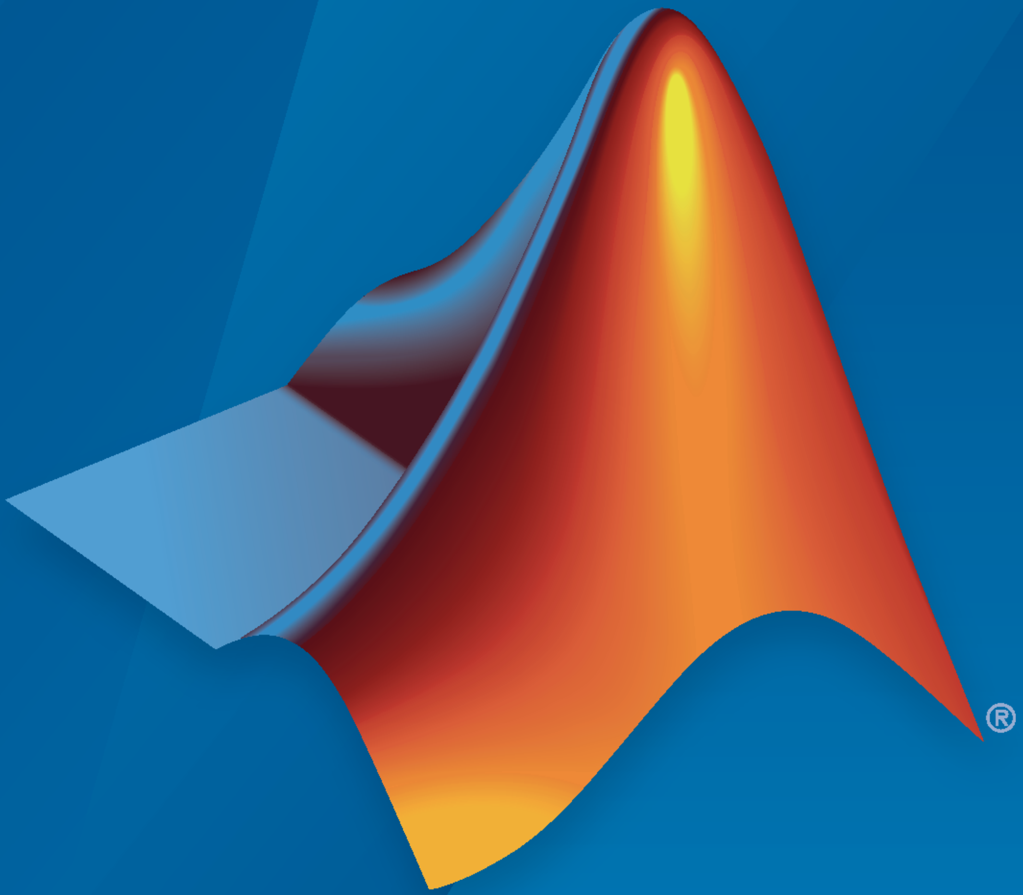
MATLAB® Compiler™

User's Guide



MATLAB®

R2025b

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

MATLAB® Compiler™ User's Guide

© COPYRIGHT 1995–2025 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 2014a)
October 2014	Online only	Revised for Version 5.2 (Release 2014b)
March 2015	Online only	Revised for Version 6.0 (Release 2015a)
September 2015	Online only	Revised for Version 6.1 (Release 2015b)
October 2015	Online only	Rereleased for Version 6.0.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 6.2 (Release 2016a)
September 2016	Online Only	Revised for Version 6.3 (Release 2016b)
March 2017	Online only	Revised for Version 6.4 (Release R2017a)
September 2017	Online only	Revised for Version 6.5 (Release R2017b)
March 2018	Online only	Revised for Version 6.6 (Release R2018a)
September 2018	Online only	Revised for Version 7.0 (Release R2018b)
March 2019	Online only	Revised for Version 7.0.1 (Release R2019a)
September 2019	Online only	Revised for Version 7.1 (Release R2019b)
March 2020	Online only	Revised for Version 8.0 (Release R2020a)
September 2020	Online only	Revised for Version 8.1 (Release R2020b)
March 2021	Online only	Revised for Version 8.2 (Release R2021a)
September 2021	Online only	Revised for Version 8.3 (Release R2021b)
March 2022	Online only	Revised for Version 8.4 (Release R2022a)
September 2022	Online only	Revised for Version 8.5 (Release R2022b)
March 2023	Online only	Revised for Version 8.6 (Release R2023a)
September 2023	Online only	Revised for Version 23.2 (R2023b)
March 2024	Online only	Revised for Version 24.1 (R2024a)
September 2024	Online only	Revised for Version 24.2 (R2024b)
March 2025	Online only	Revised for Version 25.1 (R2025a)
September 2025	Online only	Rereleased for Version 25.2 (R2025b)

Contents

1 Getting Started

MATLAB Compiler Product Description **1-2**

Steps for Deployment with MATLAB Compiler **1-3**
Write Deployable MATLAB Code **1-3**
Package Code for Target **1-3**
Test Artifacts Before Deployment **1-4**
Distribute Files to Target Platform **1-4**
Integrate Artifact with Target Language Application **1-4**
Install MATLAB Runtime **1-4**
Limitations and Restrictions **1-5**

Appropriate Tasks for MATLAB Compiler Products **1-6**

Choose Deployment Option **1-8**
Get Started with Compiler Apps **1-8**
Compile Using compiler.build Command Line Functions **1-9**
Specify Additional Packaging Options Using mcc **1-9**
Limitations **1-10**

Target-Specific Compiler Apps for MATLAB Code Deployment **1-12**
Overview of Target-Specific Compiler Apps **1-12**
MATLAB Projects Integration **1-12**
Compiler Tasks **1-12**
Upgrade Projects **1-13**
Transition of Deployment Apps **1-13**

2 MATLAB Runtime Additional Info

MATLAB Runtime Container **2-2**
Contents **2-2**
Requirements **2-2**
Download Full MATLAB Runtime Image **2-2**
Download MATLAB Runtime Without GPU Libraries **2-3**

3 Deploying Standalone Applications

Standalone Applications and Arguments **3-2**
Pass Command Line Arguments to Applications **3-2**
Handle Input Arguments in MATLAB **3-3**
Display Data to Screen Using MATLAB File **3-4**

Use Parallel Computing Toolbox in Deployed Applications **3-6**
Export Cluster Profile **3-6**
Link to Parallel Computing Toolbox Profile Within Your Code **3-6**
Pass Parallel Computing Toolbox Profile at Run Time **3-7**
Switch Between Cluster Profiles in Deployed Applications **3-7**
Sample C Code to Load Cluster Profile **3-7**

Integrate Application with Mac OS X Finder **3-9**
Overview **3-9**
Installing the Mac Application Launcher Preference Pane **3-9**
Configuring the Installation Area **3-9**
Running the Application **3-11**

Protect Code and Data in Deployable Archive **3-12**
Protect User Data and Obfuscate File Structure **3-12**
Package Code with Sensitive Information Using Secrets **3-12**
Require Decryption Key at Run Time **3-13**

Files Generated After Packaging MATLAB Functions **3-14**
Files Generated by MATLAB Compiler **3-14**

Distribute MATLAB Compiler SDK Files to Application Developers **3-17**
Distribute COM Components **3-17**
Distribute C/C++ Shared Libraries **3-17**
Distribute Java Packages **3-18**
Distribute .NET Assemblies **3-18**
Distribute Python Packages **3-18**

4 Customizing a Compiler Project

Manage Support Packages **4-2**
Using Compiler App **4-2**
Using compiler.build **4-2**
Using mcc **4-3**

5 MATLAB Code Deployment

Dependency Analysis Using MATLAB Compiler **5-2**
Function Dependency **5-2**
License and Toolbox Dependencies **5-3**
Data File Dependency **5-3**
Troubleshoot Missing Dependencies **5-3**

About Deployable Archives **5-5**
Additional Details **5-6**

MATLAB Runtime Component Cache and Deployable Archive Embedding **5-8**
Cache Size and Location **5-8**
Change Options Using Environment Variables **5-8**
Override Default Behavior **5-9**

Write Deployable MATLAB Code **5-10**
Accepted File Types for Packaging **5-10**
Packaged Applications Do Not Process MATLAB Files at Run Time **5-10**
Get Proper Licenses for Toolbox Functionality You Want to Deploy **5-11**
Use isdeployed Functions to Execute Deployment-Specific Code Paths . . **5-11**
Do Not Rely on Changing Directory or Path to Control the Execution of MATLAB Files **5-11**
Gradually Refactor Applications That Depend on Non-Deployable Functions **5-12**
Do Not Create or Use Nonconstant Static State Variables **5-12**

Calling Shared Libraries in Deployed Applications **5-14**

Include and Access Files in Packaged Applications **5-15**
Include Files in Deployable Archive **5-15**
Access Files from Deployed Functions **5-16**
Example Processing MATLAB Data for Deployed Applications **5-16**

