish between a non-initialized (invalid) array pointer and a valid array. It will try to free a pointer that is not NULL -- freeing an invalid pointer usually causes a segmentation fault or similar fatal error.

Using varargin and varargout in a MATLAB Function Interface

If your MATLAB function interface uses `varargin` or `varargout`, you must pass them as cell arrays. For example, if you have `N varargins`, you need to create one cell array of size `1-by-N`. Similarly, `varargouts` are returned back as one cell array. The length of the `varargout` is equal to the number of return values specified in the function call minus the number of actual variables passed. As in the MATLAB software, the cell array representing `varagout` has to be the last return variable (the variable preceding the first input variable) and the cell array representing `varargins` has to be the last formal parameter to the function call.

For information on creating cell arrays, refer to the C MEX function interface in the External Interfaces documentation.

For example, consider this MATLAB file interface:

```
[a,b,varargout] = myfun(x,y,z,varargin)
```

The corresponding C interface for this is

```
void mlfMyfun(int numOfRetVars, mxArray **a, mxArray **b,
              mxArray **varargout, mxArray *x, mxArray *y,
              mxArray *z, mxArray *varargin)
```

In this example, the number of elements in `varargout` is `(numOfRetVars - 2)`, where `2` represents the two variables, `a` and `b`, being returned. Both `varargin` and `varargout` are single row, multiple column cell arrays.

Caution The C++ shared library interface does not support `varargin` with zero (0) input arguments. Calling your program using an empty `mwArray` results in the compiled library receiving an empty array with `nargin = 1`. The C shared library interface allows you to call `mlfFOO(NULL)` (the compiled MATLAB code interprets this as `nargin=0`). However, calling `FOO((mwArray)NULL)` with the C++ shared library interface causes the compiled MATLAB code to see an empty array as the first input and interprets `nargin=1`.

For example, compile some MATLAB code as a C++ shared library using `varargin` as the MATLAB function's list of input arguments. Have the MATLAB code display the variable `nargin`. Call the library with function *`FOO`*`()` and it won't compile, producing this error message:

```
... 'FOO' : function does not take 0 arguments
```

Call the library as:

```
  mwArray junk;
  FOO(junk);
```

or

```
 FOO((mwArray)NULL);
```

At runtime, `nargin=1`. In MATLAB, *`FOO`*`()` is `nargin=0` and `FOO([])` is `nargin=1`.

C++ Interfaces for MATLAB Functions Using varargin and varargout. The C++ `mlx` interface for MATLAB functions does not change even if the functions use `varargin` or `varargout`. However, the C++ function interface (the second set of functions) changes if the MATLAB function is using `varargin` or `varargout`.

For examples, view the generated code for various MATLAB function signatures that use `varargin` or `varargout`.

Note For simplicity, only the relevant part of the generated C++ function signature is shown in the following examples.

---

function varargout = foo(varargin)

For this MATLAB function, the following C++ overloaded functions are generated:

```
No input no output:
void foo()

Only inputs:
void foo(const mwArray& varargin)

Only outputs:
void foo(int nargout, mwArray& varargout)

Most generic form that has both inputs and outputs:
void foo(int nargout, mwArray& varargout,
                       const mwArray& varargin)
```

function varargout = foo(i1, i2, varargin)

For this MATLAB function, the following C++ overloaded functions are generated:

```
Most generic form that has outputs and all the inputs
void foo(int nargout, mwArray& varargout, const
                             mwArray& i1, const
                             mwArray& i2, const
                              mwArray& varargin)

Only inputs:
void foo(const mwArray& i1,
      const mwArray& i2, const mwArray& varargin)
```

function [o1, o2, varargout] = foo(varargin)

For this MATLAB function, the following C++ overloaded functions are generated:

```
Most generic form that has all the outputs and inputs
void foo(int nargout, mwArray& o1, mwArray& o2,
                            mwArray& varargout,
                        const mwArray& varargin)


Only outputs:
void foo(int nargout, mwArray& o1, mwArray& o2,
                                   mwArray& varargout)
```

function [o1, o2, varargout] = foo(i1, i2, varargin)

For this MATLAB function, the following C++ overloaded function is generated:

```
Most generic form that has all the outputs and
                                      all the inputs
void foo(int nargout, mwArray& o1, mwArray& o2,

mwArray& varargout,
         const mwArray& i1, const mwArray& i2,
                       const mwArray& varargin)
```

Retrieving MCR State Information While Using Shared Libraries

When using shared libraries (note this does not apply to standalone applications), you may call functions to retrieve specific information from MCR state. For details, see “MCR Startup Options” on page 9-28.

About Memory Management and Cleanup

In this section...

"Overview" on page 13-35
"Passing mxArrays to Shared Libraries" on page 13-35

Overview

Generated C++ code provides consistent garbage collection via the object destructors and the MCR's internal memory manager optimizes to avoid heap fragmentation.

If memory constraints are still present on your system, try preallocating arrays in MATLAB. This will reduce the number of calls to the memory manager, and the degree to which the heap fragments.

Passing mxArrays to Shared Libraries

When an `mxArray` is created in an application which uses the MCR, it is created in the managed memory space of the MCR.

Therefore, it is very important that you never create `mxArrays` (or call any other MathWorks function) before calling `mclInitializeApplication`.

It is safe to call `mxDestroyArray` when you no longer need a particular `mxArray` in your code, even when the input has been assigned to a persistent or global variable in MATLAB. MATLAB uses reference counting to ensure that when `mxDestroyArray` is called, if another reference to the underlying data still exists, the memory will not be freed. Even if the underlying memory is not freed, the `mxArray` passed to `mxDestroyArray` will no longer be valid.

For more information about `mclInitializeApplication` and `mclTerminateApplication`, see "Calling a Shared Library" on page 13-12.

For more information about `mxArray`, see "Use the mxArray API to Work with MATLAB Types" on page 11-16.

14

Troubleshooting

- "Introduction" on page 14-2
- "Common Issues" on page 14-4
- "Failure Points and Possible Solutions" on page 14-5
- "Troubleshooting mbuild" on page 14-16
- "MATLAB® Compiler™" on page 14-18
- "Deployed Applications" on page 14-22

Introduction

MATLAB Compiler software converts your MATLAB programs into self-contained applications and software components and enables you to share them with end users who do not have MATLAB installed. MATLAB Compiler takes MATLAB applications (MATLAB files, MEX-files, and other MATLAB executable code) as input and generates redistributable standalone applications or shared libraries. The resulting applications and components are platform specific.

Another use of MATLAB Compiler is to build C or C++ shared libraries (DLLs on Windows) from a set of MATLAB files. You can then write C or C++ programs that can call the functions in these libraries. The typical workflow for building a shared library is to compile your MATLAB code on a development machine, write a C/C++ driver application, build an executable from the driver code, test the resulting executable on that machine, and deploy the executable and MCR to a test or customer machine without MATLAB.

Compiling a shared library is very similar to compiling an executable. The command line differs as shown:

```
mcc -B csharedlib:hellolib hello.m
```

or

```
mcc -B cpplib:hellolib hello.m
```

Once you have compiled a shared library, the next step is to create a driver application that initializes and terminates the shared library as well as invokes method calls. This driver application can be compiled and linked with your shared library with the `mbuild` command. For example:

```
mbuild helloapp.c hellolib.lib
```

or

```
mbuild helloapp.cpp hellolib.lib
```

The only header file that needs to be included in your driver application is the one generated by your `mcc` command (`hellolib.h` in the above example). See “Integrate C Shared Libraries” on page 13-4 and “Integrate C++ Shared

Libraries” on page 13-18 for examples of how to correctly access a shared library.

Common Issues

Some of the most common issues encountered when using MATLAB Compiler generated standalone executables or shared libraries are:

- **Compilation fails with an error message.** This can indicate a failure during any one of the internal steps involved in producing the final output.
- **Compilation succeeds but the application does not execute because required DLLs are not found.** All shared libraries required for your standalone executable or shared library are contained in the MATLAB Compiler Runtime (MCR). Installing the MCR is required for any of the deployment targets.
- **Compilation succeeds, and the resultant file starts to execute but then produces errors and/or generates a crash dump.**
- **The compiled program executes on the machine where it was compiled but not on other machines.**
- **The compiled program executes on some machines and not others.**

If any of these issues apply to you, search "Failure Points and Possible Solutions" on page 14-5 for common solutions.

Failure Points and Possible Solutions

In this section...

"How to Use this Section" on page 14-5

"Does the Failure Occur During Compilation?" on page 14-5

"Does the Failure Occur When Testing Your Application?" on page 14-9

"Does the Failure Occur When Deploying the Application to End Users?" on page 14-13

How to Use this Section

Use the following list of questions to diagnose some of the more common issues associated with using MATLAB Compiler software.

Does the Failure Occur During Compilation?

You typically compile your MATLAB code on a development machine, test the resulting executable on that machine, and deploy the executable and MATLAB Compiler Runtime (MCR) to a test or customer machine without MATLAB. The compilation process performs dependency analysis on your MATLAB code, creates an encrypted archive of your code and required toolbox code, generates wrapper code, and compiles the wrapper code into an executable. If your application fails to build an executable, the following questions may help you isolate the problem.

Is your selected compiler supported by MATLAB Compiler?

See the current list of supported compilers at `http://www.mathworks.com/support/compilers/current_release/`.

Are error messages produced at compile time?

See error messages in "MATLAB® Compiler™" on page 14-18.

Did you compile with the verbose flag?

Compilation can fail in MATLAB because of errors encountered by the system compiler when the generated wrapper code is compiled into an executable. Additional errors and warnings are printed when you use the verbose flag as such:

```
mcc -mv myApplication.m
```

In this example, `-m` tells MATLAB Compiler to create a standalone application and `-v` tells MATLAB Compiler and other processors to display messages about the process.

Are you compiling within or outside of MATLAB?

`mcc` can be invoked from the operating system command line or from the MATLAB prompt. When you run `mcc` inside the MATLAB environment, MATLAB will modify environment variables in its environment as necessary so `mcc` will run. Issues with `PATH`, `LD_LIBRARY_PATH`, or other environment variables seen at the operating system command line are often not seen at the MATLAB prompt. The environment that MATLAB uses for `mcc` can be listed at the MATLAB prompt. For example:

```
>>!set
```

lists the environment on Windows platforms.

```
>>!printenv
```

lists the environment on UNIX platforms. Using this path allows you to use `mcc` from the operating system command line.

Does a simple read/write application such as "Hello World" compile successfully?

Sometimes applications won't compile because of MEX-file issues, other toolboxes, or other dependencies. Compiling a `helloworld` application can determine if MATLAB Compiler is correctly set up to produce any executable. For example, try compiling:

```
function helloworld
```

```
 disp('hello world')
```

with:

```
>>mcc -mv helloworld.m
```

Have you tried to compile any of the examples in MATLAB Compiler help?

The source code for all examples is provided with MATLAB Compiler and is located in `*matlabroot*\extern\examples\compiler`, where `*matlabroot*` is the root folder of your MATLAB installation.

Did the MATLAB code compile successfully before this failure?

The three most common reasons for MATLAB code to stop compiling are:

- A change in the selection of the system compiler — It is possible to inadvertently change the system compiler for versions of MATLAB that store preferences in a common folder. For example, MATLAB 7.0.1 (R14SP1) and MATLAB 7.0.4 (R14SP2) store their preferences in the same folder. Changing the system compiler in R14SP1 will also change the system compiler in R14SP2.
- An upgrade to MATLAB that didn't include an upgrade to MATLAB Compiler — The versions of MATLAB Compiler and MATLAB must be the same in order to work together. It is possible to see conflicts in installations where the MATLAB installation is local and the MATLAB Compiler installation is on a network or vice versa.

Are you receiving errors when trying to compile a standalone executable?

If you are not receiving error messages to help you debug your standalone application, write an application to display the warnings or error messages to the console.

Are you receiving errors when trying to compile a shared library?

Errors at compile time can indicate issues with either `mcc` or `mbuild`. For troubleshooting `mcc` issues, see the previous section on compile time issues. It is recommended that your driver application be compiled and linked using `mbuild`. `mbuild` can be executed with the `-v` switch to provide additional information on the compilation process. If you receive errors at this stage, ensure that you are using the correct header files and/or libraries produced by `mcc`, in your C or C++ driver. For example:

```
mcc -B csharedlib:hellolib hello.m
```

produces `hellolib.h`, which is required to be included in your C/C++ driver, and `hellolib.lib` or `hellolib.so`, which is required on the `mbuild` command line.

Is your MATLAB object failing to load?

If your MATLAB object fails to load, it is typically a result of the MCR not finding required class definitions.

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

```
%#function class_constructor
```

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

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

MathWorks recommends and supports using `mbuild` to compile your driver application. `mbuild` is designed and tested to correctly build driver applications. It will ensure that all MATLAB header files are found by the C/C++ compiler, and that all necessary libraries are specified and found by the linker.

Are you trying to compile your driver application using Microsoft Visual Studio or another IDE?

If using an IDE, in addition to linking to the generated export library, you need to include an additional dependency to `mclmcrrt.lib`. This library is provided for all supported third-party compilers in *`matlabroot`*`\extern\lib\`*`vendor-name`*.

Are you importing the correct versions of import libraries?

If you have multiple versions of MATLAB installed on your machine, it is possible that an older or incompatible version of the library is referenced. Ensure that the only MATLAB library that you are linking to is `mclmcrrt.lib` and that it is referenced from the appropriate vendor folder. Do not reference libraries as `libmx` or `libut`. In addition, verify that your library path references the version of MATLAB that your shared library was built with.

Are you able to compile the matrixdriver example?

Typically, if you cannot compile the examples in the documentation, it indicates an issue with the installation of MATLAB or your system compiler. See "Integrate C Shared Libraries" on page 13-4 and "Integrate C++ Shared Libraries" on page 13-18 for these examples.

Do you get the `MATLAB:I18n:InconsistentLocale` Warning?

The warning message

```
MATLAB:I18n:InconsistentLocale - The system locale setting,
system_locale_name, is different from the user locale
setting, user_locale_name
```

indicates a mismatch between locale setting on Microsoft Windows systems. This may affect your ability to display certain characters. For information about changing the locale settings, see your operating system Help.

Does the Failure Occur When Testing Your Application?

After you have successfully compiled your application, the next step is to test it on a development machine and deploy it on a target machine. Typically the target machine does not have a MATLAB installation and requires that the MATLAB Compiler Runtime (MCR) be installed. A distribution includes all of the files that are required by your application to run, which include the executable, CTF archive and the MCR.

See “Deploying to Developers” on page 9-3 and “Deploying to End Users” on page 9-6 for information on distribution contents for specific application types and platforms.

Test the application on the development machine by running the application against the MCR shipped with MATLAB Compiler. This will verify that library dependencies are correct, that the CTF archive can be extracted and that all MATLAB code, MEX—files and support files required by the application have been included in the archive. If you encounter errors testing your application, the questions in the column to the right may help you isolate the problem.

Are you able to execute the application from MATLAB?

On the development machine, you can test your application’s execution by issuing `!`*`application-name`* at the MATLAB prompt. If your application executes within MATLAB but not from outside, this can indicate an issue with the system `PATH` variable.

Does the application begin execution and result in MATLAB or other errors?

Ensure that you included all necessary files when compiling your application (see the `readme.txt` file generated with your compilation for more details).

Functions that are called from your main MATLAB file are automatically included by MATLAB Compiler; however, functions that are not explicitly called, for example through EVAL, need to be included at compilation using the `-a` switch of the `mcc` command. Also, any support files like `.mat`, `.txt`, or `.html` files need to be added to the archive with the `-a` switch. There is a limitation on the functionality of MATLAB and associated toolboxes that can be compiled. Check the documentation to see that the functions used in your application’s MATLAB files are valid. Check the file `mccExcludedFiles.log` on the development machine. This file lists all functions called from your application that cannot be compiled.

Does the application emit a warning like "MATLAB file may be corrupt"?

See the listing for this error message in "MATLAB® Compiler™" on page 14-18 for possible solutions.

Do you have multiple MATLAB versions installed?

Executables generated by MATLAB Compiler are designed to run in an environment where multiple versions of MATLAB are installed. Some older versions of MATLAB may not be fully compatible with this architecture.

On Windows, ensure that the *`matlabroot`*`\runtime\win32|win64` of the version of MATLAB in which you are compiling appears ahead of *`matlabroot`*`\runtime\win32|win64` of other versions of MATLAB installed on the `PATH` environment variable on your machine.

Similarly, on UNIX, ensure that the dynamic library paths (`LD_LIBRARY_PATH` on Linux) match. Do this by comparing the outputs of `!printenv` at the MATLAB prompt and `printenv` at the shell prompt. Using this path allows you to use `mcc` from the operating system command line.

If you are testing a standalone executable or shared library and driver application, did you install the MCR?

All shared libraries required for your standalone executable or shared library are contained in the MATLAB Compiler Runtime (MCR). Installing the MCR is required for any of the deployment targets.

Do you receive an error message about a missing DLL?

Error messages indicating missing DLLs such as `mclmcrrt7x.dll` or `mclmcrrt7x.so` are generally caused by incorrect installation of the MCR. It is also possible that the MCR is installed correctly, but that the `PATH`,`LD_LIBRARY_PATH`, or `DYLD_LIBRARY_PATH` variables are set incorrectly. For information on installing the MCR on a deployment machine, refer to "Working with the MCR" on page 9-14.

Caution Do not solve these problems by moving libraries or other files within the MCR folder structure. The run-time system is designed to accommodate different MCR versions operating on the same machine. The folder structure is an important part of this feature.

Are you receiving errors when trying to run the shared library application?

Calling MATLAB Compiler generated shared libraries requires correct initialization and termination in addition to library calls themselves. For information on calling shared libraries, see "Call MATLAB® Compiler™ API Functions (mcl*) from C/C++ Code" on page 13-23.

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

- Ensure that the calls to `mclinitializeApplication` and `*libname*Initialize` are successful. The first function enables construction of MCR instances. The second creates the MCR instance required by the library named `*libname*`. If these calls are not successful, your application will not execute.
- Do not use any `mw-` or `mx`-functions before calling `mclinitializeApplication`. This includes static and global variables that are initialized at program start. Referencing `mw-` or `mx`-functions before initialization results in undefined behavior.
- Do not re-initialize (call `mclinitializeApplication`) after terminating it with `mclTerminateApplication`. The `mclinitializeApplication` and`*libname*Initialize` functions should be called only once.
- Ensure that you do not have any library calls after `mclTerminateApplication`.
- Ensure that you are using the correct syntax to call the library and its functions.

Does your system's graphics card support the graphics application?

In situations where the existing hardware graphics card does not support the graphics application, you should use software OpenGL. OpenGL libraries are visible for an application by appending `matlab/sys/opengl/lib/arch` to the `LD_LIBRARY_PATH`. For example:

```
setenv LD_LIBRARY_PATH
$LD_LIBRARY_PATH:matlab/sys/opengl/lib/arch
```

Is OpenGL properly installed on your system?

When searching for OpenGL libraries, the MCR first looks on the system library path. If OpenGL is not found there, it will use the `LD_LIBRARY_PATH` environment variable to locate the libraries. If you are getting failures due to the OpenGL libraries not being found, you can append the location of the OpenGL libraries to the `LD_LIBRARY_PATH` environment variable. For example:

```
setenv LD_LIBRARY_PATH
$LD_LIBRARY_PATH:matlab/sys/opengl/lib/arch
```

Does the Failure Occur When Deploying the Application to End Users?

After the application is working on the test machine, failures can be isolated in end-user deployment. The end users of your application need to install the MATLAB Compiler Runtime (MCR) on their machines. The MCR includes a set of shared libraries that provides support for all features of MATLAB. If your application fails during end-user deployment, the following questions in the column to the right may help you isolate the problem.

Note There are a number of reasons why your application might not deploy to end users, after running successfully in a test environment. For a detailed list of guidelines for writing MATLAB code that can be consumed by end users, see "Write Deployable MATLAB Code" on page 7-10

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 9-14 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 9-14.

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 9-14.

Caution Do not solve these problems by moving libraries or other files within the MCR folder structure. The run-time system is designed to accommodate different MCR versions operating on the same machine. The folder structure is an important part of this feature.

Do you have write access to the directory the application is installed in?

The first operation attempted by a compiled application is extraction of the CTF archive. If the archive is not extracted, the application cannot access the compiled MATLAB code and the application fails. If the application has write access to the installation folder, a subfolder named *`application-name`*`_mcr` is created the first time the application is run. After this subfolder is created, the application no longer needs write access for subsequent executions.

Are you executing a newer version of your application?

When deploying a newer version of an executable, both the executable needs to be redeployed, since it also contains the embedded CTF file. The CTF file is keyed to a specific compilation session. Every time an application is recompiled, a new, matched CTF file is created. As above, write access is required to expand the new CTF file. Deleting the existing `*application-name*_mcr` folder and running the new executable will verify that the application can expand the new CTF file.

Troubleshooting mbuild

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

Options File Not Writable. When you run `mbuild -setup`, `mbuild` makes a copy of the appropriate options file and writes some information to it. If the options file is not writable, you are asked if you want to overwrite the existing options file. If you choose to do so, the existing options file is copied to a new location and a new options file is created.

Directory or File Not Writeable. If a destination folder or file is not writable, ensure that the permissions are properly set. In certain cases, make sure that the file is not in use.

mbuild Generates Errors. If you run `mbuild filename` and get errors, it may be because you are not using the proper options file. Run `mbuild -setup` to ensure proper compiler and linker settings.

Compiler and/or Linker Not Found. On Windows, if you get errors such as `unrecognized command` or `file not found`, make sure the command-line tools are installed and the path and other environment variables are set correctly in the options file. For Microsoft Visual Studio®, for example, make sure to run `vcvars32.bat` (MSVC 6.x and earlier) or `vsvars32.bat` (MSVC 8.x and later).

mbuild Not a Recognized Command. If `mbuild` is not recognized, verify that *`matlabroot`*`\bin` is in your path. On UNIX, it may be necessary to rehash.

mbuild Works from the Shell But Not from MATLAB (UNIX). If the command

```
mcc -m hello
```

works from the UNIX command prompt but not from the MATLAB prompt, you may have a problem with your `.cshrc` file. When MATLAB launches a new C shell to perform compilations, it executes the `.cshrc` script. If this script causes unexpected changes to the `PATH` environment variable, an error may occur. You can test this before starting MATLAB by performing the following:

```
setenv SHELL /bin/sh
```

If this works correctly, then you should check your `.cshrc` file for problems setting the `PATH` environment variable.

Cannot Locate Your Compiler (Windows). If `mbuild` has difficulty locating your installed compilers, it is useful to know how it finds compilers. `mbuild` automatically detects your installed compilers by first searching for locations specified in the following environment variables:

- `MSVCDIR` for Microsoft Visual C++, Version 6.0 or 8.0

Next, `mbuild` searches the Windows registry for compiler entries.

Internal Error when Using mbuild -setup (Windows). Some antivirus software packages may conflict with the `mbuild -setup` process. If you get an error message during `mbuild -setup` of the following form

```
mex.bat: internal error in sub get_compiler_info(): don't
recognize <string>
```

then you need to disable your antivirus software temporarily and rerun `mbuild -setup`. After you have successfully run the `setup` option, you can re-enable your antivirus software.

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

MATLAB Compiler

Typically, problems that occur when building standalone applications involve `mbuild`. However, it is possible that you may run into some difficulty with MATLAB Compiler. A good source for additional troubleshooting information for the product is the MATLAB Compiler Product Support page at the MathWorks Web site.

libmwlapack: load error: stgsy2_. This error occurs when a customer has both the R13 and the R14 version of MATLAB or MCR/MGL specified in the folder path and the R14 version fails to load because of a `lapack` incompatibility.

Licensing Problem. If you do not have a valid license for MATLAB Compiler , you will get an error message similar to the following when you try to access MATLAB Compiler:

```
Error: Could not check out a Compiler License:
No such feature exists.
```

If you have a licensing problem, contact MathWorks. A list of contacts at MathWorks is provided at the beginning of this document.

`loadlibrary` usage (MATLAB `loadlibrary` command). The following are common error messages encountered when attempting to compile the MATLAB `loadlibrary` function or run an application that uses the MATLAB `loadlibrary` function with MATLAB Compiler:

- Output argument ’notfound’ was not assigned during call to ’loadlibrary’.
-

  ```
  Warning: Function call testloadlibcompile
  invokes inexact match
  d:\work\testLoadLibCompile_mcr\
  testLoadLibCompile\testLoadLibCompile.m.

  ??? Error using ==> loadlibrary
  Call to Perl failed. Possible error processing header file.
  Output of Perl command:
  Error using ==> perl
  ```

```
All input arguments must be valid strings.

Error in ==> testLoadLibCompile at 4
```

-

  ```
  MATLAB:loadlibrary:cannotgeneratemfile
  There was an error running the loader mfile.
  Use the mfilename option
  to produce a file that you can debug and fix.
  Please report this
  error to the MathWorks so we can improve this
  function.
  ??? Error using ==> feval
  Undefined function or variable 'GHlinkTest_proto'.

  Error in ==> loadtest at 6
  ```

For information about how to properly invoke the MATLAB `loadlibrary` function with MATLAB Compiler, see "Load MATLAB Libraries using loadlibrary" on page 7-16 in the Deploying MATLAB Code section in your product user's guide.

MATLAB Compiler Does Not Generate the Application. If you experience other problems with MATLAB Compiler, contact Technical Support at MathWorks at `http://www.mathworks.com/contact_TS.html`.

"MATLAB file may be corrupt" Message Appears. If you receive the message

```
This MATLAB file does not have proper version information and
may be corrupt. Please delete the extraction directory and
rerun the application.
```

when you run your standalone application that was generated by MATLAB Compiler, you should check the following:

- Do you have a `startup.m` file that calls `addpath`? If so, this will cause run-time errors. As a workaround, use `isdeployed` to have the `addpath` command execute only from MATLAB. For example, use a construct such as:

```
if ~isdeployed
 addpath(path);
end
```

- Verify that the `.ctf` archive file self extracted and that you have write permission to the folder.
- Verify that none of the files in the `<application name>_mcr` folder have been modified or removed. Modifying this folder is not supported, and if you have modified it, you should delete it and redeploy or restart the application.
- If none of the above possible causes apply, then the error is likely caused by a corruption. Delete the `<application name>_mcr` folder and run the application.

Missing Functions in Callbacks. If your application includes a call to a function in a callback string or in a string passed as an argument to the `feval` function or an ODE solver, and this is the only place in your MATLAB file this function is called, MATLAB Compiler will not compile the function. MATLAB Compiler does not look in these text strings for the names of functions to compile. See “Fixing Callback Problems: Missing Functions” on page 15-3 for more information.

"MCRInstance not available" Message Appears. If you receive the message `MCRInstance not available` when you try to run a standalone application that was generated with MATLAB Compiler, it can be that the MCR is not located properly on your path or the CTF file is not in the proper folder (if you extracted it from your binary).
The UNIX verification process is the same, except you use the appropriate UNIX path information.

To verify that the MCR is properly located on your path, from a development Windows machine, confirm that *`matlabroot`*`\runtime\win32|win64`, where *`matlabroot`* is your root MATLAB folder, appears on your system path ahead of any other MATLAB installations.

From a Windows target machine, verify that `<mcr_root>\<ver>\runtime\win32|win64`, where `<mcr_root>` is your root MCR folder, appears on your system path. To verify that the CTF file that MATLAB Compiler generated in the build process resides in the same folder as your program's file, look at the folder containing the program's file and make sure the corresponding `.ctf` file is also there.

No Info.plist file in application bundle or no... . On 64-bit Macintosh, indicates the application is not being executed through the bundle.

Deployed Applications

Failed to decrypt file. The MATLAB file "<ctf_root>\toolbox\compiler\deploy\matlabrc.m" cannot be executed. The application is trying to use a CTF archive that does not belong to it. Applications and CTF archives are tied together at compilation time by a unique cryptographic key, which is recorded in both the application and the CTF archive. The keys must match at run time. If they don't match, you will get this error.

To work around this, delete the `*_mcr` folder corresponding to the CTF archive and then rerun the application. If the same failure occurs, you will likely need to recompile the application using MATLAB Compiler and copy both the application binary and the CTF archive into the installation folder.

This application has requested the run time to terminate in an unusual way. This indicates a segmentation fault or other fatal error. There are too many possible causes for this message to list them all.

To try to resolve this problem, run the application in the debugger and try to get a stack trace or locate the line on which the error occurs. Fix the offending code, or, if the error occurs in a MathWorks library or generated code, contact MathWorks technical support.

Checking access to X display <IP-address>:0.0 . . .
If no response hit ^C and fix host or access control to host.
Otherwise, checkout any error messages that follow and fix . . .
Successful. . . . . This message can be ignored.

??? Error: File: /home/username/<MATLAB file_name> Line: 1651 Column: 8 Arguments to IMPORT must either end with ".*" or else specify a fully qualified class name: "<class_name>" fails this test. The `import` statement is referencing a Java class (`<class_name>`) that MATLAB Compiler (if the error occurs at compile time) or the MCR (if the error occurs at run time) cannot find.

To work around this, ensure that the JAR file that contains the Java class is stored in a folder that is on the Java class path. (See *`matlabroot`*`/toolbox/local/classpath.txt` for the class path.) If the error occurs at run time, the classpath is stored in *`matlabroot`*`/toolbox/local/classpath.txt` when running on the development machine. It is stored in `<mcr_root>/toolbox/local/classpath.txt` when running on a target machine.

Warning: Unable to find Java library: matlabroot\sys\java\jre\win32|win64\jre<version>\bin\client\jvm.dll Warning: Disabling Java support. This warning indicates that a compiled application can not find the Java virtual machine, and therefore, the compiled application cannot run any Java code. This will affect your ability to display graphics.

To resolve this, ensure that `jvm.dll` is in the *`matlabroot`*`\sys\java\jre\win32|win64\jre<version>\bin\client` folder and that this folder is on your system path.

Warning: matlabroot\toolbox\local\pathdef.m not found. Toolbox Path Cache is not being used. Type 'help toolbox_path_cache' for more info. The `pathdef.m` file defines the MATLAB startup path. MATLAB Compiler does not include this file in the generated CTF archive because the MCR path is a subset of the full MATLAB path.
This message can be ignored.

Undefined function or variable 'matlabrc'. When MATLAB or the MCR starts, they attempt to execute the MATLAB file `matlabrc.m`. This message means that this file cannot be found.

To work around this, try each of these suggestions in this order:

- Ensure that your application runs in MATLAB (uncompiled) without this error.
- Ensure that MATLAB starts up without this error.
- Verify that the generated CTF archive contains a file called `matlabrc.m`.
- Verify that the generated code (in the `*_mcc_component_data.c*` file) adds the CTF archive folder containing `matlabrc.m` to the MCR path.
- Delete the `*_mcr` folder and rerun the application.
- Recompile the application.

This MATLAB file does not have proper version information and may be corrupt. Please delete the extraction directory and rerun the application. The MATLAB file <MATLAB file> cannot be executed. MATLAB:err_parse_cannot_run_m_file. This message is an indication that the MCR has found nonencrypted MATLAB files on its path and has attempted to execute them. This error is often caused by the use of `addpath`, either explicitly in your application, or implicitly in a `startup.m` file. If you use `addpath` in a compiled application, you must ensure that the added folders contain only data files. (They cannot contain MATLAB files, or you'll get this error.)

To work around this, protect your calls to `addpath` with the `isdeployed` function.

This application has failed to start because mclmcrrt7x.dll was not found. Re-installing the application may fix this problem. `mclmcrrt7x.dll` contains the public interface to the MCR. This library must be present on all machines that run applications generated by MATLAB Compiler. Typically, this means that either the MCR is not installed on this machine, or that the PATH does not contain the folder where this DLL is located.

To work around this, install the MCR or modify the path appropriately. The path must contain `<mcr_root>\<version>\runtime\<arch>`, for example: `c:\mcr\v73\runtime\win32|win64`.

Linker cannot find library and fails to create standalone application (win32 and win64). If you try building your standalone application without `mbuild`, you must link to the following dynamic library:

```
mclmcrrt.lib
```

This library is found in one of the following locations, depending on your architecture:

```
matlabroot\extern\lib\win32\arch
matlabroot\extern\lib\win64\arch
```

where *`arch`* is `microsoft` or `watcom`.

Version 'GCC_4.2.0' not found. When running on Linux platforms, users may report that a run time error occurs that states that the GCC_4.2.0 library is not found by applications built with MATLAB Compiler.

To resolve this error, do the following:

1. Navigate to *`matlabroot`*`/sys/os/glnx86`.
2. Rename the following files with a prefix of `old_`:
   - `libgcc_s.so.1`
   - `libstdc++.so.6.0.8`
   - `libgfortran.so.1.0.0`

   For example, rename `libgcc_s.so.1` to `old_libgcc_s.so.1`. you must rename all three of the above files. Alternately, you can create a subfolder named `old` and move the files there.

Error: library mclmcrrt76.dll not found. This error can occur for the following reasons:

- The machine on which you are trying to run the application an different, incompatible version of the MCR installed on it than the one the application was originally built with.
- You are not running a version of MATLAB Compiler compatible with the MCR version the application was built with.

To solve this problem, on the deployment machine, install the version of MATLAB you used to build the application.

Invalid .NET Framework.\*n* Either the specified framework was not found or is not currently supported. This error occurs when the .NET Framework version your application is specifying (represented by *n*) is not supported by the current version of MATLAB Compiler. See the *MATLAB Builder NE Release Notes* for a list of supported .NET Framework versions.

`MATLAB:I18n:InconsistentLocale`. The warning message

```
MATLAB:I18n:InconsistentLocale - The system locale setting,
system_locale_name, is different from the user locale
setting, user_locale_name
```

indicates a mismatch between locale setting on Microsoft Windows systems. This may affect your ability to display certain characters. For information about changing the locale settings, see your operating system Help.

`System.AccessViolationException:  Attempted to read or write protected memory.` The message:

```
System.ArgumentException: Generate Queries
                               threw General Exception:
System.AccessViolationException: Attempted to
                         read or write protected memory.
 This is often an indication that other memory is corrupt.
```

indicates a library initialization error caused by a Microsoft Visual Studio project linked against a `MCLMCRRT7XX.DLL` placed outside *`matlabroot`*.

15

Limitations and Restrictions

- “MATLAB® Compiler™ Limitations” on page 15-2
- “Licensing Terms and Restrictions on Compiled Applications” on page 15-9
- “MATLAB Functions That Cannot Be Compiled” on page 15-10

MATLAB Compiler Limitations

In this section...

"Compiling MATLAB and Toolboxes" on page 15-2

"Fixing Callback Problems: Missing Functions" on page 15-3

"Finding Missing Functions in a MATLAB File" on page 15-5

"Suppressing Warnings on the UNIX System" on page 15-5

"Cannot Use Graphics with the -nojvm Option" on page 15-6

"Cannot Create the Output File" on page 15-6

"No MATLAB File Help for Compiled Functions" on page 15-6

"No MCR Versioning on Mac OS X" on page 15-7

"Older Neural Networks Not Deployable with MATLAB® Compiler™" on page 15-7

"Restrictions on Calling PRINTDLG with Multiple Arguments in Compiled Mode" on page 15-7

"Compiling a Function with WHICH Does Not Search Current Working Directory" on page 15-8

"Restrictions on Using C++ SETDATA to Dynamically Resize an MWArray" on page 15-8

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 "MATLAB Functions That Cannot Be Compiled" on page 15-10 for a list of functions that cannot be compiled.

Fixing Callback Problems: Missing Functions

When MATLAB Compiler creates a standalone application, it compiles the MATLAB file(s) you specify on the command line and, in addition, it compiles any other MATLAB files that your MATLAB file(s) calls. MATLAB Compiler uses a dependency analysis, which determines all the functions on which the supplied MATLAB files, MEX-files, and P-files depend.

Note If the MATLAB file associated with a p-file is unavailable, the dependency analysis will not be able to discover the p-file's dependencies.

The dependency analysis may not locate a function if the only place the function is called in your MATLAB file is a call to the function either:

- In a callback string
- In a string passed as an argument to the `feval` function or an ODE solver

---

Tip Dependent functions can also be hidden from the dependency analyzer in `.mat` files that get loaded by compiled applications. Use the `mcc -a` argument or the `%#function` pragma to identify `.mat` file classes or functions that should be supported by the `load` command.

MATLAB Compiler does not look in these text strings for the names of functions to compile.

Symptom

Your application runs, but an interactive user interface element, such as a push button, does not work. The compiled application issues this error message:

```
An error occurred in the callback: change_colormap
The error message caught was    : Reference to unknown function
                change_colormap from FEVAL in stand-alone mode.
```

Workaround

There are several ways to eliminate this error:

- Using the `%#function pragma` and specifying callbacks as strings
- Specifying callbacks with function handles
- Using the `-a` option

Specifying Callbacks as Strings. Create a list of all the functions that are specified only in callback strings and pass these functions using separate `%#function` pragma statements. This overrides the product's dependency analysis and instructs it to explicitly include the functions listed in the `%#function` pragmas.

For example, the call to the `change_colormap` function in the sample application, `my_test`, illustrates this problem. To make sure MATLAB Compiler processes the `change_colormap` MATLAB file, list the function name in the `%#function` pragma.

```
function my_test()
% Graphics library callback test application

%#function change_colormap

peaks;

p_btn = uicontrol(gcf,...
'Style', 'pushbutton',...
'Position',[10 10 133 25 ],...
'String', 'Make Black & White',...
'CallBack','change_colormap');
```

Specifying Callbacks with Function Handles. To specify the callbacks with function handles, use the same code as in the example above and replace the last line with

```
'CallBack',@change_colormap);
```

For more information on specifying the value of a callback, see the MATLAB Programming Fundamentals documentation.

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

Finding Missing Functions in a MATLAB File

To find functions in your application that may need to be listed in a `%#function` pragma, search your MATLAB file source code for text strings specified as callback strings or as arguments to the `feval`, `fminbnd`, `fminsearch`, `funm`, and `fzero` functions or any ODE solvers.

To find text strings used as callback strings, search for the characters "Callback" or "fcn" in your MATLAB file. This will find all the `Callback` properties defined by Handle Graphics objects, such as `uicontrol` and `uimenu`. In addition, this will find the properties of figures and axes that end in `Fcn`, such as `CloseRequestFcn`, that also support callbacks.

Suppressing Warnings on the UNIX System

Several warnings may appear when you run a standalone application on the UNIX system. This section describes how to suppress these warnings.

To suppress the `libjvm.so` warning, make sure you set the dynamic library path properly for your platform. See "MCR Path Settings for Run-time Deployment" on page 16-4.

You can also use the MATLAB Compiler option `-R -nojvm` to set your application's `nojvm` run-time option, if the application is capable of running without Java.

Cannot Use Graphics with the -nojvm Option

If your program uses graphics and you compile with the `-nojvm` option, you will get a run-time error.

Cannot Create the Output File

If you receive the error

```
Can't create the output file filename
```

there are several possible causes to consider:

- Lack of write permission for the folder where MATLAB Compiler is attempting to write the file (most likely the current working folder).
- Lack of free disk space in the folder where MATLAB Compiler is attempting to write the file (most likely the current working folder).
- If you are creating a standalone application and have been testing it, it is possible that a process is running and is blocking MATLAB Compiler from overwriting it with a new version.

No MATLAB File Help for Compiled Functions

If you create a MATLAB file with self-documenting online help by entering text on one or more contiguous comment lines beginning with the second line of the file and then compile it, the results of the command

```
help filename
```

will be unintelligible.

Note Due to performance reasons, MATLAB file comments are stripped out before MCR encryption.

No MCR Versioning on Mac OS X

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

Older Neural Networks Not Deployable with MATLAB Compiler

Loading networks saved from older Neural Network Toolbox versions requires some initialization routines that are not deployable. Therefore, these networks cannot be deployed without first being updated.

For example, deploying with Neural Network Toolbox Version 5.0.1 (2006b) and MATLAB Compiler Version 4.5 (R2006b) yields the following errors at run time:

```
??? Error using ==> network.subsasgn
"layers{1}.initFcn" cannot be set to non-existing
 function "initwb".
Error in ==> updatenet at 40
Error in ==> network.loadobj at 10

??? Undefined function or method 'sim' for input
arguments of type 'struct'.
Error in ==> mynetworkapp at 30
```

Restrictions on Calling PRINTDLG with Multiple Arguments in Compiled Mode

In compiled mode, only one argument can be present in a call to the MATLAB `printdlg` function (for example, `printdlg(gcf)`).

You will not receive an error when making at call to `printdlg` with multiple arguments. However, when an application containing the multiple-argument call is compiled, the compile will fail with the following error message:

```
Error using = => printdlg at 11
```

```
PRINTDLG requires exactly one argument
```

Compiling a Function with WHICH Does Not Search Current Working Directory

Using `which`, as in this example:

```
function pathtest
which myFile.mat
open('myFile.mat')
```

does not cause the current working folder to be searched in deployed applications. In addition, it may cause unpredictable behavior of the `open` function.

Use one of the following solutions as alternatives to using `which`:

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

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

- Rather than using the general `open` function, use `load` or other specialized functions for your particular file type, as `load` explicitly checks for the file in the current folder. For example:

  ```
  load myFile.mat
  ```

- Include your file in the **Files required for your application to run** area of the compiler app or the `-a` flag using `mcc`.

Restrictions on Using C++ SETDATA to Dynamically Resize an MWArray

You cannot use the C++ `SETDATA` function to dynamically resize `MWArrays`.

For instance, if you are working with the following array:

```
[1 2 3 4]
```

you cannot use `SETDATA` to increase the size of the array to a length of five elements.

Licensing Terms and Restrictions on Compiled Applications

Applications you build with a trial MATLAB Compiler license are valid for thirty (30) days only.

Applications you build with a purchased license of MATLAB Compiler have no expiration date.

MATLAB Functions That Cannot Be Compiled

Note Due to the number of active and ever-changing list of MathWorks products and functions, this is not a complete list of functions that can not be compiled. If you have a question as to whether a specific MathWorks product's function is able to be compiled or not, the definitive source is that product's documentation, not the MATLAB Compiler documentation.

Some functions are not supported in standalone mode; that is, you cannot compile them with MATLAB Compiler. These functions are in the following categories:

- Functions that print or report MATLAB code from a function, for example, the MATLAB `help` function or debug functions, will not work.
- Simulink® functions, in general, will not work.
- Functions that require a command line, for example, the MATLAB `lookfor` function, will not work.
- `clc`, `home`, and `savepath` will not do anything in deployed mode.
- Tools that allow run-time manipulation of figures

Returned values from standalone applications will be 0 for successful completion or a nonzero value otherwise.

In addition, there are functions and programs that have been identified as nondeployable due to licensing restrictions.

`mccExcludedFiles.log` lists all the functions and files excluded by `mcc` if they can not be compiled. It is created after each attempted build if there are functions or files that cannot be compiled.