6 Standalone Application Creation

Dependency Analysis Function and User Interaction with the Compilation Path **6-2**
addpath and rmpath in MATLAB **6-2**
Passing -I <directory> on the Command Line **6-2**
Passing -N and -p <directory> on the Command Line **6-2**

7 Deployment Process

About MATLAB Runtime **7-2**
Differences Between MATLAB Runtime and MATLAB **7-2**
Size and Performance Considerations for MATLAB Runtime **7-2**

Download and Install MATLAB Runtime **7-4**
Download MATLAB Runtime Installer **7-4**
Install MATLAB Runtime Interactively **7-4**
Default Install Folder **7-6**
Install MATLAB Runtime Noninteractively **7-6**
Install MATLAB Runtime Without Administrator Rights **7-8**
Install Multiple MATLAB Runtime Versions on Single Machine **7-8**
MATLAB and MATLAB Runtime on Same Machine **7-8**
Uninstall MATLAB Runtime **7-9**

Deploy Applications and MATLAB Runtime on Network Drives **7-10**
Install MATLAB Runtime on Network Drives **7-10**
Deploy Applications on Network Drives **7-10**

MATLAB Runtime on Big Data Platforms **7-12**
CLOUDERA **7-12**
Apache Ambari **7-12**
Azure HDInsight **7-12**

Install Deployed Application **7-14**
Install Application Interactively **7-14**
Install Application Noninteractively **7-15**

8 Work with the MATLAB Runtime

MATLAB Runtime Startup Options **8-2**
Set MATLAB Runtime Options **8-2**

Using MATLAB Runtime User Data Interface **8-3**
MATLAB Functions **8-3**
Set and Retrieve MATLAB Runtime Data for Shared Libraries **8-3**

Display MATLAB Runtime Initialization Messages **8-5**
Best Practices **8-5**

Ensure MATLAB Runtime Compatibility for MATLAB Compiler and MATLAB Compiler SDK Artifacts **8-6**

9 Distributing Code to an End User

Distribute MATLAB Code Using the MATLAB Runtime **9-2**
MATLAB Runtime **9-2**

10 Compiler Commands

Compiler Tips **10-2**
Deploying Applications That Call the Java Native Libraries **10-2**
Using the VER Function in a Compiled MATLAB Application **10-2**

11 Standalone Applications

Deploying Standalone Applications **11-2**
Compiling the Application **11-2**
Testing the Application **11-2**
Deploying the Application **11-3**
Running the Application **11-4**

12 Troubleshooting

Testing Failures **12-2**
Are you able to execute the application from MATLAB? **12-2**
Does the application begin execution and result in MATLAB or other errors?
. . . . **12-2**
Do you have multiple MATLAB versions installed? **12-2**
If you are testing a standalone executable or shared library and driver application, did you install MATLAB Runtime? **12-3**
Do you receive an error message about a missing DLL? **12-3**

Investigate Deployed Application Failures **12-4**
Install MATLAB Runtime **12-4**
Update Dynamic Library Path on Linux or macOS **12-4**
Error Message for Missing DLL **12-4**
Obtain Write Access to Install Directory **12-4**
Deploy Newer Version of Application **12-5**

13 Limitations and Restrictions

Limitations **13-2**
Packaging MATLAB and Toolboxes **13-2**
Callback Problems Due to Missing Functions **13-2**
Finding Missing Functions in MATLAB File **13-3**
Suppressing Warnings on UNIX System **13-4**
Cannot Use Graphics with -nojvm Option **13-4**
Cannot Create Output File **13-4**
No MATLAB File Help for Packaged Functions **13-4**
No MATLAB Runtime Versioning on Mac OS X **13-4**
Older Neural Networks Not Deployable with MATLAB Compiler **13-5**
Restrictions on Calling PRINTDLG with Multiple Arguments in Packaged Mode **13-5**
Opening File Using which and open Does Not Search Current Working Folder **13-5**
Restrictions on Using C++ SetData to Dynamically Resize an mwArray **13-6**
Accepted File Types for Packaging **13-6**

Ensure Multiplatform Portability for Compiled Applications **13-8**
MEX Files **13-8**
MATLAB Toolboxes **13-9**
Java JAR Files **13-9**
Web Apps **13-9**

Functions Not Supported for Compilation by MATLAB Compiler and MATLAB Compiler SDK **13-10**

14 Package to Docker

Package MATLAB Standalone Applications into Docker Images **14-2**
Prerequisites **14-2**
Create Function in MATLAB **14-2**
Create Standalone Application **14-3**
Package Standalone Application into Docker Image **14-3**
Test Docker Image **14-4**
Share Docker Image **14-5**

15 Reference Information

Set MATLAB Runtime Path for Deployment **15-2**
Library Path Environment Variables and MATLAB Runtime Folders **15-2**
Windows **15-3**
Linux **15-3**

macOS 15-4
Set Path Permanently on UNIX 15-4

MATLAB Compiler Licensing **15-6**
Using MATLAB Compiler Licenses for Development 15-6

Deployment Product Terms **15-7**

16 Functions

A MATLAB Compiler Quick Reference

mcc Command Arguments Listed Alphabetically **A-2**
Packaging Log and Output Folders A-4

mcc Command Line Arguments Grouped by Task **A-5**

17 Apps

18 Examples

Create Standalone Application from MATLAB **18-2**
Create Function in MATLAB 18-2
Create Standalone Application Using compiler.build.standaloneApplication 18-2
Run Standalone Application 18-3
Tips 18-4

Create Standalone Application Using Standalone Application Compiler App **18-5**
Create MATLAB Function 18-5
Create Project and Compiler Task 18-5
Specify Build Options 18-7
View Code and Package Standalone Application 18-9
Test and Deploy Application 18-9

Deploy Parallel-Enabled MATLAB Function as Standalone Application **18-11**

Access Sensitive Information in Standalone Application **18-14**
Store SFTP Credentials in Local MATLAB Vault **18-14**
Specify Secrets in Secret Manifest JSON File **18-15**
Write MATLAB Code to Deploy **18-15**
Create Standalone Application Using mcc **18-15**
Run Application **18-15**

Handle Sensitive Information in Deployed Applications **18-17**
Package Code with Secrets **18-17**
Access Secrets on MATLAB Web App Server **18-18**

Getting Started

- “MATLAB Compiler Product Description” on page 1-2
- “Steps for Deployment with MATLAB Compiler” on page 1-3
- “Appropriate Tasks for MATLAB Compiler Products” on page 1-6
- “Choose Deployment Option” on page 1-8
- “Target-Specific Compiler Apps for MATLAB Code Deployment” on page 1-12

MATLAB Compiler Product Description

Build standalone executables and web apps from MATLAB programs

MATLAB Compiler enables you to share MATLAB programs as standalone applications and web apps. With MATLAB Compiler you can also package and deploy MATLAB programs as MapReduce and Spark™ big data applications and as Microsoft® Excel® Add-ins. End users can run your applications royalty-free using MATLAB Runtime.

To provide browser-based access to your MATLAB web apps, you can host them using the development version of MATLAB Web App Server™ included with MATLAB Compiler. MATLAB programs can be packaged into software components for integration with other programming languages (with MATLAB Compiler SDK™). Large-scale deployment to enterprise systems is supported through MATLAB Production Server™.

To generate C and C++ source code from MATLAB, use MATLAB Coder™.

Steps for Deployment with MATLAB Compiler

You can use MATLAB Compiler and MATLAB Compiler SDK to package MATLAB files into deployable components that do not require MATLAB to run.

With MATLAB Compiler, you can create standalone applications, web apps, Microsoft Excel Add-ins, and MapReduce or Spark big data applications.

With MATLAB Compiler SDK, you can create C/C++ shared libraries, .NET assemblies, Java® classes, Python® packages, COM components, MATLAB Production Server deployable archives, Excel add-ins for MATLAB Production Server, and Docker® container-based microservices.

Write Deployable MATLAB Code

To package MATLAB scripts, functions, or class files to run outside of the MATLAB environment, you must first ensure the code is in a finished state and ready to be run by the end user. You can use functions such as `isdeployed` to separate code that must run before deployment. For example, in a deployed application, the MATLAB paths are fixed and cannot change, so use `isdeployed` to separate out code that modifies the path. For more information, see “Write Deployable MATLAB Code” on page 5-10.

In addition to MATLAB scripts, MEX files, and class files, you can include other types of files, such as data files, in the compiled artifact. For details, see “Include and Access Files in Packaged Applications” on page 5-15.

Package Code for Target

During the packaging process, the compiler performs the following steps:

1. Uses a dependency analysis function to include necessary supporting files. For more details, see “Dependency Analysis Using MATLAB Compiler” on page 5-2.
2. Validates MEX-files. In particular, `mexFunction` entry points are verified. For more details about including MEX-files in packaged applications, see “Include and Access Files in Packaged Applications” on page 5-15.
3. Creates a deployable archive from the input files and their dependencies. For more details, see “About Deployable Archives” on page 5-5.
4. Generates target-specific wrapper code.
5. Generates a target-specific binary package.

   For library targets such as C++ shared libraries, Java packages, or .NET assemblies, the compiler invokes the corresponding third-party compiler.