List of Unsupported Functions and Programs

`add_block`

`add_line`

List of Unsupported Functions and Programs (Continued)

`applescript`

`checkcode`

`close_system`

`colormapeditor`

`commandwindow`

Control System Toolbox™ prescale GUI

`createClassFromWsdl`

`dbclear`

`dbcont`

`dbdown`

`dbquit`

`dbstack`

`dbstatus`

`dbstep`

`dbstop`

`dbtype`

`dbup`

`delete_block`

`delete_line`

`doc`

`echo`

`edit`

`fields`

`figure_palette`

`get_param`

`help`

`home`

List of Unsupported Functions and Programs (Continued)

`inmem`

`keyboard`
`linkdata`
`linmod`
`mislocked`
`mlock`
`more`
`munlock`
`new_system`
`open_system`
`pack`
`pcode`
`plotbrowser`
`plotedit`
`plottools`
`profile`
`profsave`
`propedit`
`propertyeditor`
`publish`
`rehash`
`restoredefaultpath`
`run`
`segment`
`set_param`
`sim`

List of Unsupported Functions and Programs (Continued)

`simget`

`simset`

`sldebug`

`type`

16

Reference Information

- “MCR Path Settings for Development and Testing” on page 16-2
- “MCR Path Settings for Run-time Deployment” on page 16-4
- “MATLAB® Compiler™ Licensing” on page 16-6
- “Deployment Product Terms” on page 16-8

MCR Path Settings for Development and Testing

In this section…

"Overview" on page 16-2
"Path for Java Development on All Platforms " on page 16-2
"Path Modifications Required for Accessibility" on page 16-2
"Windows Settings for Development and Testing" on page 16-3
"Linux Settings for Development and Testing" on page 16-3
"Mac Settings for Development and Testing" on page 16-3

Overview

The following information is for developers developing applications that use libraries or components that contain compiled MATLAB code. These settings are required on the machine where you are developing your application.

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.

Path Modifications Required for Accessibility

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

```
JavaAccessBridge.dll
WindowsAccessBridge.dll
```

You may not be able to use such technologies without doing so.

Windows Settings for Development and Testing

When programming with components that are generated with MATLAB Compiler, add the following folder to your system `PATH` environment variable:

```
matlabroot\runtime\win32|win64
```

Linux Settings for Development and Testing

Add the following platform-specific folders to your dynamic library path.

Note For readability, the following commands appear on separate lines, but you must enter each `setenv` command on one line.

Linux (64-bit)

```
setenv LD_LIBRARY_PATH
   matlabroot/runtime/glnxa64:
   matlabroot/bin/glnxa64:
   matlabroot/sys/os/glnxa64:
```

Mac Settings for Development and Testing

Add the following platform-specific folders to your dynamic library path.

Note For readability, the following commands appear on separate lines, but you must enter each `setenv` command on one line.

Mac

```
setenv DYLD_LIBRARY_PATH
   matlabroot/runtime/maci64:
   matlabroot/bin/maci64:
   matlabroot/sys/os/maci64:
```

MCR Path Settings for Run-time Deployment

In this section…

"General Path Guidelines" on page 16-4

"Path for Java Applications on All Platforms" on page 16-4

"Windows Path for Run-Time Deployment" on page 16-4

"Linux Paths for Run-Time Deployment" on page 16-5

"Mac Paths for Run-Time Deployment" on page 16-5

General Path Guidelines

Regardless of platform, be aware of the following guidelines with regards to placing specific folders on the path:

- Always avoid including `bin` or `arch` on the path. Failure to do so may inhibit ability to run multiple MCR instances.
- Ideally, set the environment in a separate shell script to avoid run-time errors caused by path-related issues.

Path for Java Applications on All Platforms

When your users run applications that contain compiled MATLAB code, you must instruct them to set the path so that the system can find the MCR.

Note When you deploy a Java application to end users, they must set the class path on the target machine.

The system needs to find `.jar` files containing the MATLAB libraries. To tell the system how to locate the `.jar` files it needs, specify a `classpath` either in the `javac` command or in your system environment variables.

Windows Path for Run-Time Deployment

The following folder should be added to the system path:

```
mcr_root\version\runtime\win32|win64
```

where *`mcr_root`* refers to the complete path where the MCR library archive files are installed on the machine where the application is to be run.

`mcr_root` is version specific; you must determine the path after you install the MCR.

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

Linux Paths for Run-Time Deployment

Use these `setenv` commands to set your MCR run-time paths.

```
setenv LD_LIBRARY_PATH
   mcr_root/version/runtime/glnxa64:
   mcr_root/version/bin/glnxa64:
   mcr_root/version/sys/os/glnxa64:
```

Mac Paths for Run-Time Deployment

Use these `setenv` commands to set your MCR run-time paths.

```
setenv DYLD_LIBRARY_PATH
   mcr_root/version/runtime/maci64:
   mcr_root/version/bin/maci64:
   mcr_root/version/sys/os/maci64:
   mcr_root/sys/java/jar/maci64/jre/lib/server
```

MATLAB Compiler Licensing

Using MATLAB Compiler Licenses for Development

You can run MATLAB Compiler from the MATLAB command prompt (MATLAB mode) or the DOS/UNIX prompt (standalone mode).

MATLAB Compiler uses a *lingering license*. This means that when the MATLAB Compiler license is checked out, a timer is started. When that timer reaches 30 minutes, the license key is returned to the license pool. The license key will not be returned until that 30 minutes is up, regardless of whether `mcc` has exited or not.

Each time a compiler command is issued, the timer is reset.

Running MATLAB Compiler in MATLAB Mode

When you run MATLAB Compiler from “inside” of the MATLAB environment, that is, you run `mcc` from the MATLAB command prompt, you hold the MATLAB Compiler license as long as MATLAB remains open. To give up the MATLAB Compiler license, exit MATLAB.

Running MATLAB Compiler in Standalone Mode

If you run MATLAB Compiler from a DOS or UNIX prompt, you are running from “outside” of MATLAB. In this case, MATLAB Compiler

- Does not require MATLAB to be running on the system where MATLAB Compiler is running
- Gives the user a dedicated 30-minute time allotment during which the user has complete ownership over a license to MATLAB Compiler

Each time a user requests MATLAB Compiler , the user begins a 30-minute time period as the sole owner of the MATLAB Compiler license. Anytime during the 30-minute segment, if the same user requests MATLAB Compiler , the user gets a new 30-minute allotment. When the 30-minute interval has elapsed, if a different user requests MATLAB Compiler , the new user gets the next 30-minute interval.

When a user requests MATLAB Compiler and a license is not available, the user receives the message

```
Error: Could not check out a Compiler License.
```

This message is given when no licenses are available. As long as licenses are available, the user gets the license and no message is displayed. The best way to guarantee that all MATLAB Compiler users have constant access to MATLAB Compiler is to have an adequate supply of licenses for your users.

Deployment Product Terms

A

Add-in — A Microsoft Excel add-in is an executable piece of code that can be actively integrated into a Microsoft Excel application. Add-ins are front-ends for COM components, usually written in some form of Microsoft Visual Basic®.

Application program interface (API) — A set of classes, methods, and interfaces that is used to develop software applications. Typically an API is used to provide access to specific functionality. See `MWArray`.

Application — An end user-system into which a deployed functions or solution is ultimately integrated. Typically, the end goal for the Deployment customer is integration of a deployed MATLAB function into a larger enterprise environment application. The deployment products prepare the MATLAB function for integration by wrapping MATLAB code with enterprise-compatible source code, such as C, C++, C# (.NET), F#, and Java code.

Assembly — An executable bundle of code, especially in .NET. For example, after building a deployable .NET component with MATLAB Builder NE, the .NET developer integrates the resulting .NET assembly into a larger enterprise C# application. See *Executable*.

B

Binary — See *Executable*.

Boxed Types — Data types used to wrap opaque C structures.

Build — See *Compile*.

C

Class — A user-defined type used in C++, C#, and Java, among other object-oriented languages, that is a prototype for an object in an object-oriented language. It is analogous to a derived type in a procedural language. A class is a set of objects which share a common structure and behavior. Classes relate in a class hierarchy. One class is a specialization (a *subclass*) of another (one of its *superclasses*) or comprises other classes. Some classes use other classes in a client-server relationship. Abstract classes have no members, and concrete classes have one or more members. Differs from a *MATLAB class*

Compile — In MATLAB Compiler terminology, to compile a component involves generating a binary that wraps around MATLAB code, enabling it to execute in various computing environments. For example, when MATLAB code builds with MATLAB Builder JA, a Java wrapper provides Java code that enables the MATLAB code to execute in a Java environment.

COM component — In MATLAB Builder EX, the executable back-end code behind a Microsoft Excel add-in. In MATLAB Builder NE, an executable component, to be integrated with Microsoft COM applications.

Component — In MATLAB, a generic term used to describe the wrappered MATLAB code produced by MATLAB Compiler. You can plug these self-contained bundles of code you plug into various computing environments. The wrapper enables the compatibility between the computing environment and your code.

Console application — Any application that is executed from a system command prompt window.

CTF archive (Component Technology File) — The Component Technology File (CTF) archive is embedded by default in each generated binary by MATLAB Compiler. It houses the deployable package. All MATLAB-based content in the CTF archive uses the Advanced Encryption Standard (AES) cryptosystem. See “Additional Details” on page 7-9 in the MATLAB Compiler documentation.

D

Data Marshaling — Data conversion, usually from one type to another. Unless a MATLAB deployment customer is using type-safe interfaces, data marshaling—as from mathematical data types to MathWorks data types such as represented by the `MWArray` API—must be performed manually, often at great cost.

Deploy — The act of integrating a component into a larger-scale computing environment, usually to an enterprise application, and often to end users.

DLL — Dynamic link library. Microsoft's implementation of the shared library concept for Windows. Using DLLs is much preferred over the previous technology of static (or non-dynamic) libraries, which had to be manually linked and updated.

E

Empties — Arrays of zero (`0`) dimensions.

Executable — An executable bundle of code, made up of binary bits (zeros and ones) and sometimes called a *binary*.

F

Fields — For this definition in the context of MATLAB Data Structures, see *Structs*.

Fields and Properties — In the context of .NET, *Fields* are specialized classes used to hold data. *Properties* allow users to access class variables as if they were accessing member fields directly, while actually implementing that access through a class method.

H

Helper files — Files that support the main file or the file that calls all supporting functions. Add resources that depend upon the function that calls the supporting function to the **Shared Resources and Helper Files** section of the Deployment Tool GUI. Other examples of supporting files or resources include:

- Functions called using `eval` (or variants of `eval`)
- Functions not on the MATLAB path
- Code you want to remain private
- Code from other programs that you want to compile and link into the main file

I

Integration — Combining a deployed component's functionality with functionality that currently exists in an enterprise application. For example, a customer creates a mathematical model to forecast trends in certain commodities markets. In order to use this model in a larger-scale financial application (one written with the Microsoft .NET Framework, for instance) the deployed financial model must be integrated with existing C# applications, run in the .NET enterprise environment. Integration is usually performed by an IT developer, rather than a MATLAB Programmer, in larger environments.

Instance — For the definition of this term in context of MATLAB Production Server software, see *MATLAB Production Server Server Instance*.

J

JAR — Java archive. In computing software, a JAR file (or Java ARchive) aggregates many files into one. Software developers use JARs to distribute Java applications or libraries, in the form of classes and associated metadata and resources (text, images, etc.). Computer users can create or extract JAR files using the `jar` command that comes with a Java Development Kit (JDK).

Java-MATLAB Interface — Known as the *JMI Interface*, this is the Java interface built into MATLAB software.

JDK — The *Java Development Kit* is a free Oracle® product which provides the environment required for programming in Java. The JDK™ is available for various platforms, but most notably Oracle Solaris™ and Microsoft Windows. To build components with MATLAB Builder JA, download the JDK that corresponds to the latest version of Java supported by MATLAB.

JMI Interface — see *Java-MATLAB Interface*.

JRE — Java Run-Time Environment is the part of the Java Development Kit (JDK) required to run Java programs. It comprises the Java Virtual Machine, the Java platform core classes, and supporting files. It does not include the compiler, debugger, or other tools present in the JDK. The JRE™ is the smallest set of executables and files that constitute the standard Java platform.

M

Magic Square — A square array of integers arranged so that their sum is the same when added vertically, horizontally, or diagonally.

MATLAB Production Server Client — In the MATLAB Production Server software, clients are applications written in a language supported by MATLAB Production Server that call deployed functions hosted on a server.

MATLAB Production Server Configuration — An instance of the MATLAB Production Server containing at least one server and one client. Each configuration of the software usually contains a unique set of values in the server configuration file, `main_config`.

MATLAB Production Server Server Instance — A logical server configuration created using the `mps-new` command in MATLAB Production Server software.

MATLAB Production Server Software — Product for server/client deployment of MATLAB programs within your production systems, enabling you to incorporate numerical analytics in enterprise applications. When you use this software, Web, database, and enterprise applications connect to MATLAB programs running on MATLAB Production Server via a lightweight client library, isolating the MATLAB programs from your production system. MATLAB Production Server software consists of one or more servers and clients.

Marshaling — See *Data Marshaling*.

mbuild — MATLAB Compiler command that compiles and links C and C++ source files into standalone applications or shared libraries. For more information, see the `mbuild` function reference page.

`mcc` — The MATLAB command that invokes MATLAB Compiler. It is the command-line equivalent of using the compiler apps.

MCR — The *MATLAB Compiler Runtime* is an execution engine made up of the same shared libraries. MATLAB uses these libraries to enable the execution of MATLAB files on systems without an installed version of MATLAB. To deploy a component, you package the MCR along with it. Before you use the MCR on a system without MATLAB, run the *MCR Installer*.

MCR Installer — An installation program run to install the MATLAB Compiler Runtime on a development machine that does not have an installed version of MATLAB. Find out more about the MCR Installer by reading "Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)" on page 10-2.

MCR Singleton — See *Shared MCR Instance*.

MCR Workers — A MATLAB Compiler Runtime session. Using MATLAB Production Server software, you have the option of specifying more than one MCR session, using the `--num-workers` options in the server configurations file.

Method Attribute — In the context of .NET, a mechanism used to specify declarative information to a .NET class. For example, in the context of client programming with MATLAB Production Server software, you specify method attributes to define MATLAB structures for input and output processing.

`mxArray` *interface* — The MATLAB data type containing all MATLAB representations of standard mathematical data types.

`MWArray` *interface* — A proxy to `mxArray`. An application program interface (API) for exchanging data between your application and MATLAB. Using `MWArray`, you marshal data from traditional mathematical types to a form that can be processed and understood by MATLAB data type `mxArray`. There are different implementations of the `MWArray` proxy for each application programming language.

P

Package — The act of bundling the deployed component, along with the MCR and other files, into an installer that can be distributed to others. The compiler apps place the installer in the `for_redistribution` subfolder. In addition to the installer, the compiler apps generate a number of lose artifacts that can be used for testing or building a custom installer.

PID File — See *Process Identification File (PID File)*.

Pool — A pool of threads, in the context of server management using MATLAB Production Server software. Servers created with the software do not allocate a unique thread to each client connection. Rather, when data is available on a connection, the required processing is scheduled on a *pool*, or group, of available threads. The server configuration file option `--num-threads` sets the size of that pool (the number of available request-processing threads) in the master server process.

Process Identification File (PID File) — A file that documents informational and error messages relating to a running server instance of MATLAB Production Server software.

Program — A bundle of code that is executed to achieve a purpose. Programs usually are written to automate repetitive operations through computer processing. Enterprise system applications usually consist of hundreds or even thousands of smaller programs.

Properties — For this definition in the context of .NET, see *Fields and Properties*.

Proxy — A software design pattern typically using a class, which functions as an interface to something else. For example, `MWArray` is a proxy for programmers who need to access the underlying type `mxArray`.

S

Server Instance — See MATLAB Production Server Server Instance.

Shared Library — Groups of files that reside in one space on disk or memory for fast loading into Windows applications. Dynamic-link libraries (DLLs) are Microsoft's implementation of the shared library concept in for Microsoft Windows.

Shared MCR Instance — When using MATLAB Builder NE or MATLAB Builder JA, you can create a shared MCR instance, also known as a *singleton*. For builder NE, this only applies to COM components. When you invoke MATLAB Compiler with the `-S` option through the builders (using either `mcc` or the Deployment Tool), a single MCR instance is created for each COM or Java component in an application. You reuse this instance by sharing it among all subsequent class instances within the component. Such sharing results in more efficient memory usage and eliminates the MCR startup cost in each subsequent class instantiation. All class instances share a single

MATLAB workspace and share global variables in the MATLAB files used to build the component. MATLAB Builder NE and MATLAB Builder EX are designed to create singletons by default for .NET assemblies and COM components, respectively. For more information, see "Sharing an MCR Instance in COM or Java Applications".

Standalone application — Programs that can be executed on their own and encapsulate a self contained set of MATLAB functionality.

State — The present condition of a the MATLAB, or MCR, runtime. MATLAB functions often carry state in the form of variable values. The MATLAB Workspace itself also maintains information about global variables and path settings. When deploying functions that carry state, you must often take additional steps to ensure state retention when deploying applications that use such functions.

Structs — MATLAB Structures. Structs are MATLAB arrays with elements that you access using textual field designators. Fields are data containers that store data of a specific MATLAB type.

System Compiler — A key part of Interactive Development Environments (IDEs) such as Microsoft Visual Studio.

T

Thread — A portion of a program that can run independently of and concurrently with other portions of the program. See *pool* for additional information on managing the number of processing threads available to a server instance.

Type-safe interface — An API that minimizes explicit type conversions by hiding the `MWArray` type from the calling application. Using "Generate and Implement Type-Safe Interfaces", for example, .NET Developers work directly with familiar native data types. You can avoid performing tedious `MWArray` data marshaling by using type-safe interfaces.

W

Web Application Archive (WAR) —In computing, a Web Application Archive is a JAR file used to distribute a collection of `JavaServer` pages, servlets, Java classes, XML files, tag libraries, and static Web pages (HTML and related files) that together constitute a Web application.

Webfigure — A MathWorks representation of a MATLAB figure, rendered on the Web. Using the WebFigures feature, you display MATLAB figures on a Web site for graphical manipulation by end users. This enables them to use their graphical applications from anywhere on the Web, without the need to download MATLAB or other tools that can consume costly resources.

Windows Communication Foundation (WCF) — The Windows Communication Foundation™ is an application programming interface in the .NET Framework for building connected, service-oriented, Web-centric applications. WCF is designed in accordance with service oriented architecture principles to support distributed computing where services are consumed by client applications.

17

Functions — Alphabetical List

%#function
applicationCompiler
productionServerCompiler
ctfroot
deployprint
deploytool
figToImStream
getmcruserdata
<library>Initialize[WithHandlers]
isdeployed
ismcc
libraryCompiler
mbuild
mcc
mclGetLastErrorMessage
mclGetLogFileName
mclInitializeApplication
mclIsJVMEnabled
mclIsMCRInitialized
mclIsNoDisplaySet
mclmcrInitialize
mclRunMain
mclTerminateApplication
mclWaitForFiguresToDie
mcrinstaller
mcrversion

setmcruserdata
<library>Terminate

Purpose

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

Syntax

`%#function` *`function1`* `[`*`function2 ... functionN`*`]`

`%#function` *`object_constructor`*

Description

The `%#function` pragma informs MATLAB Compiler that the specified function(s) will be called through an `feval`, `eval`, or Handle Graphics callback.

Use the `%#function` pragma in standalone applications to inform MATLAB Compiler that the specified function(s) should be included in the compilation, whether or not MATLAB Compiler's dependency analysis detects the function(s). It is also possible to include objects by specifying the object constructor.

Without this pragma, the product's dependency analysis will not be able to locate and compile all MATLAB files used in your application. This pragma adds the top-level function as well as all the local functions in the file to the compilation.

Examples

Example 1

```
 function foo
   %#function bar

      feval('bar');

   end %function foo
```

By implementing this example, MATLAB Compiler is notified that function `bar` will be included in the compilation and is called through `feval`.

Example 2

```
function foo
   %#function bar foobar

      feval('bar');
      feval('foobar');

   end %function foo
```

In this example, multiple functions (`bar` and `foobar`) are included in the compilation and are called through `feval`.

Purpose

Build and package functions into standalone applications

Syntax

`applicationCompiler` [-win32] [[[-build] | [-project]]*`project_name`*]

Description

`applicationCompiler` opens the MATLAB standalone compiler for the creation of a new compiler project

`applicationCompiler project_name` opens the MATLAB standalone compiler app with the project preloaded.

`applicationCompiler -build project_name` runs the MATLAB standalone compiler to build the specified project. The installer is not generated.

`applicationCompiler -package project_name` runs the MATLAB standalone compiler to build and package the specified project. The installer is generated.

`applicationCompiler -win32` instructs the compiler to build a 32-bit application on a 64-bit system when you use the same MATLAB installation root (*`matlabroot`*) for both 32-bit and 64-bit versions of MATLAB.

Input Arguments

project_name - name of the project to be compiled

Specify the name of a previously saved MATLAB Compiler project. The project must be on the current path.

See Also

`deploytool` **|** `mcc`

productionServerCompiler

Purpose

Build and package functions for use with MATLAB Production Server

Syntax

`productionServerCompiler` [-win32] [[[-build] `|` [-project]]*`project_name`*]

Description

`productionServerCompiler` opens the MATLAB compiler for the creation of a new compiler project

`productionServerCompiler project_name` opens the MATLAB compiler with the project preloaded.

`productionServerCompiler -build project_name` runs the MATLAB compiler to build the specified project. The installer is not generated.

`productionServerCompiler -package project_name` runs the MATLAB compiler to build and package the specified project. The installer is generated.

`productionServerCompiler -win32` instructs the compiler to build a 32-bit application on a 64-bit system when you use the same MATLAB installation root (*`matlabroot`*) for both 32-bit and 64-bit versions of MATLAB.

Input Arguments

project_name - name of the project to be compiled

Specify the name of a previously saved MATLAB Compiler project. The project must be on the current path.

Purpose

Location of files related to deployed application (CTF archive)

Syntax

`ctfroot`

Description

`root =` *`ctfroot`* returns a string that is the name of the folder where the CTF file for the deployed application is expanded.

This function differs from `matlabroot`, which returns the path to where core MATLAB functions and libraries are located. `matlabroot` returns the root directory of the MCR when run against an installed MCR.

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

Examples

`appRoot = ctfroot;` will return the location of your deployed application files in this form: *`application_name`*`_mcr`.

Use this function to access any file that the user would have included in their project (excluding the ones in the packaging folder).

How To

- "Component Technology File (CTF Archive)" on page 7-6

deployprint

Purpose

Use to print to a printer when working with deployed Windows applications

Syntax

`deployprint`

Description

In cases where the `print` command would normally be issued when running MATLAB software, use `deployprint` when working with deployed applications.

`deployprint` is available on all platforms, however it is only required on Windows.

`deployprint` supports all of the input arguments supported by `print` except for the following.

Argument	**Description**
`-d`	Used to specify the type of the output (for example. .JPG, .BMP, etc.). `deployprint` only produces .BMP files. **Note** To print to a file, use the `print` function.
`-noui`	Used to suppress printing of user interface controls. Similar to use in MATLAB `print` function.
`-setup`	The `-setup` option is not supported.
`-s` *`windowtitle`*	MATLAB Compiler does not support Simulink.

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

- `PaperPosition`
- `PaperSize`
- `PaperUnits`

- `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`

deploytool

Purpose

Compile and package functions for external deployment

Syntax

`deploytool` [-win32] [[[-build] | [-project]]*`project_name`*]

Description

`deploytool` opens the MATLAB Compiler app.

`deploytool project_name` opens the MATLAB Compiler app with the project preloaded.

`deploytool -build project_name` runs the MATLAB Compiler to build the specified project. The installer is not generated.

`deploytool -package project_name` runs the MATLAB Compiler to build and package the specified project. The installer is generated.

`deploytool -win32` instructs the compiler to build a 32-bit application on a 64-bit system when the following are true:

- You use the same MATLAB installation root (*`matlabroot`*) for both 32-bit and 64-bit versions of MATLAB.
- You are running from a Windows command line (not a MATLAB command line).

Input Arguments

project_name - name of the project to be compiled

Specify the name of a previously saved MATLAB Compiler project. The project must be on the current path.

Purpose

Stream out figure as byte array encoded in format specified, creating signed byte array in `.png` format

Syntax

```
output type = figToImStream ('fighandle', figure_handle,
   'imageFormat', image_format, 'outputType', output_type)
```

Description

The *`output type`* `= figToImStream ('fighandle',` *`figure_handle`*`, 'imageFormat',` *`image_format`*`, 'outputType',` *`output_type`*`)` command also accepts user-defined variables for any of the input arguments, passed as a comma-separated list

The size and position of the printed output depends on the figure's `PaperPosition[mode]` properties.

Options

`figToImStream('figHandle',` *`Figure_Handle`*`, ...)` allows you to specify the figure output to be used. The Default is the current image

`figToImStream('imageFormat', [png|jpg|bmp|gif])` allows you to specify the converted image format. Default value is `png`.

`figToImStream('outputType', [int8!uint8])` allows you to specify an output byte data type. `uint8` (unsigned byte) is used primarily for .NET primitive byte. Default value is `uint8`.

Examples

Convert the current figure to a signed `png` byte array:

```
surf(peaks)
bytes = figToImStream
```

Convert a specific figure to an unsigned `bmp` byte array:

```
f = figure;
surf(peaks);
bytes = figToImStream(  'figHandle', f, ...
                        'imageFormat', 'bmp', ...
                        'outputType', 'uint8' );
```

getmcruserdata

Purpose Retrieve MATLAB array value associated with given string key

Syntax `function_value = getmcruserdata(key)`

Description The `function_value = getmcruserdata(key)` command is part of the MCR User Data interface API. It returns an empty matrix if no such key exists. For information about this function, as well as complete examples of usage, see "Using the MCR User Data Interface" on page 9-32.

Examples

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

See Also `setmcruserdata`

<library>Initialize[WithHandlers]

Purpose

Initialize 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.

Examples

```
if (!libmatrixInitialize())
{
    fprintf(stderr,
             An error occurred while initializing: \n %s  ,
            mclGetLastErrorMessage());
    return -2;
}
```

See Also

`<library>Terminate`

How To

- “Library Initialization and Termination Functions” on page 13-26

isdeployed

Purpose Determine whether code is running in deployed or MATLAB mode

Syntax `x = isdeployed`

Description `x = isdeployed` returns true (1) when the function is running in deployed mode and false (0) if it is running in a MATLAB session.

If you include this function in an application and compile the application with MATLAB Compiler, the function will return true when the application is run in deployed mode. If you run the application containing this function in a MATLAB session, the function will return false.

ismcc

Purpose

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

Syntax

```
x = ismcc
```

Description

`x = ismcc` returns true when the function is being executed by `mcc` dependency checker and false otherwise.

When this function is executed by the compilation process started by `mcc`, it will return true. This function will return false when executed within MATLAB as well as in deployed mode. To test for deployed mode execution, use `isdeployed`. This function should be used to guard code in `matlabrc`, or `hgrc` (or any function called within them, for example `startup.m` in the example on this page), from being executed by MATLAB Compiler (`mcc`) or any of the Builder products.

In a typical example, a user has `ADDPATH` calls in their MATLAB code. These can be guarded from executing using `ismcc` during the compilation process and `isdeployed` for the deployed application or component as shown in the example on this page.

Examples

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

See Also

`isdeployed` | `mcc`

libraryCompiler

Purpose Build and package functions for use in external applications

Syntax `libraryCompiler` [-win32] [[[-build] | [-project]]*`project_name`*]

Description `libraryCompiler` opens the MATLAB shared library compiler for the creation of a new compiler project

`libraryCompiler project_name` opens the MATLAB shared library compiler app with the project preloaded.

`libraryCompiler -build project_name` runs the MATLAB shared library compiler to build the specified project. The installer is not generated.

`libraryCompiler -package project_name` runs the MATLAB shared library compiler to build and package the specified project. The installer is generated.

`libraryCompiler -win32` instructs the compiler to build a 32-bit application on a 64-bit system when you use the same MATLAB installation root (*`matlabroot`*) for both 32-bit and 64-bit versions of MATLAB.

Input Arguments

project_name - name of the project to be compiled

Specify the name of a previously saved MATLAB Compiler project. The project must be on the current path.

Purpose

Compile and link source files against MATLAB generated shared libraries

Syntax

```
mbuild [option1 ... optionN] sourcefile1 [... sourcefileN]
   [objectfile1 ... objectfileN] [libraryfile1 ...
libraryfileN]
```

Description

`mbuild` compiles and links customer written C or C++ code against MATLAB generated shared libraries.

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.

Supported Source File Types

Supported types of source files are:

- `.c`
- `.cpp`

Source files that are not one of the supported types are passed to the linker.

Options

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.

Option	Description
`-c`	Compile only. Creates an object file only.
`-D<name>`	Define a symbol name to the C preprocessor. Equivalent to a `#define <name>` directive in the source.
`-D<name>=<value>`	Define a symbol name and value to the C preprocessor. Equivalent to a `#define <name> <value>` directive in the source.
`-f <optionsfile>`	Specify location and name of options file to use. Overrides the `mbuild` default options file search mechanism.
`-g`	Create an executable containing additional symbolic information for use in debugging. This option disables the `mbuild` default behavior of optimizing built object code (see the `-O` option).
`-h[elp]`	Print help for `mbuild`.
`-I<pathname>`	Add `<pathname>` to the list of folders to search for `#include` files.
`-l<name>`	Link with object library. On Windows systems, `<name>` expands to `<name>.lib` or `lib<name>.lib` and on UNIX systems, to `lib<name>.so` or `lib<name>.dylib`. Do not add a space after this switch. **Note** When linking with a library, it is essential that you first specify the path (with `-I<pathname>`, for example).

Option	Description
`-L<folder>`	Add `<folder>` to the list of folders to search for libraries specified with the `-l` option. On UNIX systems, you must also set the run-time library path, as explained in “Building Your Application on Mac or Linux ” on page B-10. Do not add a space after this switch.
`-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.
`-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>`	Supplement or override an options file variable for variable `<name>`. This option is processed after the options file is processed and all command line arguments are considered. You may need to use the shell's quoting syntax to protect characters such as spaces that have a meaning in the shell syntax. On Windows double quotes are used (e.g., `COMPFLAGS="opt1 opt2"`), and on UNIX single quotes are used (e.g., `CFLAGS='opt1 opt2'`). It is common to use this option to supplement a variable already defined. To do this, refer to the variable by prepending a `$` (e.g., `COMPFLAGS="$COMPFLAGS opt2"` on Windows or `CFLAGS='$CFLAGS opt2'` on UNIX shell).

Examples

To 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

Compile MATLAB functions for deployment

Syntax

```
mcc {-e} | {-m} [-a filename]… [-B filename[:arg]…] [-C] [-d outFolder]
[-f filename] [-g] [-I directory]… [-K] [-M string] [-N] [-o filename]
[-p path]… [-R option] [-v] [-w option[:msg]] [-win32] [-Y filename]
mfilename
```

```
mcc -l [-a filename]… [-B filename[:arg]…] [-C] [-d outFolder] [-f
filename] [-g] [-I directory]… [-K] [-M string] [-N] [-o filename]
[-p path]… [-R option] [-v] [-w option[:msg]] [-win32] [-Y filename]
mfilename…
```

```
mcc -c [-a filename]… [-B filename[:arg]…] [-C] [-d outFolder] [-f
filename] [-g] [-I directory]… [-K] [-M string] [-N] [-o filename]
[-p path]… [-R option] [-v] [-w option[:msg]] [-win32] [-Y filename]
mfilename…
```

```
mcc -W cpplib:component_name -T link:lib [-a filename]… [-B
filename[:arg]…] [-C] [-d outFolder] [-f filename] [-g] [-I directory]…
[-K] [-M string] [-N] [-o filename] [-p path]… [-R option] [-S] [-v] [-w
option[:msg]] [-win32] [-Y filename] mfilename…
```

```
mcc -W dotnet:component_name,[className], [framework_version],
security, remote_type -T link:lib [-a filename]… [-B filename[:arg]…]
[-C] [-d outFolder] [-f filename] [-I directory]… [-K] [-M string] [-N]
[-p path]… [-R option] [-S] [-v] [-w option[:msg]] [-win32] [-Y filename]
mfilename… [class{className:[mfilename]…}]…
```

```
mcc -W excel:component_name,[className], [version] -T link:lib [-a
filename]… [-b] [-B filename[:arg]…] [-C] [-d outFolder] [-f filename]
[-I directory]… [-K] [-M string] [-N] [-p path]… [-R option] [-u] [-v]
[-w option[:msg]] [-win32] [-Y filename] mfilename…
```

```
mcc -W 'java:packageName,[className]' [-a filename]… [-b]
[-B filename[:arg]…] [-C] [-d outFolder] [-f filename] [-I
directory]… [-K] [-M string] [-N] [-p path]… [-R option]
[-S] [-v] [-w option[:msg]] [-win32] [-Y filenamem] filename…
[class{className:[mfilename]…}]…
```

`mcc` -W CTF:*component_name* [-a *filename*]... [-b] [-B *filename*[:*arg*]...] [-d *outFolder*] [-f *filename*] [-I *directory*]... [-K] [-M *string*] [-N] [-p *path*]... [-R *option*] [-S] [-v] [-w *option*[:*msg*]] [-win32] [-Y *filename*m] *filename*... [class{*className*:[m*filename*]...}]...

`mcc` -?

Description

`mcc -m mfilename` compiles the function into a standalone application.

This is equivalent to `-W main -T link:exe`.

`mcc -e mfilename` compiles the function into a standalone application that does not open an MS-DOS® command window.

This is equivalent to `-W WinMain -T link:exe`.

`mcc -l mfilename...` compiles the listed functions into a C shared library and generates C wrapper code for integration with other applications.

This is equivalent to `-W lib:`*`libname`*` -T link:lib`.

`mcc -c mfilename...` generates C wrapper code for the listed functions.

This is equivalent to `-W lib:`*`libname`*` -T codegen`.

`mcc -W cpplib:component_name -T link:lib mfilename...` compiles the listed functions into a C++ shared library and generates C++ wrapper code for integration with other applications.

`mcc -W dotnet:component_name,className,framework_version,security, remote_type -T link:lib mfilename...` creates a .NET component from the specified files.

- *`component_name`* — Specifies the name of the component and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *`className`* — Specifies the name of the .NET class to be created.
- *`framework_version`* — Specifies the version of the Microsoft .NET Framework you want to use to compile the component. Specify either:
  - `0.0` — Use the latest supported version on the target machine.
  - *`version_major.version_minor`* — Use a specific version of the framework.

  Features are often version-specific. Consult the documentation for the feature you are implementing to get the Microsoft .NET Framework version requirements.
- *`security`* — Specifies whether the component to be created is a private assembly or a shared assembly.
  - To create a private assembly, specify `Private`.
  - To create a shared assembly, specify the full path to the encryption key file used to sign the assembly.
- *`remote_type`* — Specifies the remoting type of the component. Values are `remote` and `local`.

By default, the compiler generates a single class with a method for each function specified on the command line. You can instruct the compiler to create multiple classes using `class{`*`className`*`:`*`mfilename`*`...}....` *`className`* specifies the name of the class to create using *`mfilename`*.

`mcc -W excel:component_name,className, version -T link:lib mfilename...` creates a Microsoft Excel component from the specified files.

- *`component_name`* — Specifies the name of the component and its namespace, which is a period-separated list, such as `companyname.groupname.component`.

- *`className`* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the *`component_name`* as the default.
- *`version`* — Specifies the version of the component specified as *`major.minor`*.
  - *`major`* — Specifies the major version number. If you do not specify a version number, `mcc` uses the latest version.
  - *`minor`* — Specifies the minor version number. If you do not specify a version number, `mcc` uses the latest version.

`mcc -W 'java:packageName,className' mfilename...` creates a Java package from the specified files.

- *`packageName`* — Specifies the name of the Java package and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *`className`* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the last item in *`packageName`*.

By default, the compiler generates a single class with a method for each function specified on the command line. You can instruct the compiler to create multiple classes using `class{`*`className:mfilename`*`...}....` *`className`* specifies the name of the class to create using *`mfilename`*.

`mcc -W CTF:component_name` instructs the compiler to create a deployable CTF archive that is deployable in a MATLAB Production Server instance.

`mcc -?` displays help.

Tip You can issue the `mcc` command either from the MATLAB command prompt or the DOS or UNIX command line.

Options

-a Add to Archive

Add a file to the CTF archive using

```
-a filename
```

to specify a file to be directly added to the CTF archive. Multiple `-a` options are permitted. MATLAB Compiler looks for these files on the MATLAB path, so specifying the full path name is optional. These files are not passed to `mbuild`, so you can include files such as data files.

If only a folder name is included with the `-a` option, the entire contents of that folder are added recursively to the CTF archive. For example:

```
mcc -m hello.m -a ./testdir
```

In this example, `testdir` is a folder in the current working folder. All files in `testdir`, as well as all files in subfolders of `testdir`, are added to the CTF archive, and the folder subtree in `testdir` is preserved in the CTF archive.

If a wildcard pattern is included in the file name, only the files in the folder that match the pattern are added to the CTF archive and subfolders of the given path are not processed recursively. For example:

```
mcc -m hello.m -a ./testdir/*
```

In this example, all files in `./testdir` are added to the CTF archive and subfolders under `./testdir` are not processed recursively.

```
mcc -m hello.m -a ./testdir/*.m
```

In this example, all files with the extension `.m` under `./testdir` are added to the CTF archive and subfolders of `./testdir` are not processed recursively.

All files added to the CTF archive using `-a` (including those that match a wildcard pattern or appear under a folder specified using `-a`) that do not appear on the MATLAB path at the time of compilation causes a path entry to be added to the deployed application's run-time path so that they appear on the path when the deployed application or component executes.

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 is created from the CTF file the first time the EXE file is executed. The `isdeployed` function is provided to help you accommodate this difference in deployed mode.

The `-a` switch also creates a `.auth` file for authorization purposes. It ensures that the executable looks for the DLL- and H-files in the *`exe_mcr\exe`* folder.

Caution

If you use the `-a` flag to include a file that is not on the MATLAB path, the folder containing the file is added to the MATLAB dependency analysis path. As a result, other files from that folder might be included in the compiled application.

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

Note If the `-a` flag is used to include custom Java classes, standalone applications work without any need to change the `classpath` as long as the Java class is not a member of a package. The same applies for JAR files. However, if the class being added is a member of a package, the MATLAB code needs to make an appropriate call to `javaaddpath` to 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 might contain other `-B` options. A bundle file can include replacement parameters for Compiler options that accept names and version numbers. See "Using Bundle Files to Build MATLAB Code" on page 11-9 for a list of the bundle files included with MATLAB Compiler.

-C Do Not Embed CTF Archive by Default

Override automatically embedding the CTF archive in C/C++ and `main`/`Winmain` shared libraries and standalone binaries by default.

-d Specified Folder for Output

Place output in a specified folder. Use

```
-d outFolder
```

to direct the generated files to *`outFolder`*.

-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 lets you use different ANSI compilers for different invocations of MATLAB Compiler. This option is a direct pass-through to `mbuild`.

-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 lets you 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 let you debug your MATLAB files with a C/C++ debugger.

-G Debug Only

Same as `-g`.

-I Add Folder 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>
```

sets up the search path so that `directory1` is searched first for MATLAB files, followed by `directory2`. This option is important for standalone compilation where the MATLAB path is not available.

-K Preserve Partial Output Files

Direct `mcc` to not delete output files if the compilation ends prematurely, due to error.

The default behavior of `mcc` is to dispose of any partial output if the command fails to execute successfully.

-M Direct Pass Through

Define compile-time options. Use

```
-M string
```

to pass `string` directly to `mbuild`. 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 lets you 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 you 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 Folder to Path

Use in conjunction with the 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 follow.

- 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

Provides MCR run-time options. The syntax is as follows:

```
-R option
```

Option	Description
`-logfile` *`filename`*	Specify a log file name.
`-nodisplay`	Suppress the MATLAB `nodisplay` run-time warning.
`-nojvm`	Do not use the Java Virtual Machine (JVM).

Option	Description
`-startmsg`	Customizable user message displayed at MCR initialization time.
`-completemsg`	Customizable user message displayed when MCR initialization is complete.

Note Not all `-R` options are available for all mcc targets.

Caution

When running on Mac OS X, 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 Context

The standard behavior for the MCR is that every instance of a class gets its own runtime context. This runtime context includes a global MATLAB workspace for variables such as the path and a base workspace for each function in the class. If multiple instances of a class are created, each instance gets an independent context. This ensures that changes made to the global, or base, workspace in one instance of the class does not effect other instances of the same class.

In a singleton MCR, all instances of a class share the runtime context. If multiple instances of a class are created, the use the runtime context created by the first instance. This saves start up time and some resources. However, any changes made to the global workspace or the base workspace by one instance impacts all of the class instances. For example, if `instance1` creates a global variable `A` in a singleton MCR, the `instance2` will be able to use variable `A`.

-T Specify Target Stage

Specify the output target phase and type.

Use the syntax `-T` *`target`* to define the output type. Target values are as follow.

Target	Description
`compile:exe`	Generate a C/C++ wrapper file plus compile C/C++ files to object form suitable for linking into a standalone application.
`compile:lib`	Generate a C/C++ wrapper file plus compile C/C++ files to object form suitable for linking into a shared library/DLL.
`link:exe`	Same as `compile:exe` plus links object files into a standalone application.
`link:lib`	Same as `compile:lib` plus links object files into a shared library/DLL.

-u Register COM Component for the Current User

Register COM component for the current user only on the development machine. The argument applies only for generic COM component and Microsoft Excel add-in targets only.

-v Verbose

Display the compilation steps, including:

- MATLAB Compiler version number
- The source file names as they are processed
- The names of the generated output files as they are created
- The invocation of `mbuild`

The `-v` option passes the `-v` option to `mbuild` and displays information about `mbuild`.

-w Warning Messages

Display warning messages. Use the syntax

```
-w option [:<msg>]
```

to control the display of warnings. This table lists the syntaxes.

Syntax	Description
`-w list`	Generate a table that maps `<string>` to warning message for use with `enable`, `disable`, and `error`. "Warning Messages" on page C-7, lists the same information.
`-w enable`	Enable complete warnings.
`-w disable[:<string>]`	Disable specific warnings associated with `<string>`. "Warning Messages" on page C-7, lists the `<string>` values. Omit the optional `<string>` to apply the `disable` action to all warnings.
`-w enable[:<string>]`	Enable specific warnings associated with `<string>`. "Warning Messages" on page C-7, lists the `<string>` values. Omit the optional `<string>` to apply the `enable` action to all warnings.
`-w error[:<string>]`	Treat specific warnings associated with `<string>` as an error. Omit the optional `<string>` to apply the `error` action to all warnings.

Syntax	Description
`-w off[:<string>] [<filename>]`	Turn warnings off for specific error messages defined by `<string>`. You can also narrow scope by specifying warnings be turned off when generated by specific `<filename>`s.
`-w on[:<string>] [<filename>]`	Turn warnings on for specific error messages defined by `<string>`. You can also narrow scope by specifying warnings be turned on when generated by specific `<filename>`s.

It is also possible to turn warnings on or off in your MATLAB code.

For example, to turn warnings off for deployed applications (specified using `isdeployed`) in your `startup.m`, you write:

```
if isdeployed
    warning off
end
```

To turn warnings on for deployed applications, you write:

```
if isdeployed
    warning on
end
```

-win32 Run in 32-Bit Mode

Use this option to build a 32-bit application on a 64-bit system *only* when the following are true:

- You have a 32-bit installation of MATLAB.
- You use the same MATLAB installation root (*`matlabroot`*) for both 32-bit and 64-bit versions of MATLAB.

- You are running from a Windows command line.

-Y License File

Use