You can use the following options to package MATLAB code.

- The `compiler.build` and `compiler.package` functions, which allow you to package MATLAB code at the command line
- The compiler apps, which allow you to package MATLAB code using a graphical interface
- The `mcc` function, which allows you to package MATLAB code at the command line and offers additional options to control the packaging process

For more information on these packaging methods, see “Choose Deployment Option” on page 1-8.

Test Artifacts Before Deployment

You can test artifacts generated by MATLAB Compiler and MATLAB Compiler SDK using a MATLAB installation instead of MATLAB Runtime. Typically, testing is performed with the same MATLAB installation used to generate the artifact. The version of MATLAB used for testing must match the version used to create the artifact.

Once testing is successful, you can transfer the artifact to a production environment and run it using MATLAB Runtime.

Distribute Files to Target Platform

Once you create a component, distribute either the application files or an installer that contains all of the resources required to run the application on the target platform. You can create an installer using a compiler app or using the `compiler.package.installer` function.

The installer created by `compiler.package.installer` installs your MATLAB Compiler generated artifacts, any generated sample code, and optionally the full MATLAB Runtime. To create a minimal MATLAB Runtime installer that can run one or more specific MATLAB Compiler applications, see `compiler.runtime.customInstaller`.

If you create an installer containing the compiled artifacts and MATLAB Runtime, then MATLAB Runtime is installed along with the application or shared library. If you distribute files manually, ensure MATLAB Runtime is accessible from the target machine. For more information on distributing files, see “Distribute MATLAB Compiler SDK Files to Application Developers” on page 3-17.

Integrate Artifact with Target Language Application

If you use MATLAB Compiler SDK to package MATLAB functions for use in applications written in other programming languages, you must write application code in the target language that uses the packaged MATLAB functionality. For more details, see “Integrate Artifact with Target Language Application” (MATLAB Compiler SDK).

After integrating the artifact with your target language application and testing the application code against MATLAB Runtime, deploy the application along with all required files and dependencies to the end user machine.

Install MATLAB Runtime

MATLAB Runtime is a freely available set of shared libraries that enables you to run packaged MATLAB code without needing a licensed version of MATLAB. For details, see “About MATLAB Runtime” on page 7-2.

If you use an installer containing the compiled artifacts and MATLAB Runtime, then MATLAB Runtime is installed along with the application or shared library. If you distribute files manually, ensure MATLAB Runtime is accessible from the target machine at the same version or newer than the version of MATLAB used to compile the component.

You can run your deployed component on your development machine in MATLAB without needing MATLAB Runtime.

Limitations and Restrictions

You can package most MATLAB functionality that can be called directly from the command line. For a list of functions not supported by MATLAB Compiler, see "Functions Not Supported for Compilation by MATLAB Compiler and MATLAB Compiler SDK" on page 13-10.

Compiled applications can run only on the same platform on which they were developed, with some exceptions. For more details, see "Limitations" (MATLAB Compiler SDK).

See Also

`compiler.build.Results` | `compiler.package.installer` | `mcc`

Related Examples

- "Write Deployable MATLAB Code" on page 5-10
- "Include and Access Files in Packaged Applications" on page 5-15
- "About Deployable Archives" on page 5-5

Appropriate Tasks for MATLAB Compiler Products

MATLAB Compiler and MATLAB Compiler SDK let you run your MATLAB application outside the MATLAB environment. However, they are not appropriate for all external tasks you may want to perform. Some tasks require other products or MATLAB external interfaces. Use the following table to determine which product is appropriate to your needs.

Task	MATLAB Compiler	MATLAB Compiler SDK	MATLAB Coder	Simulink®	HDL Coder™	MATLAB External Interfaces
Package MATLAB applications for deployment to users who do not have MATLAB	■	■				
Package MATLAB programs as standalone applications, web apps, Microsoft Excel Add-ins, Docker images, and MapReduce or Spark big data applications	■					
Package MATLAB programs as C/C++ shared libraries, .NET assemblies, Java classes, Python packages, COM components, and Docker container-based microservices		■				

Task	MATLAB Compiler	MATLAB Compiler SDK	MATLAB Coder	Simulink®	HDL Coder™	MATLAB External Interfaces
Generate readable and portable C/C++ code from MATLAB code			■			
Generate MEX functions from MATLAB code for code verification and acceleration			■			
Integrate MATLAB code into Simulink				■		
Generate hardware description language (HDL) from MATLAB code					■	
Integrate custom C code into MATLAB with MEX files						■
Call MATLAB from C and Fortran programs						■
Task	MATLAB Compiler	MATLAB Compiler SDK	MATLAB Coder	Simulink	HDL Coder	MATLAB External Interfaces

Note Components generated by MATLAB Compiler and MATLAB Compiler SDK cannot be used in the MATLAB environment.

Choose Deployment Option

MATLAB Compiler provides several options for packaging MATLAB code into software components for standalone deployment or integration with other programming languages, depending on the desired build target. You can package code at the command line or using a compiler app associated with the build target.

- The compiler apps allow you to package MATLAB code using a graphical interface.
- The `compiler.build` and `compiler.package` functions allow you to package MATLAB code at the command line using a simplified interface.
- The `mcc` function allows you to package MATLAB code at the command line and offers additional, less common options to control the packaging process.

For more information on the steps required to package and deploy code, see “Steps for Deployment with MATLAB Compiler” on page 1-3.

Get Started with Compiler Apps

Package code using a graphical user interface with compiler apps. The apps provide a simplified interface to specify packaging options. The apps generate an installer that installs your application along with MATLAB Runtime.

The following compiler apps are available with MATLAB Compiler.

- Standalone Application Compiler — Create standalone applications and standalone Windows® applications.
- Excel Add-in Compiler — Create Excel add-ins.
- Web App Compiler — Create web apps for MATLAB Web App Server.

With MATLAB Compiler SDK, you can also use the following apps.

- C Shared Library Compiler — Create C shared libraries.
- C++ Shared Library Compiler — Create C++ shared libraries.
- .NET Assembly Compiler — Create .NET assemblies.
- COM Component Compiler — Create COM components.
- Java Package Compiler — Create Java packages.
- Python Package Compiler — Create Python packages.
- Production Server Archive Compiler — Create archives for MATLAB Production Server.

Compiler app advantages include:

- Perform related deployment tasks with a single intuitive interface.
- Maintain related information in a convenient project file.
- Your project state persists between sessions.
- Load previously stored compiler projects from a prepopulated menu.
- Automatically generate an installer to package applications for distribution.
- Export build settings as a MATLAB deployment script.

Compile Using compiler.build Command Line Functions

Functions in the `compiler.build` family provide a modern command line interface for packaging MATLAB code. Each function is associated with a specific build target and allows you to customize the packaging process using name-value arguments.

`compiler.build` functions do not generate an application installer. To create an installer for an application created using a `compiler.build` function, see `compiler.package.installer`.

Several `compiler.build` functions are available.

- `compiler.build.standaloneApplication` — Create standalone applications.
- `compiler.build.standaloneWindowsApplication` — Create standalone Windows applications.
- `compiler.build.webAppArchive` — Create web app archives.
- `compiler.build.excelAddIn` — Create Excel add-ins.
- `compiler.package.docker` — Create Docker images.

With MATLAB Compiler SDK, you can also use the following functions.

- `compiler.build.cSharedLibrary` — Create C shared libraries.
- `compiler.build.cppSharedLibrary` — Create C++ shared libraries.
- `compiler.build.dotNETAssembly` — Create .NET assemblies.
- `compiler.build.javaPackage` — Create Java packages.
- `compiler.build.pythonPackage` — Create Python packages.
- `compiler.build.productionServerArchive` — Create production server archives.
- `compiler.build.excelClientForProductionServer` — Create Excel add-ins for MATLAB Production Server.
- `compiler.build.comComponent` — Create COM components.
- `compiler.package.microserviceDockerImage` — Create microservice Docker images.

`compiler.build` advantages include:

- If you do not require an installer, it is faster to execute `compiler.build` than use the compiler app workflow.
- Functions have a familiar command line interface similar to other MATLAB functions.
- Perform batch deployment tasks using deployment scripts or using a `compiler.build.<`*`Target`*`>Options` object.
- Use the `compiler.build.Results` output for post-processing tasks, such as creating an installer.
- Packaging options have descriptive names instead of flags.

Specify Additional Packaging Options Using mcc

`mcc` is a MATLAB function that allows you to compile MATLAB code for deployment to any available build target.

`mcc` has many option flags that let you control the output and packaging method. For example, you can use the `-R` option to specify run time options for MATLAB Runtime. Use `mcc` if you require fine control of the packaging process.

`mcc` does not generate an application installer. To create an installer for an application created using `mcc`, see `compiler.package.installer`.

`mcc` advantages include:

- Specify advanced compiler options not available with other packaging methods.
- Run `mcc` in the MATLAB Command Window or at the system command prompt.
- The Spark application target is only available with `mcc`.

Limitations

compiler.build Limitations

- These functions offer most, but not all, of the packaging options available with `mcc`.

  Note If you call a `compiler.build` function with the `Verbose` option set to `true`, the output displays the `mcc` command used to compile the artifact. To specify additional `mcc` build options, you can append them to this command and recompile.
- You cannot create Spark applications using a `compiler.build` function. To create a Spark application, use `mcc`.
- You cannot create Hadoop® jobs using a `compiler.build` function. To create a Hadoop application, use `mcc`.

Compiler App Limitations

- The compiler apps offer most, but not all, of the packaging options available with `mcc`.

  For additional customization, you can specify `mcc` option flags using the **Files required for your application to run** section in any compiler app except Web App Compiler.
- You cannot create Excel add-ins for MATLAB Production Server using a compiler app. To create Excel add-ins for MATLAB Production Server, use the `compiler.build.excelClientForProductionServer` function.
- You cannot create Docker images or microservice images using a compiler app. To create a Docker image, use the `compiler.package.docker` or `compiler.package.microserviceDockerImage` function.
- You cannot create Spark applications using a compiler app. To create a Spark application, use the `mcc` function.
- You cannot create Spark applications using a compiler app. To create a Spark application, use the `mcc` function.
- Compiler projects may not be compatible across MATLAB releases. When you open deployment projects created in releases before R2025a, MATLAB automatically upgrades them to MATLAB projects. For more information, see “Target-Specific Compiler Apps for MATLAB Code Deployment” on page 1-12.

mcc Limitations

- You cannot create web apps using `mcc`. Create web apps using the Web App Compiler app or the `compiler.build.webAppArchive` function.
- You cannot create standalone Docker images using `mcc`. To create a standalone Docker image, use the `compiler.package.docker` function.
- You cannot create microservice Docker images using `mcc`. To create a microservice Docker image, use the `compiler.package.microserviceDockerImage` function.

See Also

`compiler.build.Results` | `mcc` | `compiler.package.installer`

Related Examples

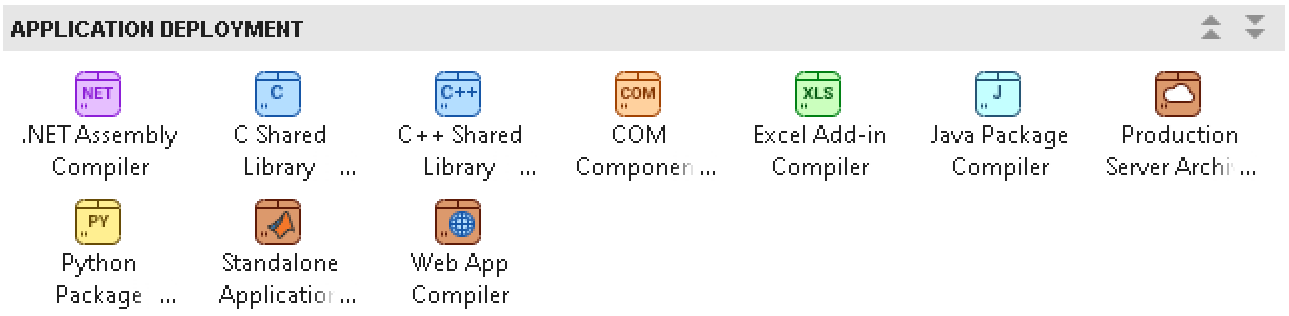
- "Steps for Deployment with MATLAB Compiler" on page 1-3
- "Appropriate Tasks for MATLAB Compiler Products" on page 1-6

Target-Specific Compiler Apps for MATLAB Code Deployment

Starting in R2025a, MATLAB Compiler and MATLAB Compiler SDK include target-specific compiler apps to package MATLAB code. These new apps replace the Application Compiler, Web App Compiler, Library Compiler, and Production Server Compiler apps. The updated design streamlines deployment workflows by providing a dedicated app for each deployment target and integrating these apps with MATLAB projects for better organization and dependency management.

To access the new compiler apps:

1. Click the **Apps** tab in MATLAB.
2. Navigate to the **Application Deployment** section in the drop-down menu.
3. Select the compiler app that corresponds to your deployment target.



Overview of Target-Specific Compiler Apps

Each compiler app includes a tailored user interface, called a compiler task, for configuring and packaging MATLAB code for a specific deployment target. When you use these apps for the first time, you go through a guided workflow to familiarize yourself with the configuration process. Once you understand the process, you can disable the guided workflow for a quicker experience.

MATLAB Projects Integration

Deployment projects are now integrated into MATLAB projects, simplifying the management of packaging configurations and dependencies. Dependencies are automatically managed within the MATLAB project, minimizing the need for manual setup and reducing the likelihood of errors. Packaging configurations are organized as compiler tasks alongside other project components, ensuring that all related assets are stored in a single, centralized location.

Compiler Tasks

The first step to creating a target-specific deployment artifact is to create a compiler task. The compiler task is the main mechanism by which you can configure different aspects of your deployment.

Create Compiler Task

To create a compiler task:

1 Go to the **Apps** tab and locate the app corresponding to your deployment target.
2 Click the app to open the **Create Compiler Task** dialog.
3 Choose one of the following options:

   - Start a new project and create a compiler task – MATLAB creates a new project and a compiler task for your target.
   - Add a compiler task to a recent project – Adds a new compiler task to an existing, recently opened project.
   - Browse for a project and add a compiler task – Select an existing project and add a compiler task to it.

If the MATLAB files you want to deploy are already included in a project, open the project, go to the **Project** tab, in the **Tools** section, click **Compiler Task Manager**. Select a deployment target, and MATLAB creates a compiler task with a corresponding name and opens it in the document area.

Manage Compiler Tasks

A project can have multiple compiler tasks, with each task associated with a different deployment target. For example:

- One compiler task can deploy a MATLAB app as a standalone application.
- Another compiler task can deploy the same app as a web application.

To view all compiler tasks in a project, open **Compiler Task Manager**.

Upgrade Projects

When you open deployment projects created in releases before R2025a, MATLAB automatically upgrades them to MATLAB projects. During this process, the system creates a target-specific compiler task and transfers any existing dependencies into the new project.

Transition of Deployment Apps

Deployment Target	**Compiler App *(Before R2025a)***	**Compiler App *(Starting in R2025a)***
Standalone Application	Application Compiler	Standalone Application Compiler
Excel Add-In	Library Compiler	Excel Add-in Compiler
Deployable Archive (Hadoop)	Library Compiler	No dedicated app. Use the `mcc` command.
Spark	No dedicated app. Use the `mcc` command.	No dedicated app. Use the `mcc` command.
Web Apps	Web App Compiler	Web App Compiler
C Shared Library	Library Compiler	C Shared Library Compiler
C++ Shared Library	Library Compiler	C++ Shared Library Compiler
Generic COM Component	Library Compiler	COM Component Compiler
Java Package	Library Compiler	Java Package Compiler

Deployment Target	**Compiler App *(Before R2025a)***	**Compiler App *(Starting in R2025a)***
.NET Assembly	Library Compiler	.NET Assembly Compiler
Python Package	Library Compiler	Python Package Compiler
Deployable Archive (MATLAB Production Server)	Production Server Compiler	Production Server Archive Compiler
Deployable Archive with Excel Integration (MATLAB Production Server)	Production Server Compiler	No dedicated. Use the `compiler.build.excelClientForProductionServer` function.

See Also

Related Examples

- “Choose Deployment Option” on page 1-8

2

MATLAB Runtime Additional Info

MATLAB Runtime Container

MATLAB Runtime is a standalone set of shared libraries that enables the execution of compiled MATLAB applications or components on computers that do not have MATLAB installed. It's a way to distribute MATLAB code to other users, even if they don't have MATLAB installed themselves. The compiled applications and components are created using MATLAB Compiler and MATLAB Compiler SDK.

You can download a MATLAB Runtime container from the MathWorks container registry for use in your development environment. This may be particularly helpful in CI/CD pipelines where you can streamline integration by enabling quick pulls of the MATLAB Runtime container whenever needed. At present, when fetching a MATLAB Runtime image using Docker on any system, the image retrieved is always based on Linux®.

Starting in R2024b, there are two images available: A full MATLAB Runtime image with GPU support and an image without GPU libraries.

Contents

MATLAB Runtime container includes:

- Ubuntu® Linux base image.
- MATLAB Runtime.
- Dependencies to run MathWorks products.

Requirements

To use a MATLAB Runtime container, you need:

- A host machine with Docker installed.

Download Full MATLAB Runtime Image

In releases prior to R2024b, to download a full MATLAB Runtime container image that contains GPU libraries, execute the following command at a system terminal:

```
docker pull containers.mathworks.com/matlab-runtime:<releaseName>
```

The term `releaseName` refers to the specific version of the MATLAB release. It is denoted with a lowercase `r` followed by the year and edition of the release. For instance, if you want a MATLAB Runtime container for the second edition of the 2023 release, you would specify it as `r2023b`.

Starting in R2024b, download a full MATLAB Runtime container image that contains GPU libraries by executing the following command at a system terminal:

```
docker pull containers.mathworks.com/matlab-runtime:<releaseName>-full
```

The URL has the suffix `-full` to distinguish it from the image without GPU support.