```
-Y license.lic
```

to override the default license file with the specified argument.

mclGetLastErrorMessage

Purpose Last error message from unsuccessful function call

Syntax `const char* mclGetLastErrorMessage()`

Description This function returns a function error message (usually in the form of `false` or `-1`).

Example

```
char *args[] = { "-nodisplay" };
if(!mclInitializeApplication(args, 1))
{
    fprintf(stderr,
             An error occurred while initializing: \n %s  ,
            mclGetLastErrorMessage());
    return -1;
}
```

See Also `mclInitializeApplication` | `mclTerminateApplication` | `<library>Initialize[WithHandlers]` | `<library>Terminate`

Purpose Retrieve name of log file used by MCR

Syntax `const char* mclGetLogFileName()`

Description Use `mclGetLogFileName()` to retrieve the name of the log file used by the MCR. Returns a character string representing log file name used by MCR. For more information, see "MCR Startup Options" on page 9-28 in the User's Guide.

Examples

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

mclInitializeApplication

Purpose Set up application state shared by all (future) MCR instances created in current process

Syntax

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

Description MATLAB Compiler-generated standalone executables contain auto-generated code to call this function; users of shared libraries must call this function manually. Call only once per process. The function takes an array of strings (possibly of zero length) and a count containing the size of the string array. The string array may contain the following MATLAB command line switches, which have the same meaning as they do when used in MATLAB. :

- `-appendlogfile`
- `-Automation`
- `-beginfile`
- `-debug`
- `-defer`
- `-display`
- `-Embedding`
- `-endfile`
- `-fork`
- `-java`
- `-jdb`
- `-logfile`
- `-minimize`
- `-MLAutomation`
- `-noaccel`
- `-nodisplay`

- `-noFigureWindows`
- `-nojit`
- `-nojvm`
- `-noshelldde`
- `-nosplash`
- `-r`
- `-Regserver`
- `-shelldde`
- `-singleCompThread`
- `-student`
- `-Unregserver`
- `-useJavaFigures`
- `-mwvisual`
- `-xrm`

Caution

`mclInitializeApplication` must be called once only per process. Calling `mclInitializeApplication` more than once may cause your application to exhibit unpredictable or undesirable behavior.

Caution

When running on Mac, if `-nodisplay` is used as one of the options included in `mclInitializeApplication`, then the call to `mclInitializeApplication` must occur before calling `mclRunMain`.

Examples

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

```
const char *args[] = { "-nodisplay" };
if (! mclInitializeApplication(args, 1))
{
   fprintf(stderr,
            An error occurred while initializing: \n %s  ,
           mclGetLastErrorMessage());
   return -1;
}
```

See Also

`mclTerminateApplication`

How To

- "Initializing and Terminating Your Application with mclInitializeApplication and mclTerminateApplication" on page 13-13

Purpose

Determine if MCR was launched with instance of Java Virtual Machine (JVM)

Syntax

```
bool mclIsJVMEnabled()
```

Description

Use `mclIsJVMEnabled()` to determine if the MCR was launched with an instance of a Java Virtual Machine (JVM). Returns `true` if MCR is launched with a JVM instance, else returns `false`. For more information, see “MCR Startup Options” on page 9-28 in the User’s Guide.

Examples

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

mclIsMCRInitialized

Purpose Determine if MCR has been properly initialized

Syntax

```
bool mclIsMCRInitialized()
```

Description Use `mclIsMCRInitialized()` to determine whether or not the MCR has been properly initialized. Returns

- `true` if MCR is already initialized
- `false` if the MCR is not initialized

For more information, see "MCR Startup Options" on page 9-28.

Note This method can only be called once the MCR's proxy library has been initiated.

Examples

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

Purpose

Determine if `-nodisplay` mode is enabled

Syntax

```
bool mclIsNoDisplaySet()
```

Description

Use `mclIsNoDisplaySet()` to determine if `-nodisplay` mode is enabled. Returns `true` if `-nodisplay` is enabled, else returns `false`. For more information, see “MCR Startup Options” on page 9-28 in the User’s Guide.

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

Examples

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

mclmcrInitialize

Purpose

Initializes the MCR proxy library

Syntax

```
mclmcrInitialize();
```

Description

mclmcrInitialize is called before any other MATLAB APIs. It initializes the library used to create the MCR proxy used by all other MATLAB generated APIs.

See Also

`mclInitializeApplication`

Purpose

Mechanism for creating identical wrapper code across all compiler platform environments

Syntax

```
typedef int (*mclMainFcnType)(int, const char **);

int mclRunMain(mclMainFcnType run_main,
               int argc,
               const char **argv)
```

run_main

Name of function to execute after MCR set-up code.

argc

Number of arguments being passed to `run_main` function. Usually, `argc` is received by application at its `main` function.

argv

Pointer to an array of character pointers. Usually, `argv` is received by application at its `main` function.

Description

As you need to provide wrapper code when creating an application which uses a C or C++ shared library created by MATLAB Compiler, `mclRunMain` enables you with a mechanism for creating identical wrapper code across all MATLAB Compiler platform environments.

`mclRunMain` is especially helpful in Macintosh OS X environments where a run loop must be created for correct MCR operation.

When an OS X run loop is started, if `mclInitializeApplication` specifies the `-nojvm` or `-nodisplay` option, creating a run loop is a straight-forward process. Otherwise, you must create a Cocoa framework. The Cocoa frameworks consist of libraries, APIs, and Runtimes that form the development layer for all of Mac OS X.

Generally, the function pointed to by `run_main` returns with a pointer (return value) to the code that invoked it. When using Cocoa on the

Macintosh, however, when the function pointed to by `run_main` returns, the MCR calls exit before the return value can be received by the application, due to the inability of the underlying code to get control when Cocoa is shut down.

Caution

You should not use `mclRunMain` if your application brings up its own full graphical environment.

Note In non-Macintosh environments, `mclRunMain` acts as a wrapper and doesn't perform any significant processing.

Examples

Call using this basic structure:

```
int returncode = 0;
mclInitializeApplication(NULL,0);
returncode = mclRunMain((mclmainFcn)
                   my_main_function,0,NULL);
```

See Also

`mclInitializeApplication`

mclTerminateApplication

Purpose

Close down all MCR-internal application state

Syntax

```
bool mclTerminateApplication(void)
```

Description

Call this function once at the end of your program to close down all MCR-internal application state. Call only once per process. After you have called this function, you cannot call any further MATLAB Compiler-generated functions or any functions in any MATLAB library.

Caution

`mclTerminateApplication` must be called once only per process. Calling `mclTerminateApplication` more than once may cause your application to exhibit unpredictable or undesirable behavior.

Caution

`mclTerminateApplication` will close any visible or invisible figures before exiting. If you have visible figures that you would like to wait for, use `mclWaitForFiguresToDie`.

Examples

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

```
mclInitializeApplication(NULL,0);
if (!libmatrixInitialize()){
    fprintf(stderr,
             An error occurred while initializing: \n %s  ,
            mclGetLastErrorMessage());
    return -1;
}
```

At your program's exit point, call `mclTerminateApplication` to properly shut the application down:

```
mxDestroyArray(in1); in1=0;
mxDestroyArray(in2); in2 = 0;
mclTerminateApplication();
return 0;
```

See Also

`mclInitializeApplication`

How To

- “Initializing and Terminating Your Application with mclInitializeApplication and mclTerminateApplication” on page 13-13

Purpose

Enable deployed applications to process Handle Graphics events, enabling figure windows to remain displayed

Syntax

```
void mclWaitForFiguresToDie(HMCRINSTANCE instReserved)
```

Description

Calling `void mclWaitForFiguresToDie` enables the deployed application to process Handle Graphics events.

`NULL` is the only parameter accepted for the MCR instance (`HMCRINSTANCE instReserved`).

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

`mclWaitForFiguresToDie` blocks all open figures. This function runs until no visible figures remain. At that point, it displays a warning if there are invisible figures present. This function returns only when the last figure window is manually closed — therefore, this function should be called after the library launches at least one figure window. This function may be called multiple times.

If this function is not called, any figure windows initially displayed by the application briefly appear, and then the application exits.

Note `mclWaitForFiguresToDie` will block the calling program only for MATLAB figures. It will not block any Java GUIs, ActiveX controls, and other non-MATLAB GUIs unless they are embedded in a MATLAB figure window.

Examples

```
int run_main(int argc, const char** argv)
{
  int some_variable = 0;
  if (argc > 1)
    test_to_run = atoi(argv[1]);

    /* Initialize application */
  if( !mclInitializeApplication(NULL,0) )
  {
 fprintf(stderr,
     An error occurred while
      initializing: \n %s  ,
    mclGetLastErrorMessage());
 return -1;
}

if (test_to_run == 1 || test_to_run == 0)
{
 /* Initialize ax1ks library */
 if (!libax1ksInitialize())
 {
   fprintf(stderr,
      An error occurred while
      initializing: \n %s  ,
     mclGetLastErrorMessage());
   return -1;
 }
}

if (test_to_run == 2 || test_to_run == 0)
{
 /* Initialize simple library */
 if (!libsimpleInitialize())
 {
   fprintf(stderr,
      An error occurred while
      initializing: \n %s  ,
     mclGetLastErrorMessage());
   return -1;
 }
}

/* your code here
```

```
    /* your code here
    /* your code here
    /* your code here
    /*
    /* Block on open figures */
     mclWaitForFiguresToDie(NULL);
    /* Terminate libraries */
     if (test_to_run == 1 || test_to_run == 0)
      libax1ksTerminate();
     if (test_to_run == 2 || test_to_run == 0)
      libsimpleTerminate();
    /* Terminate application */
     mclTerminateApplication();
        return(0);
}
```

How To

- “Terminating Figures by Force In a Standalone Application” on page 11-23

mcrinstaller

Purpose

Display version and location information for MCR installer corresponding to current platform

Syntax

```
[INSTALLER_PATH, MAJOR, MINOR, PLATFORM, LIST]
= mcrinstaller;
```

Description

Displays information about available MCR installers using the format: `[INSTALLER_PATH, MAJOR, MINOR, PLATFORM, LIST] = mcrinstaller;` where:

- *`INSTALLER_PATH`* is the full path to the installer for the current platform.
- *`MAJOR`* is the major version number of the installer.
- *`MINOR`* is the minor version number of the installer.
- *`PLATFORM`* is the name of the current platform (returned by `COMPUTER(`*`arch`*`)`).
- *`LIST`* is a cell array of strings containing the full paths to MCR installers for other platforms. This list is non-empty only in a multi-platform MATLAB installation.

Note You must distribute the MATLAB Compiler Runtime library to your end users to enable them to run applications developed with MATLAB Compiler. Prebuilt MCR installers for all licensed platforms ship with MATLAB Compiler.

See "Working with the MCR" on page 9-14 for more information about the MCR installer.

Examples

Find MCR Installer Locations

Display locations of MCR Installers for platform. This example shows output for a `win64` system.

```
mcrinstaller

The WIN64 MCR Installer, version 7.16, is:
    X:\jobx\clusterc\current\matlab\toolbox\compiler\
                            deploy\win64\MCRInstaller.exe

MCR installers for other platforms are located in:
    X:\jobx\clusterc\current\matlab\toolbox\compiler\
                     deploy\win64
    win64 is the value of COMPUTER(win64) on
           the target machine.

For more information, read your local MCR Installer help.
Or see the online documentation at MathWorks' web site. (Page
                                              may load slowly.)

ans =

X:\jobx\clusterc\current\matlab\toolbox\compiler\
                            deploy\win64\MCRInstaller.exe
```

Purpose Determine version of installed MATLAB Compiler Runtime (MCR)

Syntax

```
[major, minor] = mcrversion;
```

Description The MCR version number consists of two digits, separated by a decimal point. This function returns each digit as a separate output variable: `[major, minor] = mcrversion;` Major and minor are returned as integers.

If the version number ever increases to three or more digits, call `mcrversion` with more outputs, as follows:

```
[major, minor, point] = mcrversion;
```

At this time, all outputs past "minor" are returned as zeros.

Typing only `mcrversion` will return the major version number only.

Examples

```
mcrversion
ans =
     7
```

setmcruserdata

Purpose

Associate MATLAB data value with string key

Syntax

`function` `setmcruserdata(`*`key`*`, `*`value`*`)`

Description

The *`function`* `setmcruserdata(`*`key`*`, `*`value`*`)` command is part of the MCR User Data interface API. For information about this function, as well as complete examples of usage, see “Using the MCR User Data Interface” on page 9-32.

Examples

In C++:

```
mxArray *key = mxCreateString("ParallelProfile");
mxArray *value = mxCreateString("\usr\userdir\config.settings");
if (!setmcruserdata(key, value))
{
    fprintf(stderr,
             Could not set MCR user data: \n %s   ,
            mclGetLastErrorMessage());
    return -1;
}
```

In C:

```
mxArray *key = mxCreateString("ParallelProfile");
mxArray *value = mxCreateString("\usr\userdir\config.settings");
if (!mlfSetmcruserdata(key, value))
{
    fprintf(stderr,
             Could not set MCR user data: \n %s   ,
            mclGetLastErrorMessage());
    return -1;
}
```

See Also

`getmcruserdata`

<library>Terminate

Purpose

Free 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.

Examples

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

```
/* Call the library intialization routine and ensure the
*  library was initialized properly. */
if (!libmatrixInitialize())
{
    fprintf(stderr,
             An error occurred while initializing: \n %s  ,
            mclGetLastErrorMessage());
    return -2;
}
else
    ...
```

Near the end of your program (but before calling `mclTerminateApplication`) free resources allocated by the MCR instance associated with library `libmatrix`:

```
/* Call the library termination routine */
libmatrixTerminate();
/* Free the memory created */
mxDestroyArray(in1); in1=0;
mxDestroyArray(in2); in2 = 0;
}
```

See Also

`<library>Initialize[WithHandlers]`

How To

- "Library Initialization and Termination Functions" on page 13-26

MATLAB Compiler Quick Reference

- "Common Uses of MATLAB® Compiler™ " on page A-2
- "mcc Command Arguments Listed Alphabetically" on page A-4
- "mcc Command Line Arguments Grouped by Task" on page A-8

Common Uses of MATLAB Compiler

In this section...
"Create a Standalone Application" on page A-2
"Create a Library" on page A-2

Create a Standalone Application

Example 1

To create a standalone application from `mymfile.m`, use

```
mcc -m mymfile
```

Example 2

To create a standalone application from `mymfile.m`, look for `mymfile.m` in the folder `/files/source`, and put the resulting C files and in `/files/target`, use

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

Example 3

To create a standalone application `mymfile1` from `mymfile1.m` and `mymfile2.m` using a single `mcc` call, use

```
mcc -m mymfile1 mymfile2
```

Create a Library

Example 1

To create a C shared library from `foo.m`, use

```
mcc -l foo.m
```

Example 2

To create a C shared library called `library_one` from `foo1.m` and `foo2.m`, use

```
mcc -W lib:library_one -T link:lib foo1 foo2
```

Note You can add the `-g` option to any of these for debugging purposes.

---

mcc Command Arguments Listed Alphabetically

Bold entries in the Comment column indicate default values.

Option	Description	Comment
`-a` *`filename`*	Add `filename` to the CTF archive.	None
`-b`	Generate Excel compatible formula function.	Requires MATLAB Builder EX
`-B filename[:arg[,arg]]`	Replace `-B filename` on the `mcc` command line with the contents of `filename`.	The file should contain only `mcc` command-line options. These are MathWorks included options files: • `-B csharedlib:foo` — C shared library • `-B cpplib:foo` — C++ library
`-c`	Generate C wrapper code.	Equivalent to `-T codegen`
`-C`	Directs `mcc` to not embed the CTF archive in C/C++ and main/Winmain shared libraries and standalone binaries by default.	See "MCR Component Cache and CTF Archive Embedding" on page 11-11 for more information.
`-d directory`	Place output in specified folder.	None

Option	Description	Comment
`-e`	Suppresses appearance of the MS-DOS Command Window when generating a standalone application.	Use `-e` in place of the `-m` option. Available for Windows only. Use with `-R` option to generate error logging. Equivalent to `-W WinMain -T link:exe` The standalone app compiler suppresses the MS-DOS command window by default. To unsuppress it, unselect **Do not require Windows Command Shell (console) for execution** in the app's **Additional Runtime Settings** area.
`-f filename`	Use the specified options file, `filename`, when calling `mbuild`.	`mbuild -setup` is recommended.
`-g`	Generate debugging information.	None
`-G`	Same as `-g`	None
`-I directory`	Add folder to search path for MATLAB files.	MATLAB path is automatically included when running from MATLAB, but not when running from a DOS/UNIX shell.
`-K`	Directs `mcc` to not delete output files if the compilation ends prematurely, due to error.	`mcc`'s default behavior is to dispose of any partial output if the command fails to execute successfully.
`-l`	Macro to create a function library.	Equivalent to `-W lib -T link:lib`
`-m`	Macro to generate a standalone application.	Equivalent to `-W main -T link:exe`
`-M string`	Pass string to `mbuild`.	Use to define compile-time options.

Option	Description	Comment
`-N`	Clear the path of all but a minimal, required set of folders.	None
`-o outputfile`	Specify name of final output file.	Adds appropriate extension
`-p directory`	Add `directory` to compilation path in an order-sensitive context.	Requires `-N` option

`-R` *option*	Specify run-time options for MCR.	*option* =
		`-nojvm`
		`-nodisplay`
		`-logfile` *filename*
		`-startmsg`
		`-completemsg` *filename*

`-S`	Create Singleton MCR.	For COM components only. Requires MATLAB Builder NE or MATLAB Builder EX.
`-u`	Registers COM component for current user only on development machine	Valid only for generic COM components and Microsoft Excel add-ins (requiring MATLAB Builder EX)
`-T`	Specify the output target phase and type.	Default is `codegen`.
`-v`	Verbose; display compilation steps.	None

`-w` *option*	Display warning messages.	*option* = `list`
		level
		level:*string*
		`where` *level* = `disable`
		`enable`
		`error`
		`error`
		`[off:`*string* `|` `on:`*string*`]`

Option	Description	Comment

`-W` *`type`*	Control the generation of function wrappers.	*`type`* = `main`
		`cpplib:<string>`
		`lib:<string>`
		`none`
		`com:compname,clname,version`

`-Y licensefile`	Use `licensefile` when checking out a MATLAB Compiler license.	None
`-?`	Display help message.	None

mcc Command Line Arguments Grouped by Task

Bold entries in the Comment column indicate default values.

COM Components

Option	Description	Comment
`-u`	Registers COM component for current user only on development machine	Valid only for generic COM components and Microsoft Excel add-ins (requiring MATLAB Builder EX)

CTF Archive

Option	Description	Comment
`-a` *`filename`*	Add `filename` to the CTF archive.	None
`-C`	Directs `mcc` to not embed the CTF archive in C/C++ and main/Winmain shared libraries and standalone binaries by default.	See "MCR Component Cache and CTF Archive Embedding" on page 11-11 for more information.

Debugging

Option	Description	Comment
`-g`	Generate debugging information.	None
`-G`	Same as `-g`	None
`-K`	Directs `mcc` to not delete output files if the compilation ends prematurely, due to error.	`mcc`'s default behavior is to dispose of any partial output if the command fails to execute successfully.
`-v`	Verbose; display compilation steps.	None

`-W` *`type`*	Control the generation of function wrappers.	*`type`* = `main`
		`cpplib:<string>`
		`lib:<string>`
		`none`
		`com:compname, clname,version`

`-?`	Display help message.	None

Dependency Function Processing

Option	Description	Comment
`-a` *`filename`*	Add `filename` to the CTF archive.	None

Licenses

Option	Description	Comment
`-Y licensefile`	Use `licensefile` when checking out a MATLAB Compiler license.	None

MATLAB Builder EX

Option	Description	Comment
`-b`	Generate Excel compatible formula function.	Requires MATLAB Builder EX
`-u`	Registers COM component for current user only on development machine	Valid only for generic COM components and Microsoft Excel add-ins (requiring MATLAB Builder EX)

MATLAB Path

Option	Description	Comment
`-I directory`	Add folder to search path for MATLAB files.	MATLAB path is automatically included when running from MATLAB, but not when running from a DOS/UNIX shell.
`-N`	Clear the path of all but a minimal, required set of folders.	None
`-p directory`	Add `directory` to compilation path in an order-sensitive context.	Requires `-N` option

mbuild

Option	Description	Comment
`-f filename`	Use the specified options file, `filename`, when calling `mbuild`.	`mbuild -setup` is recommended.
`-M string`	Pass string to `mbuild`.	Use to define compile-time options.

MATLAB Compiler Runtime (MCR)

Option	Description	Comment

`-R` *`option`*	Specify run-time options for MCR.	*`option`* `=`
		`-nojvm`
		`-nodisplay`
		`-logfile` *`filename`*
		`-startmsg`
		`-completemsg` *`filename`*

`-S`	Create Singleton MCR.	Requires MATLAB Builder NE

Override Default Inputs

Option	Description	Comment
`-B filename[:arg[,arg]]`	Replace `-B filename` on the `mcc` command line with the contents of `filename` (bundle).	The file should contain only `mcc` command-line options. These are MathWorks included options files: • `-B csharedlib:foo` — C shared library • `-B cpplib:foo` — C++ library

Override Default Outputs

Option	Description	Comment
`-d directory`	Place output in specified folder.	None
`-o outputfile`	Specify name of final output file.	Adds appropriate extension
`-e`	Suppresses appearance of the MS-DOS Command Window when generating a standalone application.	Use `-e` in place of the *`-m`* option. Available for Windows only. Use with `-R` option to generate error logging. Equivalent to `-W WinMain -T link:exe` The standalone app compiler suppresses the MS-DOS command window by default. To unsuppress it, unselect **Do not require Windows Command Shell (console) for execution** in the app's **Additional Runtime Settings** area.

Wrappers and Libraries

Option	Description	Comment
`-c`	Generate C wrapper code.	Equivalent to `-T codegen`
`-l`	Macro to create a function library.	Equivalent to `-W lib -T link:lib`
`-m`	Macro to generate a standalone application.	Equivalent to `-W main -T link:exe`

`-W` *`type`*	Control the generation of function wrappers.	*`type`* = `main`
		`cpplib:<string>`
		`lib:<string>`
		`none`
		`com:compname,clname,version`

B

Using MATLAB Compiler on Mac or Linux

- “Overview” on page B-2
- “Installing MATLAB® Compiler™ on Mac or Linux” on page B-3
- “Writing Applications for Mac or Linux” on page B-4
- “Building Your Application on Mac or Linux ” on page B-10
- “Testing Your Application on Mac or Linux” on page B-11
- “Set MCR Paths on Mac or Linux with Scripts” on page B-12

Overview

If you use MATLAB Compiler on Linux or Macintosh systems, use this appendix as a quick reference to common tasks.

Installing MATLAB Compiler on Mac or Linux

In this section...

"Installing MATLAB® Compiler™" on page B-3
"Custom Configuring Your Options File" on page B-3
"Install Apple Xcode from DVD on Maci64" on page B-3

Installing MATLAB Compiler

See "Supported ANSI C and C++ UNIX Compilers" on page 2-2 for general installation instructions and information about supported compilers.

Custom Configuring Your Options File

To modify the current linker settings, or disable a particular set of warnings, locate your options file for your "UNIX Operating System" on page 2-6, and view instructions for "Changing the Options File" on page 2-6.

Install Apple Xcode from DVD on Maci64

When installing on 64-bit Macintosh systems, install the Apple Xcode from the installation DVD.

Writing Applications for Mac or Linux

In this section...

"Objective-C/C++ Applications for Apple's Cocoa API" on page B-4

"Where's the Example Code?" on page B-4

"Preparing Your Apple Xcode Development Environment" on page B-4

"Build and Run the Sierpinski Application" on page B-5

"Running the Sierpinski Application" on page B-7

Objective-C/C++ Applications for Apple's Cocoa API

Apple Xcode, implemented in the Objective-C language, is used to develop applications using the Cocoa framework, the native object-oriented API for the Mac OS X operating system.

This article details how to deploy a graphical MATLAB application with Objective C and Cocoa, and then deploy it using MATLAB Compiler.

Where's the Example Code?

You can find example Apple Xcode, header, and project files in `matlabroot/extern/examples/compiler/xcode`.

Preparing Your Apple Xcode Development Environment

To run this example, you should have prior experience with the Apple Xcode development environment and the Cocoa framework.

The example in this article is ready to build and run. However, before you build and run your own applications, you must do the following (as has been done in our example code):

1. Build the shared library with MATLAB Compiler using either the Deployment Tool or `mcc`.

2 Compile application code against the component's header file and link the application against the component library and `libmwmclmcrrt`. See "Set MCR Paths on Mac or Linux with Scripts" on page B-12 and "Solving Problems Related to Setting MCR Paths on Mac or Linux" on page B-12 for information about and MCR paths and `libmwmclmcrrt`.

3 In your Apple Xcode project:

- Specify `mcc` in the project target (Build Component Library in the example code).
- Specify target settings in `HEADER_SEARCH_PATHS`.
  - Specify directories containing the component header.
  - Specify the path *`matlabroot`*`/extern/include`.
  - Define `MWINSTALL_ROOT`, which establishes the install route using a relative path.
- Set `LIBRARY_SEARCH_PATHS` to any directories containing the component's shared library, as well as to the path *`matlabroot`*`/runtime/maci64`.

Build and Run the Sierpinski Application

In this example, you deploy the graphical Sierpinski function (`sierpinski.m`, located at *`matlabroot`*`/extern/examples/compiler`).