Download MATLAB Runtime Without GPU Libraries

To download a MATLAB Runtime container image for R2025a that does not include any GPU libraries, execute the following command at a system terminal:

```
docker pull containers.mathworks.com/matlab-runtime:r2025a
```

See Also

Related Examples

- “About MATLAB Runtime” on page 7-2

Deploying Standalone Applications

Standalone Applications and Arguments

In this section...
"Pass Command Line Arguments to Applications" on page 3-2
"Handle Input Arguments in MATLAB" on page 3-3
"Display Data to Screen Using MATLAB File" on page 3-4

You can pass arguments to standalone applications created using MATLAB Compiler in the same way that you pass input arguments to any console-based application on the command line. Type the application name followed by one or more input arguments separated by spaces.

When you run a standalone application, the main program passes any command line arguments to your *main* function—the function you listed first when you created the application.

Pass Command Line Arguments to Applications

Refer to the following table for examples on passing different argument types, such as files, numbers or letters, matrices, and MATLAB variables to a standalone application named `myapp`.

To Pass....	Use This Syntax....	Notes
A file named `helpfile`	`myapp path/to/helpfile`	If the path to the file contains spaces, surround the path with double quotes.
Numbers or letters	`myapp 1 2 3 a b c`	Do not use commas or other separators between the numbers and letters you pass.
Matrices as input	`myapp "[1 2 3]" "[4 5 6]"`	Place double quotes around each matrix argument.

MATLAB variables	At the MATLAB Command Window, type:
	`for k=1:10`
	`cmd = ['myapp ',string(k)];`
	`system(cmd);`
	`end`
		To pass a numeric MATLAB variable to a program as input, you must convert it to a character vector or string.

You can also call a standalone application from MATLAB. The following examples show how to pass the numbers and letters `1 2 3 a b c` to a standalone application named `myapp` at the MATLAB command window.

Using system, dos, or unix

Specify the entire command as a character vector (including input arguments) using the `system` command.

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

Using ! (Bang) Operator

Use the `!` (bang) operator from within MATLAB.

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

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

Using Windows Batch File

To run a standalone application with arguments by double-clicking it, you can create a batch file that calls the standalone application with the specified input arguments. For example, create `runmyapp.bat` with the following code.

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

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

The last two lines of code in `runmyapp.bat` ensure 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 the icon for `runmyapp.bat`.

Handle Input Arguments in MATLAB

By default, the input arguments you pass to your standalone application from a system prompt are received as character vector input. MATLAB code that accepts character vectors or strings as inputs does not need to be modified before packaging.

If your code expects the data in a different format (for example, `double`), you must do one or both of the following:

- Convert the character vector input to the required format in your MATLAB code before processing it. For example, you can use the `string` and then `double` functions to convert the character vector input to numerical data.
- Create your application with the behavior to automatically treat numeric inputs as MATLAB doubles by using the `TreatInputsAsNumeric` option with `compiler.build.standaloneApplication`, the **Treat all inputs to the app as numeric MATLAB doubles** option in the Standalone Application Compiler app, or the `mcc -n` flag. For details on the differences between packaging options, see “Choose Deployment Option” on page 1-8.

Here are two methods to convert input arguments to character vectors in your MATLAB code.

Method 1

Use `ischar` to test whether the input argument `z` is a character vector, and if so, convert it to a `double`.

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

if ischar(z)
    z=double(string(z));
else
```

```
    z=z;
end
x=2*z
y=z^2;
disp(y)
```

Method 2

Use `isdeployed` to test whether the function is running in deployed mode, and if so, convert `z` to a `double`.

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

if isdeployed
    z=double(string(z));
end
x=2*z
y=z^2;
disp(y)
```

Display Data to Screen Using MATLAB File

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. By default, deployed applications output text to the standard output and standard error streams.

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

- Do not use semicolons to suppress commands that yield your return data.
- Use the `disp` function to display the variable value, then redirect the output to other applications using command line redirects (the > operator) or pipes (||) on non-Windows systems.
- Display data visually using MATLAB graphics, such as with the `plot` function. Graphics are displayed in new windows when you run the application, and the application runs until all graphics are closed.

For example, create a program that plots a histogram of the number of times each letter in the alphabet appears in an input string. The function takes a single input and has zero outputs.

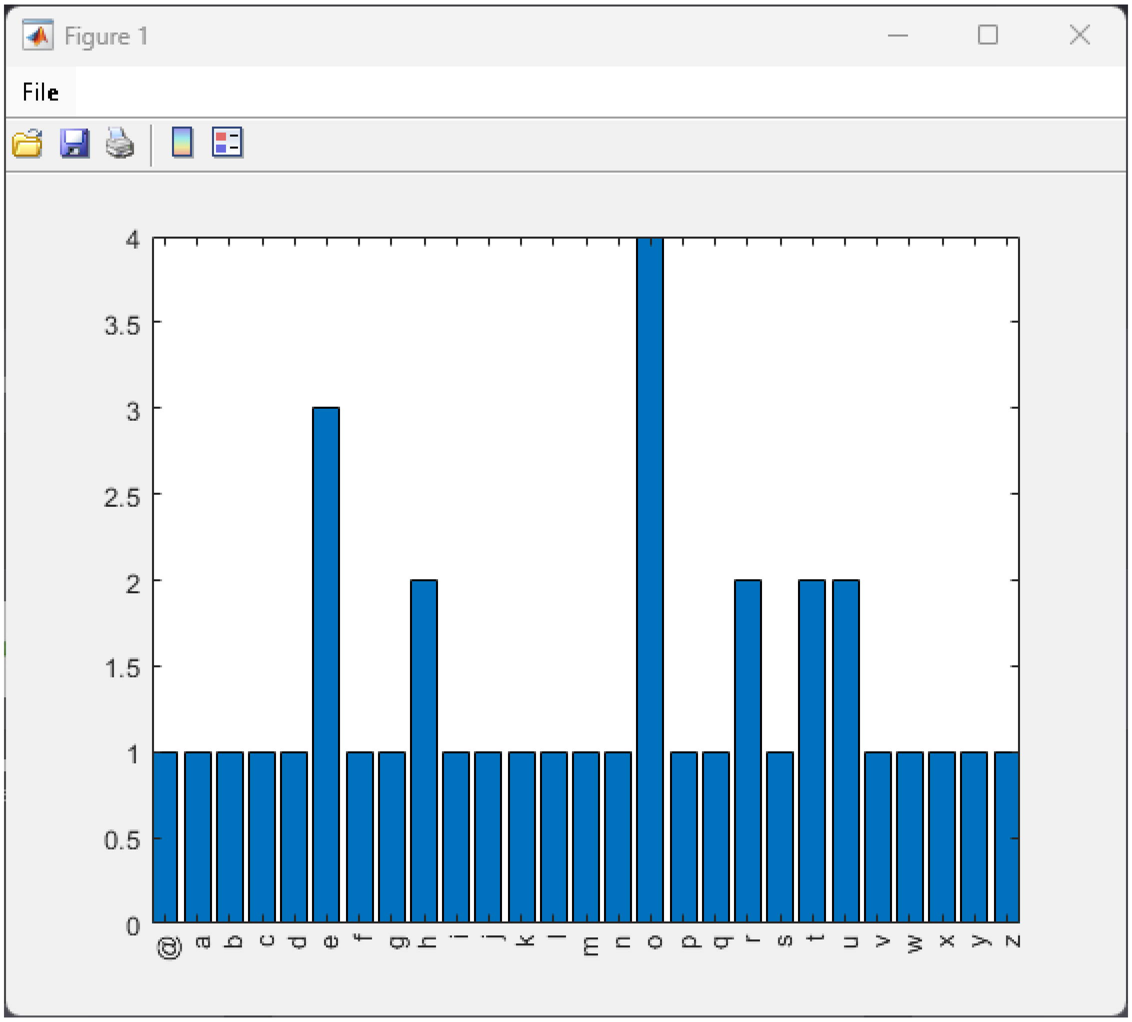
```
function freq(msg)
  % Remove spaces
  msg(msg == ' ') = [];

  % Convert to lower case, and map the letters to numbers.
  % a=1, b=2, etc.
  msg = lower(msg) - double('a') + 1;

  % Map non-letter characters to zero.
  msg(msg < 1) = 0;
  msg(msg > 26) = 0;

 % Display a histogram of the letter frequency in the message.
  count = hist(msg, 27);
  bar(0:26, count);
  labels = char([double('@'), double('a'):double('z')])';
  set(gca, 'XTick', 0:26+0.5, 'XTickLabel', labels);
 axis tight
```

Create a standalone application using `compiler.build.standaloneApplication`.

```
compiler.build.standaloneApplication('freq.m',Verbose=true);
```

Test the application at the MATLAB Command Window.

```
!freq "The quick brown fox jumps over the lazy dog!"
```



See Also

`compiler.build.standaloneApplication` | Standalone Application Compiler | `mcc`

More About

- “Write Deployable MATLAB Code” on page 5-10
- “Choose Deployment Option” on page 1-8
- “Create Standalone Application from MATLAB” on page 18-2
- “Include and Access Files in Packaged Applications” on page 5-15

Use Parallel Computing Toolbox in Deployed Applications

An application that uses the Parallel Computing Toolbox can use cluster profiles that are in your MATLAB preferences folder. To find this folder, use `prefdir`.

For instance, when you create a standalone application, all of the profiles available in your Cluster Profile Manager will be available in the application.

Your application can also use a cluster profile given in an external file. To enable your application to use this file, you can either:

1. Link to the file within your code.
2. Pass the location of the file at run time.

Export Cluster Profile

To export a cluster profile to an external file:

1. In the Home tab, in the **Environment** section, select **Parallel** > **Create and Manage Clusters**.
2. In the **Cluster Profile Manager** dialog, select a profile, and in the **Manage** section, click **Export**.

Link to Parallel Computing Toolbox Profile Within Your Code

To enable your application to use a cluster profile given in an external file, you can link to the file from your code. In this example, you will use absolute paths, relative paths, and the MATLAB search path to link to cluster profiles. Note that since each link is specified before you compile, you must ensure that each link does not change.

To set the cluster profile for your application, you can use the `setmcruserdata` function.

As your MATLAB preferences folder is bundled with your application, any relative links to files within the folder will always work. In your application code, you can use the *`myClusterProfile.mlsettings`* file found within the MATLAB preferences folder.

```
mpSettingsPath = fullfile(prefdir, 'myClusterProfile.mlsettings');
setmcruserdata('ParallelProfile', mpSettingsPath);
```

The function `fullfile` gives the absolute path for the external file. The argument given by `mpSettingsPath` must be an absolute path. If the user of your application has a cluster profile located on their file system at an absolute path that will not change, link to it directly:

```
mpSettingsPath = '/path/to/myClusterProfile.mlsettings';
setmcruserdata('ParallelProfile', mpSettingsPath);
```

This is a good practice if the cluster profile is centrally managed for your application. If the user of your application has a cluster profile that is held locally, you can expand a relative path to it from the current working directory:

```
mpSettingsPath = fullfile(pwd, '../rel/path/to/myClusterProfile.mlsettings');
setmcruserdata('ParallelProfile', mpSettingsPath);
```

This is a good practice if the user of your standalone application should supply their own cluster profile. Any files that you add to your application at compilation are added to the MATLAB search path. Therefore, you can also bundle a cluster profile that is held externally with your application. First, use `which` to get the absolute path to the cluster profile. Then, link to it.

```
mpSettingsPath = which('myClusterProfile.mlsettings');
setmcruserdata('ParallelProfile', mpSettingsPath);
```

Finally, compile at the command line and add the cluster profile.

```
mcc -a /path/to/myClusterProfile.mlsettings -m myApp.m;
```

Note that to run your application before you compile, you must manually add `/path/to/` to your MATLAB search path.

Pass Parallel Computing Toolbox Profile at Run Time

If the user of your application *myApp* has a cluster profile that is selected at run time, you can specify this at the command line.

```
myApp -mcruserdata ParallelProfile:/path/to/myClusterProfile.mlsettings
```

Note that when you use the `setmcruserdata` function in your code, you override the use of the `-mcruserdata` flag.

Switch Between Cluster Profiles in Deployed Applications

When you use the `setmcruserdata` function, you remove the ability to use any of the profiles available in your Cluster Profile Manager. To re-enable the use of the profiles in **Cluster Profile Manager**, use the `parallel.mlSettings` file.

```
mpSettingsPath = '/path/to/myClusterProfile.mlsettings';
setmcruserdata('ParallelProfile', mpSettingsPath);

% SOME APPLICATION CODE

origSettingsPath = fullfile(prefdir, 'parallel.mlsettings');
setmcruserdata('ParallelProfile', origSettingsPath);

% MORE APPLICATION CODE
```

Sample C Code to Load Cluster Profile

You can call the `mcruserdata` function natively in C and C++ applications built with MATLAB Compiler SDK.

```
mxArray *key = mxCreateString("ParallelProfile");
mxArray *value = mxCreateString("/path/to/myClusterProfile.mlsettings");
if (!setmcruserdata(key, value))
{
    fprintf(stderr,
            "Could not set MCR user data: \n %s ",
            mclGetLastErrorMessage());
    return -1;
}
```

See Also

`setmcruserdata` | `getmcruserdata`

Related Examples

- “Using MATLAB Runtime User Data Interface” on page 8-3
- “Specify Parallel Computing Toolbox Profile in .NET Application” (MATLAB Compiler SDK)
- “Specify Parallel Computing Toolbox Profile in Java Application” (MATLAB Compiler SDK)

Integrate Application with Mac OS X Finder

In this section...

"Overview" on page 3-9
"Installing the Mac Application Launcher Preference Pane" on page 3-9
"Configuring the Installation Area" on page 3-9
"Running the Application" on page 3-11

Overview

Mac graphical applications, opened through the Mac OS X finder utility, require additional configuration if MATLAB software or MATLAB Runtime are not installed in default locations.

Installing the Mac Application Launcher Preference Pane

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

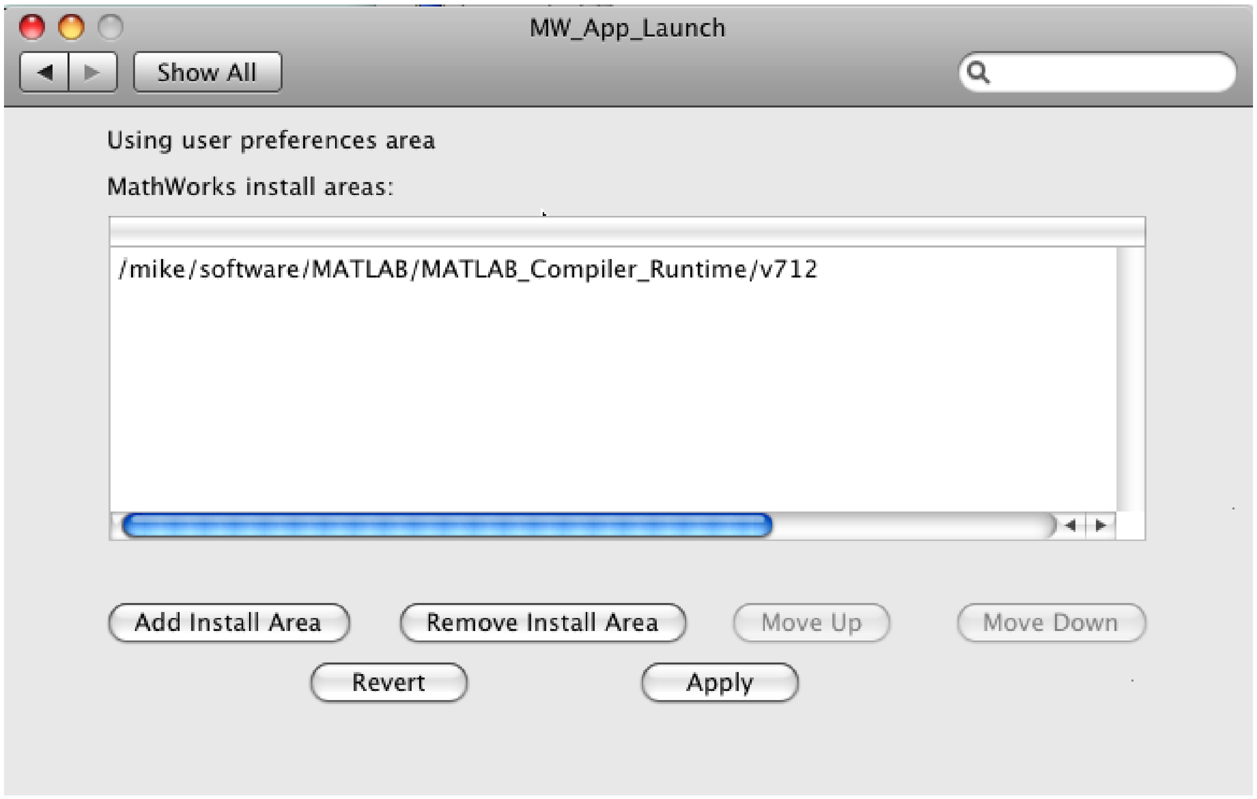
1. In the Mac OS X Finder, navigate to `install_area/toolbox/compiler/maci64`.
2. Double-click **MW_App_Launch.prefPane**.

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

Configuring the Installation Area

After you install the preference pane, you configure the installation area.

1. Open the preference pane by clicking the apple logo in the upper left corner of the desktop.
2. Click **System Preferences**. The **MW_App_Launch** preference pane appears in the **Other** area.