```
function [x, y] = sierpinski(iterations, draw)
% SIERPINSKI Calculate (optionally draw) the points
%  in Sierpinski's triangle

% Copyright 2004 The MathWorks, Inc.

    % Three points defining a nice wide triangle
    points = [0.5 0.9 ; 0.1 0.1 ; 0.9 0.1];

    % Select an initial point
    current = rand(1, 2);

    % Create a figure window
    if (draw == true)
        f = figure;
        hold on;
```

```
   end

   % Pre-allocate space for the results, to improve performance
   x = zeros(1,iterations);
   y = zeros(1,iterations);

   % Iterate
   for i = 1:iterations

       % Select point at random
       index = floor(rand * 3) + 1;

       % Calculate midpoint between current point and random point
       current(1) = (current(1) + points(index, 1)) / 2;
       current(2) = (current(2) + points(index, 2)) / 2;

       % Plot that point
       if draw, line(current(1),current(2));, end
x(i) = current(1);
       y(i) = current(2);

   end

   if (draw)
       drawnow;
   end
```

1 Using the Mac Finder, locate the Apple Xcode project (*`matlabroot`*`/extern/examples/compiler/xcode`). Copy files to a working directory to run this example, if needed.

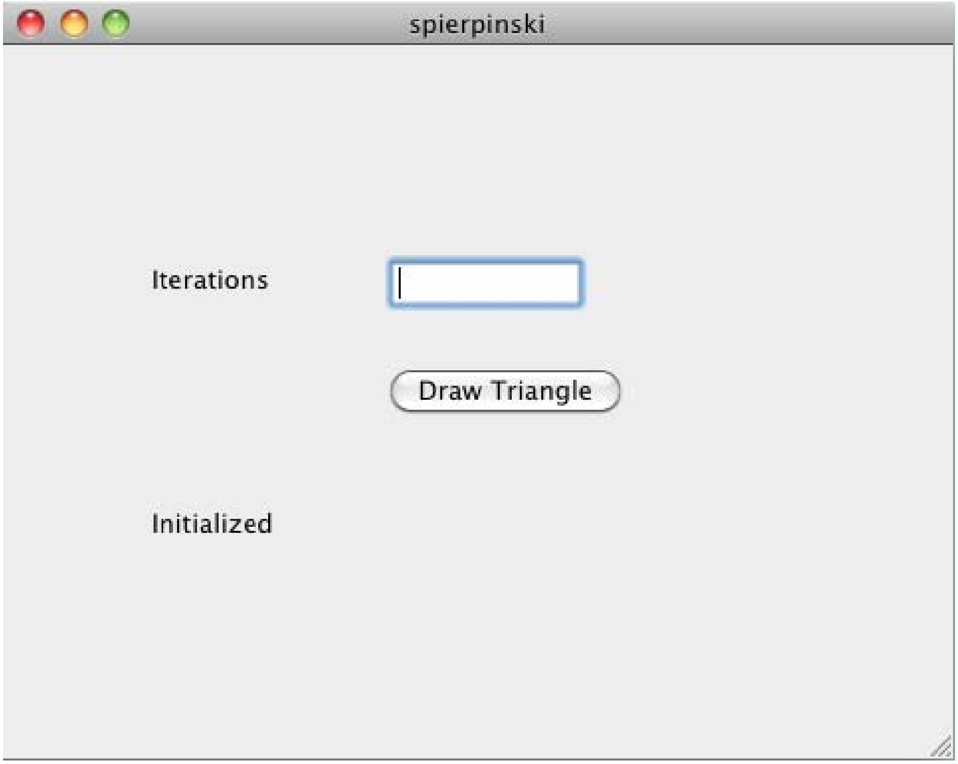
2 Open `sierpinski.xcodeproj`. The development environment starts.

3 In the **Groups and Files** pane, select **Targets**.

4 Click **Build and Run**. The make file runs that launches MATLAB Compiler (`mcc`).

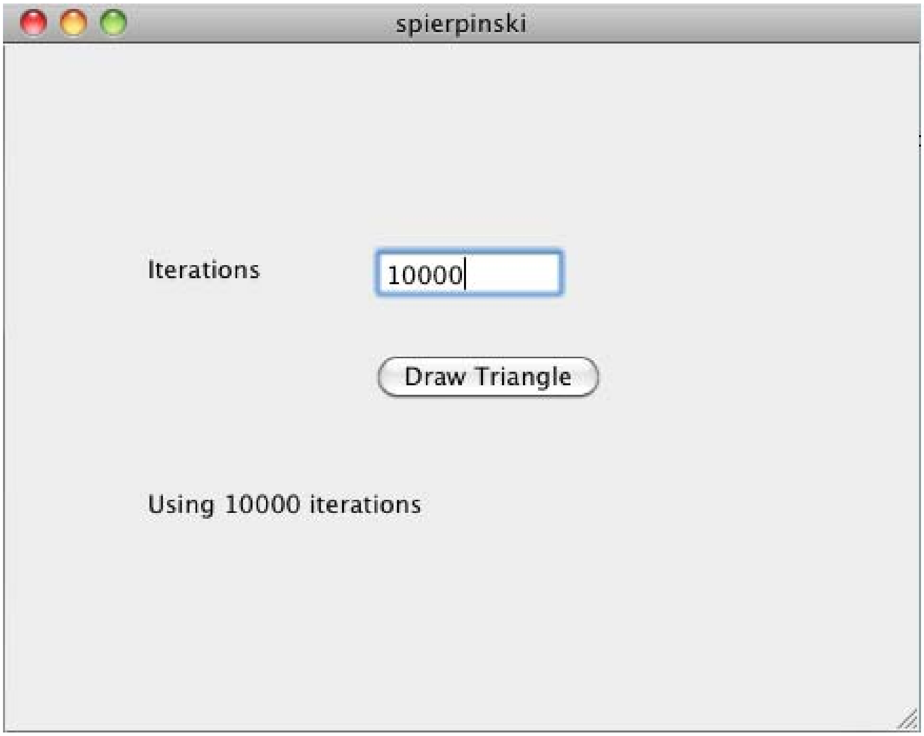
Running the Sierpinski Application

Run the **Sierpinski** application from the build output directory. The following GUI appears:


spierpinski
Iterations
Draw Triangle
Initialized

MATLAB Sierpinski Function Implemented in the Mac Cocoa Environment

1 In the **Iterations** field, enter an integer such as 10000:


spierpinski
Iterations
Draw Triangle

10000

Using 10000 iterations

2 Click **Draw Triangle**. The following figure appears:

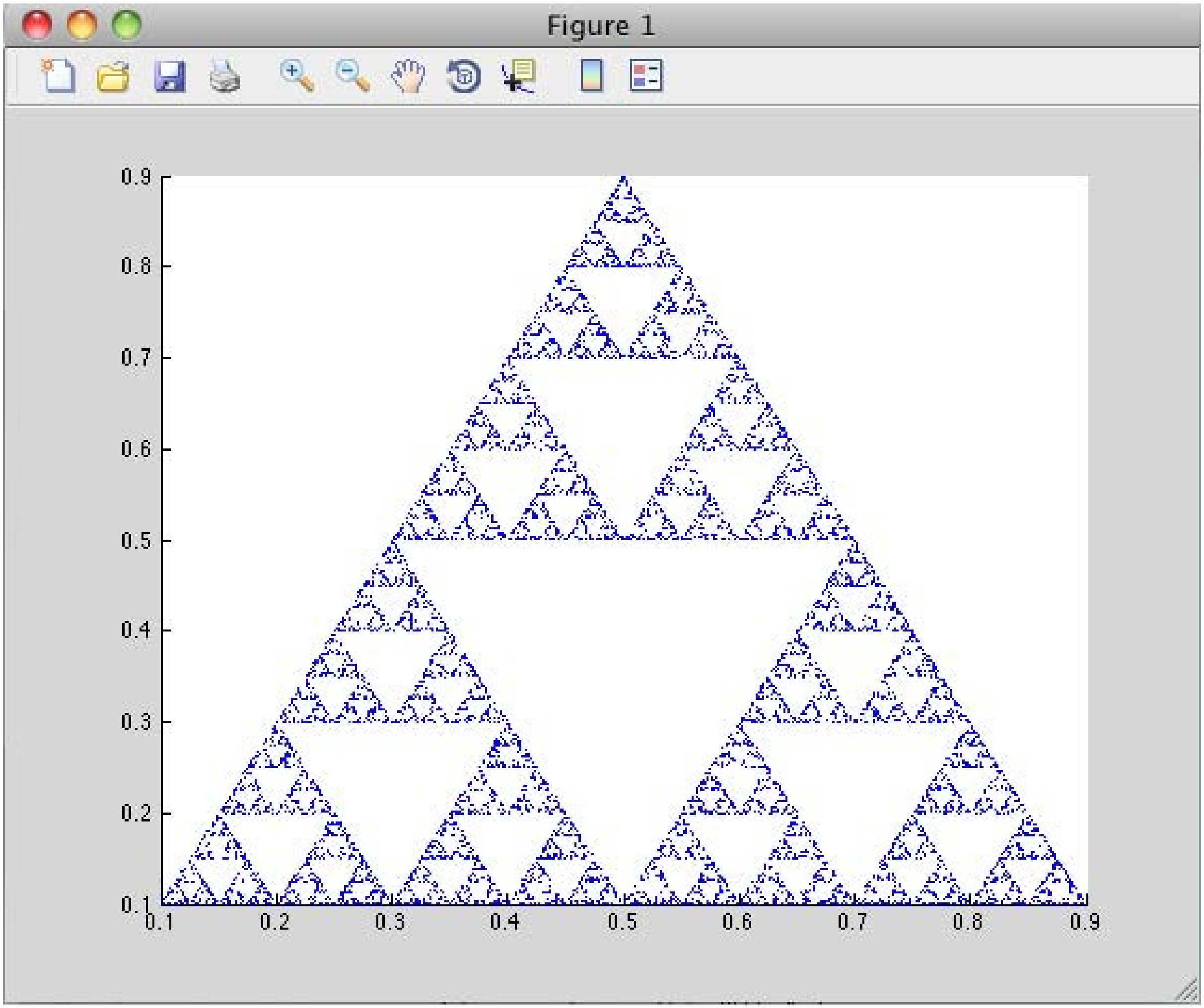


Figure 1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

Building Your Application on Mac or Linux

In this section...

"Compiling Your Application with the Compiler Apps" on page B-10

"Compiling Your Application with the Command Line" on page B-10

Compiling Your Application with the Compiler Apps

When running a graphical interface from your Mac or Linux desktop, use "Create and Install a Standalone Application from MATLAB Code" on page 1-12 as a template for building a standalone application with the Application Compiler. Use "Create a C/C++ Shared Library from MATLAB Code" on page 1-21 for creating a shared library with the Library Compiler.

Compiling Your Application with the Command Line

For compiling your application at the command line, there are separate Macintosh and non-Macintosh instructions for Mac or Linux platforms.

On Non-Mac i64 Platforms

Use the section "Input and Output Files" on page 8-2 for lists of files produced and supplied to `mcc` when building a "Standalone Executable" on page 8-2, "C Shared Library" on page 8-3, or "C++ Shared Library" on page 8-5.

On Maci64

Use the section "Input and Output Files" on page 8-2 for lists of files produced and supplied to `mcc` when building a "Macintosh 64 (Maci64)" on page 8-7 application.

Testing Your Application on Mac or Linux

On Windows, deployed applications automatically modify the system `PATH` variable.

On Mac OS X or Linux, deployed applications do not modify the system `PATH` variable. You must perform this step manually.

Set MCR Paths on Mac or Linux with Scripts

When you build applications, associated shell scripts (`run_`*`application`*`.sh`) are automatically generated in the same folder as your binary. By running these scripts, you can conveniently set the path to your MCR location.

Solving Problems Related to Setting MCR Paths on Mac or Linux

Use the following to solve common problems and issues:

I tried running SETENV on Mac and the command failed

If the `setenv` command fails with a message similar to `setenv:  command not found` or `setenv:  not found`, you are not using a C Shell command interpreter (such as `csh` or `tcsh`).

Set the environment variables using the `export` command using the format `export` *`my_variable=my_value`*.

For example, to set `DYLD_LIBRARY_PATH`, run the following command:

```
export DYLD_LIBRARY_PATH = mcr_root/v711/runtime/maci64:mcr_root/
...
```

My Mac application fails with “Library not loaded” or “Image not found” even though my EVs are set

If you set your environment variables, you may still receive the following message when you run your application:

```
imac-joe-user:~ joeuser$ /Users/joeuser/Documents/MATLAB/Dip/Dip ; exit;
dyld: Library not loaded: @loader_path/libmwmclmcrrt.7.11.dylib
Referenced from: /Users/joeuser/Documents/MATLAB/Dip/Dip
Reason: image not found
Trace/BPT trap
logout
```

You may have set your environment variables initially, but they were not set up as persistent variables. Do the following:

1 In your home directory, open a file such as `.bashrc` or `.profile` file in your log-in shell.

2 In either of these types of log-in shell files, add commands to set your environment variables so that they persist. For example, to set `DYLD_LIBRARY_PATH` in this manner, you enter the following in your file:

```
# Setting PATH for MCR

DYLD_LIBRARY_PATH=/Users/joeuser/Desktop/mcr/v711/runtime/maci64:
/Users/joeuser/Desktop/mcr/v711/sys/os/maci64:/Users/joeuser/Desktop/
mcr//v711/bin/maci64
export DYLD_LIBRARY_PATH

?
```

Note The `DYLD_LIBRARY_PATH=` statement is one statement that must be entered as a single line. The statement is shown on different lines, in this example, for readability only.

Error and Warning Messages

- "About Error and Warning Messages" on page C-2
- "Compile-Time Errors" on page C-3
- "Warning Messages" on page C-7
- "Dependency Analysis Errors" on page C-10

About Error and Warning Messages

This appendix lists and describes error messages and warnings generated by MATLAB Compiler. Compile-time messages are generated during the compile or link phase. It is useful to note that most of these compile-time error messages should not occur if the MATLAB software can successfully execute the corresponding MATLAB file.

Use this reference to:

- Confirm that an error has been reported
- Determine possible causes for an error
- Determine possible ways to correct an error

When using MATLAB Compiler, if you receive an internal error message, record the specific message and report it to Technical Support at `http://www.mathworks.com/contact_TS.html`.

Compile-Time Errors

Error: An error occurred while shelling out to mex/mbuild (error code = errorno). Unable to build (specify the -v option for more information). MATLAB Compiler reports this error if `mbuild` or `mex` generates an error.

Error: An error occurred writing to file "filename": reason. The file can not be written. The reason is provided by the operating system. For example, you may not have sufficient disk space available to write the file.

Error: Cannot write file "filename" because MCC has already created a file with that name, or a file with that name was specified as a command line argument. MATLAB Compiler has been instructed to generate two files with the same name. For example:

```
mcc -W lib:liba liba -t  % Incorrect
```

Error: Could not check out a Compiler license. No additional MATLAB Compiler licenses are available for your workgroup.

Error: Initializing preferences required to run the application. The `.ctf` file and the corresponding target (standalone application or shared library) created using MATLAB Compiler do not match. Ensure that the `.ctf` file and the target file are created as output from the same `mcc` command. Verify the time stamp of these files to ensure they were created at the same time. Never combine the `.ctf` file and the target application created during execution of different `mcc` commands.

Error: File: "filename" not found. A specified file can not be found on the path. Verify that the file exists and that the path includes the file's location. You can use the `-I` option to add a folder to the search path.

Error: File: "filename" is a script MATLAB file and cannot be compiled with the current Compiler. MATLAB Compiler cannot compile script MATLAB files. To learn how to convert script MATLAB files to function MATLAB files, see "Converting Script MATLAB Files to Function MATLAB Files" on page 11-17.

Error: File: filename Line: # Column: # A variable cannot be made storageclass1 after being used as a storageclass2. You cannot change a variable's storage class (global/local/persistent). Even though MATLAB allows this type of change in scope, MATLAB Compiler does not.

Error: Found illegal whitespace character in command line option: "string". The strings on the left and right side of the space should be separate arguments to MCC. For example:

```
mcc('-m', '-v', 'hello')% Correct
mcc('-m -v', 'hello')   % Incorrect
```

Error: Improper usage of option -optionname. Type "mcc -?" for usage information. You have incorrectly used a MATLAB Compiler option. For more information about MATLAB Compiler options, see "mcc Command Arguments Listed Alphabetically" on page A-4, or type `mcc -?` at the command prompt.

Error: libraryname library not found. MATLAB has been installed incorrectly.

Error: No source files were specified (-? for help). You must provide MATLAB Compiler with the name of the source file(s) to compile.

Error: "optionname" is not a valid -option option argument. You must use an argument that corresponds to the option. For example:

```
mcc -W main ... % Correct
mcc -W mex  ... % Incorrect
```

Error: Out of memory. Typically, this message occurs because MATLAB Compiler requests a larger segment of memory from the operating system than is currently available. Adding additional memory to your system can alleviate this problem.

Error: Previous warning treated as error. When you use the `-w error` option, this error appears immediately after a warning message.

Error: The argument after the -option option must contain a colon. The format for this argument requires a colon. For more information, see "mcc Command Arguments Listed Alphabetically" on page A-4, or type `mcc -?` at the command prompt.

Error: The environment variable MATLAB must be set to the MATLAB root directory. On UNIX, the `MATLAB` and `LM_LICENSE_FILE` variables must be set. The `mcc` shell script does this automatically when it is called the first time.

Error: The license manager failed to initialize (error code is errornumber). You do not have a valid MATLAB Compiler license or no additional MATLAB Compiler licenses are available.

Error: The option -option is invalid in modename mode (specify -? for help). The specified option is not available.

Error: The specified file "filename" cannot be read. There is a problem with your specified file. For example, the file is not readable because there is no read permission.

Error: The -optionname option requires an argument (e.g. "proper_example_usage"). You have incorrectly used a MATLAB Compiler option. For more information about MATLAB Compiler options, see "mcc Command Arguments Listed Alphabetically" on page A-4, or type `mcc -?` at the command prompt.

Error: -x is no longer supported. MATLAB Compiler no longer generates MEX-files because there is no longer any performance advantage to doing so. The MATLAB JIT Accelerator makes compilation for speed obsolete.

Error: Unable to open file "filename":<string>. There is a problem with your specified file. For example, there is no write permission to the output folder, or the disk is full.

Error: Unable to set license linger interval (error code is errornumber). A license manager failure has occurred. Contact Technical Support with the full text of the error message.

Error: Unknown warning enable/disable string: warningstring. `-w enable:`, `-w disable:`, and `-w error:` require you to use one of the warning string identifiers listed in "Warning Messages" on page C-7.

Error: Unrecognized option: -option. The option is not a valid option. See "mcc Command Arguments Listed Alphabetically" on page A-4, for a complete list of valid options for MATLAB Compiler, or type `mcc -?` at the command prompt.

Warning Messages

This section lists the warning messages that MATLAB Compiler can generate. Using the `-w` option for `mcc`, you can control which messages are displayed. Each warning message contains a description and the warning message identifier string (in parentheses) that you can enable or disable with the `-w` option. For example, to produce an error message if you are using a trial MATLAB Compiler license to create your standalone application, you can use:

```
mcc -w error:trial_license -mvg hello
```

To enable all warnings except those generated by the `save` command, use:

```
mcc -w enable -w disable:trial_license ...
```

To display a list of all the warning message identifier strings, use:

```
mcc -w list -m mfilename
```

For additional information about the `-w` option, see "mcc Command Arguments Listed Alphabetically" on page A-4.

Warning: File: filename Line: # Column: # The #function pragma expects a list of function names. *(pragma_function_missing_names)* This pragma informs MATLAB Compiler that the specified function(s) provided in the list of function names will be called through an `feval` call. This will automatically compile the selected functions.

Warning: MATLAB file "filename" was specified on the command line with full path of "pathname", but was found on the search path in directory "directoryname" first. (*specified_file_mismatch*) MATLAB Compiler detected an inconsistency between the location of the MATLAB file as given on the command line and in the search path. MATLAB Compiler uses the location in the search path. This warning occurs when you specify a full path name on the `mcc` command line and a file with the same base name (file name) is found earlier on the search path. This warning is issued in the following example if the file `afile.m` exists in both `dir1` and `dir2`:

```
mcc -m -I /dir1 /dir2/afile.m
```

Warning: The file filename was repeated on MATLAB Compiler command line. *(repeated_file)* This warning occurs when the same file name appears more than once on the compiler command line. For example:

```
mcc -m sample.m sample.m % Will generate the warning
```

Warning: The name of a shared library should begin with the letters "lib". "libraryname" doesn't. *(missing_lib_sentinel)* This warning is generated if the name of the specified library does not begin with the letters "lib". This warning is specific to UNIX and does not occur on the Windows operating system. For example:

```
mcc -t -W lib:liba -T link:lib a0 a1 % No warning
mcc -t -W lib:a -T link:lib a0 a1    % Will generate a warning
```

Warning: All warnings are disabled. *(all_warnings)* This warning displays all warnings generated by MATLAB Compiler. This warning is disabled.

Warning: A line has num1 characters, violating the maximum page width (num2). *(max_page_width_violation)* This warning is generated if there are lines that exceed the maximum page width, `num2`. This warning is disabled.

Warning: The option -optionname is ignored in modename mode (specify -? for help). *(switch_ignored)* This warning is generated if an option is specified on the `mcc` command line that is not meaningful in the specified mode. This warning is enabled.

Warning: Unrecognized Compiler pragma "pragmaname". *(unrecognized_pragma)* This warning is generated if you use an unrecognized pragma. This warning is enabled.

Warning: "functionname1" is a MEX- or P-file being referenced from "functionname2". *(mex_or_p_file)* This warning is generated if `functionname2` calls `functionname1`, which is a MEX- or P-file. This warning is enabled.

Note A link error is produced if a call to this function is made from standalone code.

Trial Compiler license. The generated application will expire 30 days from today, on date. *(trial_license)* This warning displays the date that the deployed application will expire. This warning is enabled.

Dependency Analysis Errors

In this section...

"MCR/Dispatcher Errors" on page C-10
"XML Parser Errors" on page C-10

MCR/Dispatcher Errors

These errors originate directly from the MCR/Dispatcher. If one of these error occurs, report it to Technical Support at MathWorks at `http://www.mathworks.com/contact_TS.html`.

XML Parser Errors

These errors appear as

```
depfun Error: XML error: <message>
```

Where `<message>` is a message returned by the XML parser. If this error occurs, report it to Technical Support at MathWorks at `http://www.mathworks.com/contact_TS.html`.

C++ Utility Library Reference

Data Conversion Restrictions for the C++ MWArray API

Currently, returning a Java object to your application, from a compiled MATLAB function, is unsupported.

Primitive Types

The `mwArray` API supports all primitive types that can be stored in a MATLAB array. This table lists all the types.

Type	**Description**	**mxClassID**
`mxChar`	Character type	`mxCHAR_CLASS`
`mxLogical`	Logical or Boolean type	`mxLOGICAL_CLASS`
`mxDouble`	Double-precision floating-point type	`mxDOUBLE_CLASS`
`mxSingle`	Single-precision floating-point type	`mxSINGLE_CLASS`
`mxInt8`	1-byte signed integer	`mxINT8_CLASS`
`mxUint8`	1-byte unsigned integer	`mxUINT8_CLASS`
`mxInt16`	2-byte singed integer	`mxINT16_CLASS`
`mxUint16`	2-byte unsigned integer	`mxUINT16_CLASS`
`mxInt32`	4-byte signed integer	`mxINT32_CLASS`
`mxUint32`	4-byte unsigned integer	`mxUINT32_CLASS`
`mxInt64`	8-byte signed integer	`mxINT64_CLASS`
`mxUint64`	8-byte unsigned integer	`mxUINT64_CLASS`

C++ Utility Classes

Purpose

String class used by the `mwArray` API to pass string data as output from certain methods

Description

The `mwString` class is a simple string class used by the `mwArray` API to pass string data as output from certain methods.

Required Headers

- `mclcppclass.h`
- `mclmcrrt.h`

Tip MATLAB Compiler automatically includes these header files in the header file generated for your MATLAB functions.

Constructors

mwString()

Description

Create an empty string.

mwString(char* str)

Description

Create a new string and initialize the string's data with the supplied char buffer.

Arguments

`char* str`	Null terminated character buffer

mwString(mwString& str)

Description

Create a new string and initialize the string's data with the contents of the supplied string.

Arguments

`mwString& str`	Initialized `mwString` instance

Methods

int Length() const

Description

Return the number of characters in string.

Example

```
mwString str("This is a string");
int len = str.Length();
```

Operators

operator const char* () const

Description

Return a pointer to internal buffer of string.

Example

```
mwString str("This is a string");
const char* pstr = (const char*)str;
```

mwString& operator=(const mwString& str)

Description

Copy the contents of one string into a new string.

Arguments

`mwString& str`	Initialized `mwString` instance to copy

Example

```
mwString str("This is a string");
mwString new_str = str;
```

mwString& operator=(const char* str)

Description

Copy the contents of a null terminated character buffer into a new string.

Arguments

`char* str`	Null terminated character buffer to copy

Example

```
const char* pstr = "This is a string";
mwString str = pstr;
```

bool operator==(const mwString& str) const

Description

Test two `mwString` instances for equality. If the characters in the string are the same, the instances are equal.

Arguments

`mwString& str`	Initialized `mwString` instance

Example

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str == str2);
```

bool operator!=(const mwString& str) const

Description

Test two `mwString` instances for inequality. If the characters in the string are not the same, the instances are inequal.

Arguments

`mwString& str`	Initialized `mwString` instance

Example

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str != str2);
```

bool operator<(const mwString& str) const

Description

Compare two strings and return `true` if the first string is lexicographically less than the second string.

Arguments

`mwString& str`	Initialized `mwString` instance

Example

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str < str2);
```

bool operator<=(const mwString& str) const

Description

Compare two strings and return `true` if the first string is lexicographically less than or equal to the second string.

Arguments

`mwString& str`	Initialized `mwString` instance

Example

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str <= str2);
```

bool operator>(const mwString& str) const

Description

Compare two strings and return `true` if the first string is lexicographically greater than the second string.

Arguments

`mwString& str`	Initialized `mwString` instance

Example

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str > str2);
```

bool operator>=(const mwString& str) const

Description

Compare two strings and return `true` if the first string is lexicographically greater than or equal to the second string.

Arguments

`mwString& str`	Initialized `mwString` instance

Example

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str >= str2);
```

friend std::ostream& operator<<(std::ostream& os, const mwString& str)

Description

Copy contents of input string to specified `ostream`.

Arguments

`std::ostream& os`	Initialized `ostream` instance to copy sting into
`mwString& str`	Initialized `mwString` instance to copy

Example

```
#include <ostream>
mwString str("This is a string");
std::cout << str << std::endl;
```

mwException

Purpose

Exception type used by the `mwArray` API and the C++ interface functions

Description

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`.

Required Headers

- `mclcppclass.h`
- `mclmcrrt.h`

Tip MATLAB Compiler automatically includes these header files in the header file generated for your MATLAB functions.

Constructors

mwException()

Description

Construct new `mwException` with default error message.

mwException(char* msg)

Description

Create an `mwException` with a specified error message.

Arguments

`char* msg`	Null terminated character buffer to use as the error message

mwException(mwException& e)

Description

Create a copy of an `mwException`.

Arguments

`mwException& e`	Initialized `mwException` instance to copy

mwException(std::exception& e)

Description

Create new `mwException` from existing `std::exception`.

Arguments

`std::exception& e`	`std::exception` to copy

Methods

char* what() const throw()

Description

Return the error message contained in this exception.

Example

```
try
{
    ...
}
catch (const std::exception& e)
{
    std::cout << e.what() << std::endl;
}
```

void print_stack_trace()

Description

Print the stack trace to `std::cerr`.

Operators

mwException& operator=(const mwException& e)

Description

Copy the contents of one exception into a new exception.

Arguments

`mwException& e`	An initialized `mwException` instance to copy

Example

```
try
{
    ...
}
catch (const mwException& e)
{
    mwException e2 = e;
    throw e2;
}
```

mwException& operator=(const std::exception& e)

Description

Copy the contents of one exception into a new exception.

Arguments

`std::exception& e`	`std::exception` to copy

Example

```
try
{
    ...
}
catch (const std::exception& e)
{
    mwException e2 = e;
    throw e2;
}
```

Purpose

Class used to pass input/output arguments to MATLAB Compiler generated C++ interface functions

Description

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.

Required Headers

- `mclcppclass.h`
- `mclmcrrt.h`

Tip MATLAB Compiler automatically includes these header files in the header file generated for your MATLAB functions.

Constructors

mwArray()

Description

Construct empty array of type `mxDOUBLE_CLASS`.

mwArray(mxClassID mxID)

Description

Construct empty array of specified type.

Arguments

`mxClassID mxID`	Valid `mxClassID` specifying the type of array to construct. See the “Work with mxArrays” for more information on `mxClassID`.

mwArray(mwSize num_rows, mwSize num_cols, mxClassID mxID, mxComplexity cmplx = mxREAL)

Description

Create a 2–D matrix of the specified type and complexity. For numeric types, the matrix can be either real or complex. 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.

Arguments

`mwSize num_rows`	Number of rows in the array
`mwSize num_cols`	Number of columns in the array
`mxClassID mxID`	Valid `mxClassID` specifying the type of array to construct. See the “Work with mxArrays” for more information on `mxClassID`.
`mxComplexity cmplx`	Complexity of the array to create. Valid values are `mxREAL` and `mxCOMPLEX`. The default value is `mxREAL`.

mwArray(mwSize num_dims, const mwSize* dims, mxClassID mxID, mxComplexity cmplx = mxREAL)

Description

Create an `n`-dimensional array of the specified type and complexity. For numeric types, the array can be either real or complex. 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.

Arguments

`mwSize num_dims`	Number of dimensions in the array
`const mwSize* dims`	Dimensions of the array
`mxClassID mxID`	Valid `mxClassID` specifying the type of array to construct. See the "Work with mxArrays" for more information on `mxClassID`.
`mxComplexity cmplx`	Complexity of the array to create. Valid values are `mxREAL` and `mxCOMPLEX`. The default value is `mxREAL`.

mwArray(const char* str)

Description

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.

Arguments

`const char* str`	Null-terminated character buffer used to initialize the array

mwArray(mwSize num_strings, const char** str)

Description

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`.

Arguments

`mwSize num_strings`	Number of strings in the input array
`const char** str`	Array of null-terminated strings

mwArray(mwSize num_rows, mwSize num_cols, int num_fields, const char** fieldnames)

Description

Create a matrix of type `mxSTRUCT_CLASS`, with the specified field names. All elements are initialized with empty cells.

Arguments

`mwSize num_rows`	Number of rows in the array
`mwSize num_cols`	Number of columns in the array
`int num_fields`	Number of fields in the `struct` matrix.
`const char** fieldnames`	Array of null-terminated strings representing the field names

mwArray(mwSize num_dims, const mwSize* dims, int num_fields, const char** fieldnames)

Description

Create an `n`-dimensional array of type `mxSTRUCT_CLASS`, with the specified field names. All elements are initialized with empty cells.

Arguments

`mwSize num_dims`	Number of dimensions in the array
`const mwSize* dims`	Dimensions of the array
`int num_fields`	Number of fields in the `struct` matrix.
`const char** fieldnames`	Array of null-terminated strings representing the field names

mwArray(const mwArray& arr)

Description

Create a deep copy of an existing array.

Arguments

`mwArray& arr`	`mwArray` to copy

mwArray(<type> re)

Description

Create a real scalar array.

The scalar array is created with the type of the input argument.

Arguments

`<type> re`	Scalar value to initialize the array. `<type>` can be any of the following: • `mxDouble` • `mxSingle` • `mxInt8` • `mxUint8` • `mxInt16` • `mxUint16` • `mxInt32` • `mxUint32` • `mxInt64` • `mxUint64` • `mxLogical`

mwArray(<type> re, <type> im)

Description

Create a complex scalar array.

The scalar array is created with the type of the input argument.

Arguments

`<type> re`	Scalar value to initialize the real part of the array
`<type> im`	Scalar value to initialize the imaginary part of the array

`<type>` can be any of the following:

- `mxDouble`
- `mxSingle`
- `mxInt8`
- `mxUint8`
- `mxInt16`
- `mxUint16`
- `mxInt32`
- `mxUint32`
- `mxInt64`
- `mxUint64`
- `mxLogical`

Methods

mwArray Clone() const

Description

Create a new array representing deep copy of array.

Example

```
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray b = a.Clone();
```

mwArray SharedCopy() const

Description

Create a shared copy of an existing array. The new array and the original array both point to the same data.

Example

```
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray b = a.SharedCopy();
```

mwArray Serialize() const

Description

Serialize an array into bytes. A 1-by-`n` numeric matrix of type `mxUINT8_CLASS` is returned containing the serialized data. The data can be deserialized back into the original representation by calling `mwArray::Deserialize()`.

Example

```
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray b = a.Serialize();
```

mxClassID ClassID() const

Description

Determine the type of the array. See the “Work with mxArrays” for more information on `mxClassID`.

Example

```
mwArray a(2, 2, mxDOUBLE_CLASS);
mxClassID id = a.ClassID();
```

int ElementSize() const

Description

Determine the size, in bytes, of an element of array type.

Example

```
mwArray a(2, 2, mxDOUBLE_CLASS);
int size = a.ElementSize();
```

size_t ElementSize() const

Description

Determine the size, in bytes, of an element of array type.

Example

```
mwArray a(2, 2, mxDOUBLE_CLASS);
int size = a.ElementSize();
```

mwSize NumberOfElements() const

Description

Determine the total size of the array.

Example

```
mwArray a(2, 2, mxDOUBLE_CLASS);
int n = a.NumberOfElements();
```

mwSize NumberOfNonZeros() const

Description

Determine the size of the of the array's data. If the underlying array is not sparse, this returns the same value as `NumberOfElements()`.

Example

```
mwArray a(2, 2, mxDOUBLE_CLASS);
int n = a.NumberOfNonZeros();
```

mwSize MaximumNonZeros() const

Description

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()`.

Example

```
mwArray a(2, 2, mxDOUBLE_CLASS);
int n = a.MaximumNonZeros();
```

mwSize NumberOfDimensions() const

Description

Determine the dimensionality of the array.

Example

```
mwArray a(2, 2, mxDOUBLE_CLASS);
int n = a.NumberOfDimensions();
```

int NumberOfFields() const

Description

Determine the number of fields in a `struct` array. If the underlying array is not of type `struct`, zero is returned.

Example

```
const char* fields[] = {"a", "b", "c"};
mwArray a(2, 2, 3, fields);
int n = a.NumberOfFields();
```

mwString GetFieldName(int index)

Description

Determine the name of a given field in a `struct` array. If the underlying array is not of type `struct`, an exception is thrown.

Arguments

`int index`	Index of the field to name. Indexing starts at zero.

Example

```
const char* fields[] = {"a", "b", "c"};
mwArray a(2, 2, 3, fields);
mwString tempname = a.GetFieldName(1);
const char* name = (const char*)tempname;
```

mwArray GetDimensions() const

Description

Determine the size of each dimension in the array. The size of the returned array is 1-by-`NumberOfDimensions()`.

Example

```
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray dims = a.GetDimensions();
```

bool IsEmpty() const

Description

Determine if an array is empty.

Example

```
mwArray a;
bool b = a.IsEmpty();
```

bool IsSparse() const

Description

Determine if an array is sparse.

Example

```
mwArray a(2, 2, mxDOUBLE_CLASS);
bool b = a.IsSparse();
```

bool IsNumeric() const

Description

Determine if an array is numeric.

Example

```
mwArray a(2, 2, mxDOUBLE_CLASS);
bool b = a.IsNumeric();
```

bool IsComplex() const

Description

Determine if an array is complex.

Example

```
mwArray a(2, 2, mxDOUBLE_CLASS, mxCOMPLEX);
bool b = a.IsComplex();
```

bool Equals(const mwArray& arr) const

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.

Arguments

`mwArray& arr`	Array to compare to array.

Example

```
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray b(1, 1, mxDOUBLE_CLASS);
a = 1.0;
```

```
b = 1.0;
bool c = a.Equals(b);
```

int CompareTo(const mwArray& arr) const

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.

Arguments

`mwArray& arr`	Array to compare to array.

Example

```
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray b(1, 1, mxDOUBLE_CLASS);
a = 1.0;
b = 1.0;
int n = a.CompareTo(b);
```

int HashCode() const

Description

Constructs a unique hash value form the underlying bytes in the array. Therefore, arrays of different types will have different hash codes, even if they are initialized with the same data.

Example

```
mwArray a(1, 1, mxDOUBLE_CLASS);
int n = a.HashCode();
```

mwString ToString() const

Description

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.

Example

```
mwArray a(1, 1, mxDOUBLE_CLASS, mxCOMPLEX);
```

```
a.Real() = 1.0;
a.Imag() = 2.0;
printf("%s\n", (const char*)(a.ToString()));
```

mwArray RowIndex() const

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.

Example

```
#include <stdio.h>
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray rows = a.RowIndex();
```

mwArray ColumnIndex() const

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.

Example

```
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray rows = a.ColumnIndex();
```

void MakeComplex()

Description

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.

Example

```
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);
```

mwArray Get(mwSize num_indices, ...)

Description

Fetches a single element at a specified index. The index is passed by first passing the number of indices followed by a comma-separated list of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing), in which case the element at the specified 1-based offset is returned, accessing data in column-wise order, or `NumberOfDimensions()` (multiple subscript indexing), in which case, the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the *`i`*th index has the valid range: `1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

Arguments

`mwSize 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.

Example

```
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 = a.Get(2, 1, 2);
x = a.Get(2, 2, 2);
```

mwArray Get(const char* name, mwSize num_indices, ...)

Description

Fetches a single element at a specified field name and index. This method may only be called on an array that is of type `mxSTRUCT_CLASS`. An `mwException` is thrown if the underlying array is not a `struct` array. The field name passed must be a valid field name in the `struct` array. The index is passed by first passing the number of indices followed by a comma-separated list of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing), in which case the element at the specified 1-based offset is returned, accessing data in column-wise order, or `NumberOfDimensions()` (multiple subscript indexing), in which case, the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the ith index has the valid range: `1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

Arguments

`char* name`	Null-terminated character buffer containing the name of the field
`mwSize 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.

Example

```
const char* fields[] = {"a", "b", "c"};

mwArray a(1, 1, 3, fields);
mwArray b = a.Get("a", 1, 1);
mwArray b = a.Get("b", 2, 1, 1);
```

mwArray Get(mwSize num_indices, const mwIndex* index)

Description

Fetches a single element at a specified index. The index is passed by first passing the number of indices, followed by an array of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing), in which case the element at the specified 1-based offset is returned, accessing data in column-wise order, or `NumberOfDimensions()` (multiple subscript indexing), in which case, the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the ith index has the valid range: `1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

Arguments

`mwSize num_indices`	Size of index array
`mwIndex* index`	Array of at least size `num_indices` containing the indices

Example

```
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 = a.Get(2, index);
index[0] = 2;
index[1] = 2;
x = a.Get(2, index);
```

mwArray Get(const char* name, mwSize num_indices, const mwIndex* index)

Description

Fetches 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 thrown if an invalid number of indices is passed in or if any index is out of bounds.

Arguments

`char* name`	Null-terminated character buffer containing the name of the field
`mwSize num_indices`	Number of indices passed in
`mwIndex* index`	Array of at least size `num_indices` containing the indices

Example