Using user preferences area

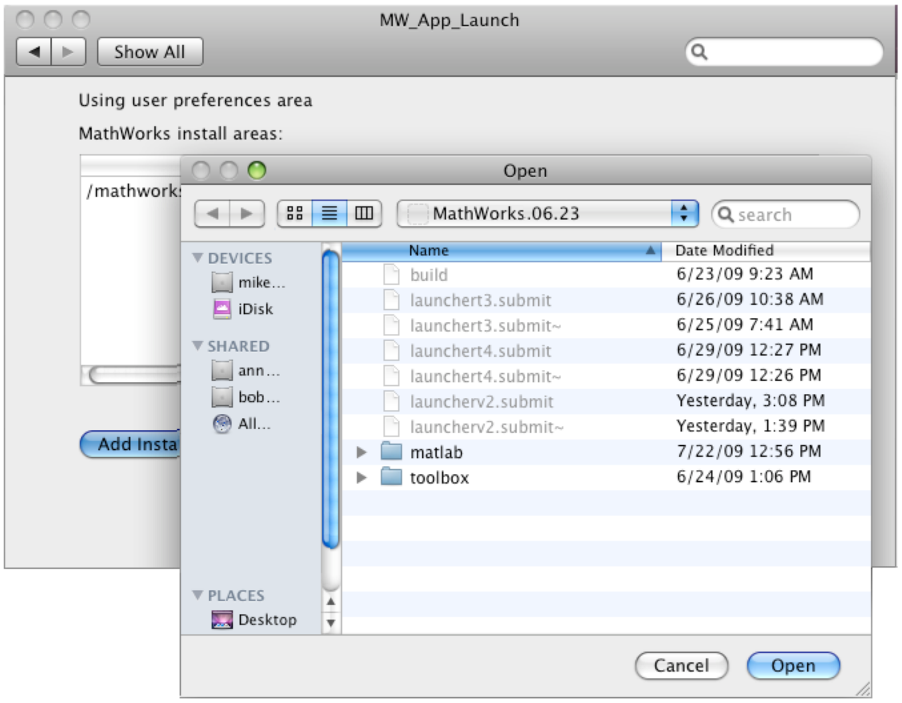
MathWorks install areas:

/mike/software/MATLAB/MATLAB_Compiler_Runtime/v712

3 Define an installation area on your system by clicking **Add Install Area**.

4 Define the default installation path by browsing to it.

5 Click **Open**.



Using user preferences area

MathWorks install areas:

/mathworks

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** — Define the path on your system where your applications install by default.
- **Remove Install Area** — Remove a previously defined installation area.
- **Move Up** — After selecting an installation area, click 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 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** — Save changes and exit MathWorks Application Launcher.
- **Revert** — Exit MathWorks Application Launcher without saving any changes.

Running the Application

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

The location of the bundle is determined by whether you use `mcc`, `compiler.build.standaloneApplication`, or the Standalone Application Compiler app to build the application:

- If you use the Standalone Application Compiler app, the application bundle is placed in the `build` subfolder of the output folder.
- If you use `mcc`, the application bundle is placed in the current working folder or in the output folder, as specified by the `mcc -d` switch.
- If you use `compiler.build.standaloneApplication`, the application bundle is placed in the *`ExecutableName`*`standaloneApplication` folder or in the output folder, as specified by the `OutputDir` option.

See Also

Standalone Application Compiler | `mcc` | `compiler.build.standaloneApplication`

More About

- "Create Standalone Application from MATLAB" on page 18-2

Protect Code and Data in Deployable Archive

You can protect deployed application code with packaging options that allow you to obscure files and folder names, store secret information, and encrypt the deployable archive. Use these options separately or together to increase the security of your application.

For general information on protecting MATLAB source code, see “Security Considerations to Protect Your Source Code”.

Protect User Data and Obfuscate File Structure

When MATLAB Compiler creates a deployable archive, MATLAB code files (plain text MATLAB files or P-code files) are encrypted using the standard AES-256 algorithm. By default, the names of files and the directory structure are not obscured and other file types (such as MAT, FIG, MEX, and so on) are not encrypted.

For all deployment targets, you can obscure the names of files and the directory structure and also encrypt other file types (such as MAT, FIG, MEX, and so on). The encrypted files remain encrypted on the disk but are decrypted in memory to their original form at run time.

You can protect your MATLAB code and data using one or more of the following options, depending on your packaging method.

- Use the `mcc -s` option to obfuscate folder structures and file names in the deployable archive (CTF file) and place user code and data into a user package within the archive. During runtime, MATLAB code and data is decrypted and loaded directly from the user package rather than extracted to the file system.
- Use the `mcc -j` option to automatically convert all `.m` files to P-files before packaging.
- Use the `ObfuscateArchive` option in a `compiler.build` function to obfuscate folder structures and file names in the deployable archive, and convert all `.m` files into P-files before packaging. This option is equivalent to using `mcc` with both `-s` and `-j` specified.
- Use the `-s` or `-j` options in the **Additional Runtime Settings** area of the compiler apps.

For information on selecting a packaging method, see “Choose Deployment Option” on page 1-8.

Package Code with Sensitive Information Using Secrets

For standalone applications and web apps, if the MATLAB code you want to deploy handles sensitive strings of data, such as passwords, you can avoid putting them in your MATLAB code by storing them in your MATLAB vault as secrets. Then, you can package the deployable archive with secrets, which are decrypted at run time.

To include secrets in the deployable archive, store each secret in your MATLAB vault before packaging using `setSecret`. Retrieve the secrets in your deployed MATLAB code using `getSecret`. Then, specify the secret names in a secrets manifest JSON file using one of the following options, depending on your packaging method.

- Use the `SecretsManifest` option in a `compiler.build` function.
- Use the `mcc -J` option.
- Use the `-J` option in the **Additional Runtime Settings** area of the compiler apps.

For more information on deploying code that uses secrets, see “Handle Sensitive Information in Deployed Applications” on page 18-17.

For an example on creating a standalone application that uses secrets, see “Access Sensitive Information in Standalone Application” on page 18-14.

Require Decryption Key at Run Time

To control application access at run time, you can specify an AES encryption key and a MEX file loader interface to retrieve the decryption key.

Specify key and loader files during packaging using one of the following equivalent options, depending on your packaging method.

- Use the `ExternalEncryptionKey` option in a `compiler.build` function.
- Use the `mcc -k` option.
- Use the `-k` option in the **Additional Runtime Settings** area of the compiler apps.

For more details, see the `mcc -k` entry.

Package C++ Shared Libraries without MEX Loader

For C++ shared libraries, as an alternative to specifying both key and MEX loader at compile time, you can specify only the encryption key. You then provide the hex encoded 64 byte decryption key at runtime in your C++ application as an argument for the `initMATLABLibrary` function using the MATLAB Data API or the `<library>InitializeWithKey` function using the `mwArray` API. For this workflow, the syntax is:

```
mcc mfilename1 -W 'cpplib:library_name' -k '<keyfile>'
```

See Also

`compiler.build.standaloneApplication` | Standalone Application Compiler | `mcc` | `getSecret`

Related Examples

- “Security Considerations to Protect Your Source Code”
- “Choose Deployment Option” on page 1-8
- “Keep Sensitive Information Out of Code”
- “Access Sensitive Information in Standalone Application” on page 18-14

Files Generated After Packaging MATLAB Functions

When you create a deployable component using MATLAB Compiler, files that correspond to the build target are generated in the output folder. Depending on which packaging method you use, files may be located in various subfolders in the output folder. For more details on the available packaging methods, see “Choose Deployment Option” (MATLAB Compiler SDK). For a list of files generated by MATLAB Compiler SDK, see “Files Generated After Packaging MATLAB Functions” (MATLAB Compiler SDK).

The `compiler.build` family of functions place output files in a folder named after the main file and target type. For instance, `magicsquareStandaloneApplication`.

The Compiler apps generate an `output` folder and place output files in a subfolder named `build`. The apps optionally create a `package` subfolder if you create an installer. The installer installs all of the binary artifacts required for distributing a compiled component and optionally installs MATLAB Runtime. You can also generate an installer using the `compiler.package.installer` function. Distribute the installer to users who do not have MATLAB installed on their machines. If you do not create an installer, you can manually distribute the set of files required to integrate the component according to the component type.

Since R2025a: For information on files generated using a prior version of the Compiler apps, see Files Generated After Packaging MATLAB Functions.

Files Generated by MATLAB Compiler

MATLAB Compiler generates the following files in the build output folder. The intermediate artifacts not listed here are generated as a result of packaging of the MATLAB files. They are not significant to the user.

All Targets

The following files are generated for all deployment targets.

File	Description
`buildresult.json`	JSON file containing information on runtime dependencies included in the package. The information corresponds to the `RuntimeDependencies` property of the `compiler.build.Results` object.
`GettingStarted.html`	HTML file containing packaging information and next steps.
`includedSupportPackages.txt`	Lists all support files included in the package.
`mccExcludedFiles.log`	Log file that contains a list of any toolbox functions that were not included in the package. For information on excluding data files, see `%#exclude`.
`PackagingLog.html`	HTML file containing information on the `mcc` command used and output from the packaging process.

File	Description
readme.txt	Contains information on deployment prerequisites and the list of files to package for deployment.
requiredMCRProducts.txt	Contains product IDs of products required to run the package. For more information on product IDs, see matlab.codetools.requiredFilesAndProducts. This file will be removed in a future release.
unresolvedSymbols.txt	Lists dependencies not found during packaging. If this file is not empty, you must locate the required dependencies and place them in the search path before recompiling.