```
const char* fields[] = {"a", "b", "c"};
int index[2] = {1, 1};
mwArray a(1, 1, 3, fields);
mwArray b = a.Get("a", 1, index);
mwArray b = a.Get("b", 2, index);
```

mwArray Real()

Description

Accesses the real part of a complex array. The returned `mwArray` is considered real and has the same dimensionality and type as the original.

Complex arrays consist of Complex numbers, which are 1 X 2 vectors (pairs). For example, if the number is `3+5i`, then the pair is `(3,5i)`. An array of Complex numbers is therefore two dimensional `(N X 2)`, where

N is the number of complex numbers in the array. `2+4i, 7-3i, 8+6i` would be represented as `(2,4i) (7,3i) (8,6i)`. Complex numbers have two components, real and imaginary.

The MATLAB function `Real`can be applied to an array of Complex numbers. It extracts the corresponding part of the Complex number. For example, `REAL(3,5i) == 3`.

Example

```
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);
```

mwArray Imag()

Description

Accesses the imaginary part of a complex array. The returned `mwArray` is considered real and has the same dimensionality and type as the original.

Complex arrays consist of Complex numbers, which are 1 X 2 vectors (pairs). For example, if the number is `3+5i`, then the pair is `(3,5i)`. An array of Complex numbers is therefore two dimensional `(N X 2)`, where *N* is the number of complex numbers in the array. `2+4i, 7-3i, 8+6i` would be represented as `(2,4i) (7,3i) (8,6i)`. Complex numbers have two components, real and imaginary.

The MATLAB function `Imag` can be applied to an array of Complex numbers. It extracts the corresponding part of the Complex number. For example, `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.

Example

```
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.Imag().SetData(idata, 4);
```

void Set(const mwArray& arr)

Description

Assign shared copy of input array to currently referenced cell for arrays of type `mxCELL_CLASS` and `mxSTRUCT_CLASS`.

Arguments

`mwArray& arr`	`mwArray` to assign to currently referenced cell

Example

```
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray b(2, 2, mxINT16_CLASS);
mwArray c(1, 2, mxCELL_CLASS);
c.Get(1,1).Set(a);
c.Get(1,2).Set(b);
```

void GetData(<numeric-type>* buffer, mwSize len) const

Description

Copies the array's data into supplied numeric buffer.

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.

Arguments

`<numeric-type>* buffer`	Buffer to receive copy. 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`
`mwSize len`	Maximum length of buffer. A maximum of `len` elements will be copied.

Example

```
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);
```

void GetLogicalData(mxLogical* buffer, mwSize len) const

Description

Copies the array's data into supplied `mxLogical` buffer.

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.

Arguments

`mxLogical* buffer`	Buffer to receive copy
`mwSize len`	Maximum length of buffer. A maximum of `len` elements will be copied.

Example

```
mxLogical data[4] = {true, false, true, false};
mxLogical data_copy[4] ;
mwArray a(2, 2, mxLOGICAL_CLASS);
a.SetLogicalData(data, 4);
a.GetLogicalData(data_copy, 4);
```

void GetCharData(mxChar* buffer, mwSize len) const

Description

Copies the array's data into supplied `mxChar` buffer.

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.

Arguments

`mxChar** buffer`	Buffer to receive copy
`mwSize len`	Maximum length of buffer. A maximum of `len` elements will be copied.

Example

```
mxChar data[6] = {'H', 'e' , `l' , 'l' , 'o' , '\0'};
mxChar data_copy[6] ;
mwArray a(1, 6, mxCHAR_CLASS);
a.SetCharData(data, 6);
a.GetCharData(data_copy, 6);
```

void SetData(<numeric-type>* buffer, mwSize len) const

Description

Copies the data from supplied numeric buffer into the array.

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.

Arguments

`<numeric-type>* buffer`	Buffer containing data to copy. 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`
`mwSize len`	Maximum length of buffer. A maximum of `len` elements will be copied.

Example

```
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);
```

void SetLogicalData(mxLogical* buffer, mwSize len) const

Description

Copies the data from the supplied `mxLogical` buffer into the array.

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.

Arguments

`mxLogical* buffer`	Buffer containing data to copy
`mwSize len`	Maximum length of buffer. A maximum of `len` elements will be copied.

Example

```
mxLogical data[4] = {true, false, true, false};
mxLogical data_copy[4] ;
mwArray a(2, 2, mxLOGICAL_CLASS);
a.SetLogicalData(data, 4);
a.GetLogicalData(data_copy, 4);
```

void SetCharData(mxChar* buffer, mwSize len) const

Description

Copies the data from the supplied `mxChar` buffer into the array.

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.

Arguments

`mxChar** buffer`	Buffer containing data to copy
`mwSize len`	Maximum length of buffer. A maximum of `len` elements will be copied.

Example

```
mxChar data[6] = {'H', 'e' , `l' , 'l' , 'o' , '\0'};
mxChar data_copy[6] ;
mwArray a(1, 6, mxCHAR_CLASS);
a.SetCharData(data, 6);
a.GetCharData(data_copy, 6);
```

static mwArray Deserialize(const mwArray& arr)

Description

Deserializes an array that has been serialized with `mwArray::Serialize()`. The input array must be of type `mxUINT8_CLASS` and contain the data from a serialized array. If the input data does not represent a serialized `mwArray`, the behavior of this method is undefined.

Arguments

`mwArray& arr`	`mwArray` that has been obtained by calling `mwArray::Serialize`

Example

```
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
mwArray a(1,4,mxDOUBLE_CLASS);
a.SetData(rdata, 4);
mwArray b = a.Serialize();
a = mwArray::Deserialize(b);
```

static mwArray NewSparse(mwSize rowindex_size, const mwIndex* rowindex, mwSize colindex_size, const mwIndex* colindex, mwSize data_size, const mxDouble* rData, mwSize num_rows, mwSize num_cols, mwSize nzmax)

Description

Creates real sparse matrix of type `double` with specified number of rows and columns.

The lengths of input row, column index, and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated throughout the construction of the matrix.

If any element of the `rowindex` or `colindex` array is greater than the specified values in `num_rows` or `num_cols` respectively, an exception is thrown.

Arguments

`mwSize rowindex_size`	Size of `rowindex` array
`mwIndex* rowindex`	Array of row indices of non-zero elements
`mwSize colindex_size`	Size of `colindex` array
`mwIndex* colindex`	Array of column indices of non-zero elements
`mwSize data_size`	Size of data array
`mxDouble* rData`	Data associated with non-zero row and column indices
`mwSize num_rows`	Number of rows in matrix
`mwSize num_cols`	Number of columns in matrix
`mwSize nzmax`	Reserved storage for sparse matrix. If `nzmax` is zero, storage will be set to `max{rowindex_size, colindex_size, data_size}`.

Example

This example constructs a sparse 4 X 4 tridiagonal matrix:

```
2 -1  0  0
-1  2 -1  0
0 -1  2 -1
0  0 -1  2
```

The following code, when run:

```
double rdata[]        =
           {2.0, -1.0, -1.0, 2.0, -1.0,
            -1.0, 2.0, -1.0, -1.0, 2.0};
mwIndex row_tridiag[] =
           {1,    2,    1,   2,    3,
            2,   3,    4,    3,   4  };
```

```
mwIndex col_tridiag[] =
           {1,    1,    2,   2,    2,
              3,   3,    3,    4,   4  };

mwArray mysparse =
           mwArray::NewSparse(10, row_tridiag,
                              10, col_tridiag,
                              10, rdata, 4, 4, 10);
std::cout << mysparse << std::endl;
```

will display the following output to the screen:

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

static mwArray NewSparse(mwSize rowindex_size, const mwIndex* rowindex, mwSize colindex_size, const mwIndex* colindex, mwSize data_size, const mxDouble* rdata, mwSize nzmax)

Description

Creates real sparse matrix of type `double` with number of rows and columns inferred from input data.

The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated through out the construction of the matrix.

The number of rows and columns in the created matrix are calculated form the input `rowindex` and `colindex` arrays as `num_rows = max{rowindex}`, `num_cols = max{colindex}`.

Arguments

`mwSize rowindex_size`	Size of `rowindex` array
`mwIndex* rowindex`	Array of row indices of non-zero elements
`mwSize colindex_size`	Size of `colindex` array
`mwIndex* colindex`	Array of column indices of non-zero elements
`mwSize data_size`	Size of data array
`mxDouble* rData`	Data associated with non-zero row and column indices
`mwSize nzmax`	Reserved storage for sparse matrix. If `nzmax` is zero, storage will be set to `max{rowindex_size, colindex_size, data_size}`.

Example

In this example, we construct a sparse 4 X 4 identity matrix. The value of 1.0 is copied to each non-zero element defined by row and column index arrays:

```
double one = 1.0;
mwIndex row_diag[] = {1, 2, 3, 4};
mwIndex col_diag[] = {1, 2, 3, 4};

mwArray mysparse =
  mwArray::NewSparse(4, row_diag,
                     4, col_diag,
                     1, &one,
                     0);
std::cout << mysparse << std::endl;
```

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

static mwArray NewSparse(mwSize rowindex_size, const mwIndex* rowindex, mwSize colindex_size, const mwIndex* colindex, mwSize data_size, const mxDouble* rdata, const mxDouble* idata, mwSize num_rows, mwSize num_cols, mwSize nzmax)

Description

Creates complex sparse matrix of type `double` with specified number of rows and columns.

The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated through out the construction of the matrix.

If any element of the `rowindex` or `colindex` array is greater than the specified values in `num_rows`, `num_cols`, respectively, then an exception is thrown.

Arguments

`mwSize rowindex_size`	Size of `rowindex` array
`mwIndex* rowindex`	Array of row indices of non-zero elements
`mwSize colindex_size`	Size of `colindex` array
`mwIndex* colindex`	Array of column indices of non-zero elements
`mwSize data_size`	Size of data array
`mxDouble* rData`	Real part of data associated with non-zero row and column indices

`mxDouble* iData`	Imaginary part of data associated with non-zero row and column indices
`mwSize num_rows`	Number of rows in matrix
`mwSize num_cols`	Number of columns in matrix
`mwSize nzmax`	Reserved storage for sparse matrix. If `nzmax` is zero, storage will be set to `max{rowindex_size, colindex_size, data_size}`.

Example

This example constructs a complex tridiagonal matrix:

```
double rdata[] =
  {2.0, -1.0, -1.0, 2.0, -1.0, -1.0, 2.0, -1.0, -1.0, 2.0};
double idata[] =
  {20.0, -10.0, -10.0, 20.0, -10.0, -10.0, 20.0, -10.0,
                                                -10.0, 20.0};
mwIndex row_tridiag[] =
  {1,    2,    1,   2,    3,    2,   3,    4,    3,   4};
mwIndex col_tridiag[] =
  {1,    1,    2,   2,    2,    3,   3,    3,    4,   4};

mwArray mysparse = mwArray::NewSparse(10, row_tridiag,
                                      10, col_tridiag,
                                      10, rdata,
                                      idata, 4, 4, 10);
std::cout << mysparse << std::endl;
```

It displays the following output to the screen:

```
(1,1)       2.0000 +20.0000i
(2,1)      -1.0000 -10.0000i
(1,2)      -1.0000 -10.0000i
```

```
(2,2)      2.0000 +20.0000i
(3,2)     -1.0000 -10.0000i
(2,3)     -1.0000 -10.0000i
(3,3)      2.0000 +20.0000i
(4,3)     -1.0000 -10.0000i
(3,4)     -1.0000 -10.0000i
(4,4)      2.0000 +20.0000i
```

static mwArray NewSparse(mwSize rowindex_size, const mwIndex* rowindex, mwSize colindex_size, const mwIndex* colindex, mwSize data_size, const mxDouble* rdata, const mxDouble* idata, mwSize nzmax)

Description

Creates complex sparse matrix of type `double` with number of rows and columns inferred from input data.

The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated through out the construction of the matrix.

The number of rows and columns in the created matrix are calculated form the input `rowindex` and `colindex` arrays as `num_rows = max{rowindex}, num_cols = max{colindex}`.

Arguments

`mwSize rowindex_size`	Size of `rowindex` array
`mwIndex* rowindex`	Array of row indices of non-zero elements
`mwSize colindex_size`	Size of `colindex` array
`mwIndex* colindex`	Array of column indices of non-zero elements
`mwSize data_size`	Size of data array
`mxDouble* rData`	Real part of data associated with non-zero row and column indices

`mxDouble* iData`	Imaginary part of data associated with non-zero row and column indices
`mwSize nzmax`	Reserved storage for sparse matrix. If `nzmax` is zero, storage will be set to `max{rowindex_size, colindex_size, data_size}`.

Example

This example constructs a complex matrix by inferring dimensions and storage allocation from the input data.

```
mwArray mysparse =
   mwArray::NewSparse(10, row_tridiag,
                      10, col_tridiag,
                      10, rdata, idata,
                      0);
std::cout << mysparse << std::endl;

(1,1)      2.0000 +20.0000i
(2,1)     -1.0000 -10.0000i
(1,2)     -1.0000 -10.0000i
(2,2)      2.0000 +20.0000i
(3,2)     -1.0000 -10.0000i
(2,3)     -1.0000 -10.0000i
(3,3)      2.0000 +20.0000i
(4,3)     -1.0000 -10.0000i
(3,4)     -1.0000 -10.0000i
(4,4)      2.0000 +20.0000i
```

static mwArray NewSparse(mwSize rowindex_size, const mwIndex* rowindex, mwSize colindex_size, const mwIndex* colindex, mwSize data_size, const mxLogical* rdata, mwSize num_rows, mwSize num_cols, mwSize nzmax)

Description

Creates logical sparse matrix with specified number of rows and columns.

The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated throughout the construction of the matrix.

If any element of the `rowindex` or `colindex` array is greater than the specified values in `num_rows`, `num_cols`, respectively, then an exception is thrown.

Arguments

`mwSize rowindex_size`	Size of `rowindex` array
`mwIndex* rowindex`	Array of row indices of non-zero elements
`mwSize colindex_size`	Size of `colindex` array
`mwIndex* colindex`	Array of column indices of non-zero elements
`mwSize data_size`	Size of data array
`mxLogical* rData`	Data associated with non-zero row and column indices
`mwSize num_rows`	Number of rows in matrix
`mwSize num_cols`	Number of columns in matrix
`mwSize nzmax`	Reserved storage for sparse matrix. If `nzmax` is zero, storage will be set to `max{rowindex_size, colindex_size, data_size}`.

Example

This example creates a sparse logical 4 X 4 tridiagonal matrix, assigning `true` to each non-zero value:

```
mxLogical one = true;
mwIndex row_tridiag[] =
      {1,    2,    1,   2,    3,
       2,   3,    4,    3,   4};
mwIndex col_tridiag[] =
      {1,    1,    2,   2,    2,
       3,   3,    3,    4,   4};

mwArray mysparse =
      mwArray::NewSparse(10, row_tridiag,
                         10, col_tridiag,
                          1, &one,
                          4, 4, 10);
std::cout << mysparse << std::endl;
```

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

static mwArray NewSparse(mwSize rowindex_size, const mwIndex* rowindex, mwSize colindex_size, const mwIndex* colindex, mwSize data_size, const mxLogical* rdata, mwSize nzmax)

Description

Creates logical sparse matrix with number of rows and columns inferred from input data.

The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated through out the construction of the matrix.

The number of rows and columns in the created matrix are calculated form the input `rowindex` and `colindex` arrays as `num_rows = max {rowindex}, num_cols = max {colindex}`.

Arguments

`mwSize rowindex_size`	Size of `rowindex` array
`mwIndex* rowindex`	Array of row indices of non-zero elements
`mwSize colindex_size`	Size of `colindex` array
`mwIndex* colindex`	Array of column indices of non-zero elements
`mwSize data_size`	Size of data array
`mxLogical* rData`	Data associated with non-zero row and column indices
`mwSize nzmax`	Reserved storage for sparse matrix. If `nzmax` is zero, storage will be set to `max{rowindex_size, colindex_size, data_size}`.

Example

This example uses the data from the first example, but allows the number of rows, number of columns, and allocated storage to be calculated from the input data:

```
mwArray mysparse =
    mwArray::NewSparse(10, row_tridiag,
                       10, col_tridiag,
                       1, &one,
                       0);
std::cout << mysparse << std::endl;

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

static mwArray NewSparse (mwSize num_rows, mwSize num_cols, mwSize nzmax, mxClassID mxID, mxComplexity cmplx = mxREAL)

Description

Creates an empty sparse matrix. All elements in an empty sparse matrix are initially zero, and the amount of allocated storage for non-zero elements is specified by `nzmax`.

Arguments

`mwSize num_rows`	Number of rows in matrix
`mwSize num_cols`	Number of columns in matrix
`mwSize nzmax`	Reserved storage for sparse matrix

`mxClassID mxID`	Type of data to store in matrix. Currently, sparse matrices of type `double` precision and `logical` are supported. Pass `mxDOUBLE_CLASS` to create a `double` precision sparse matrix. Pass `mxLOGICAL_CLASS` to create a `logical` sparse matrix.
`mxComplexity cmplx`	Complexity of matrix. Pass `mxCOMPLEX` to create a `complex` sparse matrix and `mxREAL` to create a `real` sparse matrix. This argument may be omitted, in which case the default complexity is `real`

Example

This example constructs a real 3 X 3 empty sparse matrix of type `double` with reserved storage for 4 non-zero elements:

```
mwArray mysparse = mwArray::NewSparse
                (3, 3, 4, mxDOUBLE_CLASS);
std::cout << mysparse << std::endl;

All zero sparse: 3-by-3
```

static double GetNaN()

Description

Get value of `NaN` (Not-a-Number).

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.

Example

```
double x = mwArray::GetNaN();
```

static double GetEps()

Description

Returns the value of the MATLAB `eps` variable. This variable is the distance from 1.0 to the next largest floating-point number. Consequently, it is a measure of floating-point accuracy. The MATLAB `pinv` and `rank` functions use `eps` as a default tolerance.

Example

```
double x = mwArray::GetEps();
```

static double GetInf()

Description

Returns the value of the MATLAB internal `Inf` variable. `Inf` is a permanent variable representing IEEE arithmetic positive infinity. The value of `Inf` is built into the system; you cannot modify it.

Operations that return `Inf` include

- Division by 0. For example, 5/0 returns `Inf`.
- Operations resulting in overflow. For example, `exp(10000)` returns `Inf` because the result is too large to be represented on your machine.

Example

```
double x = mwArray::GetInf();
```

static bool IsFinite(double x)

Description

Determine whether or not a value is finite. A number is finite if it is greater than `-Inf` and less than `Inf`.

Arguments

`doulbe x`	Value to test for finiteness

Example

```
bool x = mwArray::IsFinite(1.0);
```

static bool IsInf(double x)

Description

Determines 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.

Arguments

`doulbe x`	Value to test for infiniteness

Example

```
bool x = mwArray::IsInf(1.0);
```

static bool IsNaN(double x)

Description

Determines whether or not the value is `NaN`. `NaN` is the IEEE arithmetic representation for Not-a-Number. `NaN` is obtained as a result of mathematically undefined operations such as

- `0.0/0.0`
- `Inf-Inf`

The system understands a family of bit patterns as representing `NaN`. In other words, `NaN` is not a single value, rather it is a family of numbers that the MATLAB software (and other IEEE-compliant applications) use to represent an error condition or missing data.

Arguments

`doulbe x`	Value to test for `NaN`

Example

```
bool x = mwArray::IsNaN(1.0);
```

Operators

mwArray operator()(mwIndex i1, mwIndex i2, mwIndex i3, ...,)

Description

Fetches a single element at a specified index. The index is passed as a comma-separated list of 1-based indices. This operator is overloaded to support 1 through 32 indices. The valid number of indices that can be passed in is either 1 (single subscript indexing), in which case the element at the specified 1-based offset is returned, accessing data in column-wise order, or `NumberOfDimensions()` (multiple subscript indexing), in which case, the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the ith index has the valid range: `1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

Arguments

`mwIndex i1, mwIndex i2,` `mwIndex i3, ...,`	Comma-separated list of input indices

Example

```
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 = a(1,2);
x = a(2,2);
```

mwArray operator()(const char* name, mwIndex i1, mwIndex i2, mwIndex i3, ...,)

Description

Fetches a single element at a specified field name and index. This method may only be called on an array that is of type `mxSTRUCT_CLASS`. An `mwException` is thrown if the underlying array is not a `struct` array. The field name passed must be a valid field name in the `struct` array. The index is passed by first passing the number of indices, followed by an array of 1-based indices. This operator is overloaded to support 1 through 32 indices. The valid number of indices that can be passed in is either 1 (single subscript indexing), in which case the element at the specified 1-based offset is returned, accessing data in column-wise order, or `NumberOfDimensions()` (multiple subscript indexing), in which case, the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the ith index has the valid range: `1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

Arguments

`char* name`	Null terminated string containing the field name to get
`mwIndex i1, mwIndex i2,` `mwIndex i3, ...,`	Comma-separated list of input indices

Example

```
const char* fields[] = {"a", "b", "c"};
int index[2] = {1, 1};
mwArray a(1, 1, 3, fields);
mwArray b = a("a", 1, 1);
mwArray b = a("b", 1, 1);
```

mwArray& operator=(const <type>& x)

Description

Sets a single scalar value. This operator is overloaded for all numeric and logical types.

Arguments

`const <type>& x`	Value to assign

Example

```
mwArray a(2, 2, mxDOUBLE_CLASS);
a(1,1) = 1.0;
a(1,2) = 2.0;
a(2,1) = 3.0;
a(2,2) = 4.0;
```

operator <type>() const

Description

Fetches a single scalar value. This operator is overloaded for all numeric and logical types.

Example

```
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 = (double)a(1,2);
x = (double)a(2,1);
x = (double)a(2,2);
```