Excel Add-in

File	Description
_install.bat	The file that registers the generated dll file.
filename.bas	VBA module file that can be imported into a VBA project.
filename.xla	Excel add-in that can be added directly to Excel. You do not need both .bas file and .xla file; one of them is sufficient.
*filename*_2_0.dll	The generated dll that needs to be registered using mwregsvr.exe or regsvr32.exe.

Standalone Application

File	Description
filename.exe (Windows) or *filename* (Linux or Mac)	Standalone executable file that contains your application.
run_*filename*.sh (Linux and Mac only)	Shell script file that sets the library path and executes the application. This file is only generated on Linux and Mac systems.
splash.png	File that is used as the splash screen image. When the executable starts, the file is read from the same folder where the executable is located, and the splash screen is displayed.

Web App

File	Description
filename.ctf	Deployable archive file that contains your web app.

See Also
mcc | compiler.build.Results

More About

- “Create Standalone Application from MATLAB” on page 18-2
- “Create Excel Add-In from MATLAB”

Distribute MATLAB Compiler SDK Files to Application Developers

In this section...

"Distribute COM Components" on page 3-17
"Distribute C/C++ Shared Libraries" on page 3-17
"Distribute Java Packages" on page 3-18
"Distribute .NET Assemblies" on page 3-18
"Distribute Python Packages" on page 3-18

After you create a component using MATLAB Compiler SDK, distribute files and integrate them in an application in the target language.

The Compiler apps optionally generate an installer that packages all of the binary artifacts required for distributing a compiled component. The installer is located in the `package` folder of the compiler project. You can also generate an installer using the `compiler.package.installer` function. If you do not create an installer, manually distribute the set of files required to integrate the component according to the component type. For more details on the available methods to package functions, see "Choose Deployment Option" (MATLAB Compiler SDK).

Depending on the deployment target, there may be additional steps required before you can run the application. For instance, if you build a C++ shared library and then write code for a C++ application, you can compile the C++ application executable using `mbuild`. Files generated after packaging are not included in the installer generated by the Compiler app. You can manually distribute the executable along with MATLAB Runtime or include the executable in an installer using the `AdditionalFiles` option of the `compiler.package.installer` function.

In order to run the application, the target machine must have access to MATLAB Runtime that matches the version of MATLAB used to compile the component, at the same update level or newer. For details, see "About MATLAB Runtime" (MATLAB Compiler SDK).

Distribute COM Components

Distribute the following files to integrate a component in an application.

- *`packageName`*`.dll` — COM component
- `_install.bat` — script that registers the component (to register manually, see "Register COM Component" (MATLAB Compiler SDK))
- Function signatures of the deployed MATLAB functions (for details, see "Customize Code Suggestions and Completions")

Distribute C/C++ Shared Libraries

Distribute the following files to integrate a C/C++ shared library in an application.

- One of the following:
  - *`libraryName`*`.lib`/`.dylib`/`.so` — `mwArray` API shared library

  - `*libraryName*.ctf` — MATLAB Data API deployable archive
- `*libraryName*.h/.hpp` — header file
- Function signatures of the deployed MATLAB functions (for details, see "Customize Code Suggestions and Completions")

Distribute Java Packages

Distribute the following files to integrate a Java package in an application.

- *packageName*`.jar` — Java package
- Function signatures of the deployed MATLAB functions (for details, see "Customize Code Suggestions and Completions")

Include directions for adding the required JAR files to the Java CLASSPATH.

At a minimum, the CLASSPATH must include:

- *mcrroot*`/toolbox/javabuilder/jar/javabuilder.jar`
- Generated Java package
- JAR files for the application

Distribute .NET Assemblies

Distribute the following files to integrate a .NET assembly in an application.

- One of the following:
  - *libraryName* `.dll` — `MWArray` API assembly
  - *libraryName* `.ctf` — MATLAB Data API deployable archive
- Function signatures of the deployed MATLAB functions (for details, see "Customize Code Suggestions and Completions")
- *assemblyName*`.xml` — documentation files (optional)
- *assemblyName*`.pdb` — program database file containing debugging information (optional)

Distribute Python Packages

Distribute the following files to integrate a Python package in an application.

- `setup.py` — Python installer
- `_init_.py` — initialization script for the Python package
- *packageName* `.ctf` — deployable archive
- Function signatures of the deployed MATLAB functions (for details, see "Customize Code Suggestions and Completions")

See Also

Related Examples

- “Files Generated After Packaging MATLAB Functions” (MATLAB Compiler SDK)
- “Choose Deployment Option” (MATLAB Compiler SDK)
- “About MATLAB Runtime” (MATLAB Compiler SDK)
- “Customize Code Suggestions and Completions”

Customizing a Compiler Project

Manage Support Packages

Many MATLAB toolboxes use support packages to interact with hardware or to provide additional processing capabilities. If the MATLAB code you want to package has support package dependencies, MATLAB Compiler can detect them automatically using a dependency analysis process.

The dependency analysis process creates a list of all installed support packages that your MATLAB code requires. The list is determined based on these criteria:

- MATLAB Compiler detects that your code has a dependency on the support package.
- MATLAB Compiler detects a dependency on a product for which you have one or more support packages installed.
- Your code is dependent on the base product of the support package.

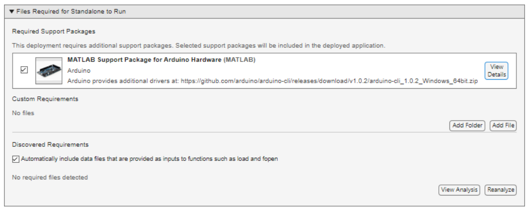
For more information, see “Dependency Analysis Using MATLAB Compiler” on page 5-2.

You can modify the list of included support packages using one of the following workflows, depending on your packaging method. For more information on selecting a packaging method, see “Choose Deployment Option” on page 1-8.

Some support packages require third-party drivers that the compiler cannot package. In this case, you need to instruct end users to download and install the required drivers.

Using Compiler App

If your MATLAB code uses a toolbox with an installed support package, the compiler apps display the package in the **Required Support Packages** section.



▼ Files Required for Standalone to Run

Required Support Packages

This deployment requires additional support packages. Selected support packages will be included in the deployed application.

☑ MATLAB Support Package for Arduino Hardware (MATLAB)
Arduino
Arduino provides additional drivers at: https://github.com/arduino/arduino-cli/releases/download/v1.0.2/arduino-cli_1.0.2_Windows_64bit.zip

Custom Requirements

No files

Discovered Requirements

☑ Automatically include data files that are provided as inputs to functions such as load and fopen

No required files detected

Deselect support packages that are not required by your application.

To inform end users of required dependencies, such as the need to install a third-party driver, you can edit the installation notes in the **Installer Notes** section of the app.

Using compiler.build

You can use the `SupportPackages` option with functions in the `compiler.build` family, such as `compiler.build.standaloneApplication`, to specify the method for including support packages.

- `"autodetect"` (default) — The dependency analysis process detects and includes the required support packages automatically. This is the default option.
- `"none"` — No support packages are included. Using this option can cause run-time errors.

- A string scalar, character vector, or cell array of character vectors — Only the specified support packages are included. To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

  For example, specify two support packages, `Deep Learning Toolbox Converter for TensorFlow Models` and `Deep Learning Toolbox Model for Places365-GoogLeNet Network`, using `compiler.build.standaloneApplication`.

  ```
  compiler.build.standaloneApplication("myapp.m", ...
  "SupportPackages",{"Deep Learning Toolbox Converter for TensorFlow Models", ...
  "Deep Learning Toolbox Model for Places365-GoogLeNet Network"})
  ```

Using mcc

If your MATLAB code uses a toolbox with an installed support package, use the`-a` flag with the `mcc` command when packaging the code to specify supporting files in the support package folder. For example, if your function uses the `Sound Card Support from Data Acquisition Toolbox` support package, run the following command:

```
mcc -m -v test.m -a C:\MATLAB\SupportPackages\R2025b\toolbox\daq\supportpackages\daqaudio ...
-a 'C:\MATLAB\SupportPackages\R2025b\resources\daqaudio'
```

See Also

Related Examples

- “Dependency Analysis Using MATLAB Compiler” on page 5-2
- “Choose Deployment Option” on page 1-8
- “Target-Specific Compiler Apps for MATLAB Code Deployment” on page 1-12

MATLAB Code Deployment

- “Dependency Analysis Using MATLAB Compiler” on page 5-2
- “About Deployable Archives” on page 5-5
- “MATLAB Runtime Component Cache and Deployable Archive Embedding” on page 5-8
- “Write Deployable MATLAB Code” on page 5-10
- “Calling Shared Libraries in Deployed Applications” on page 5-14
- “Include and Access Files in Packaged Applications” on page 5-15

Dependency Analysis Using MATLAB Compiler

In this section...

"Function Dependency" on page 5-2

"License and Toolbox Dependencies" on page 5-3

"Data File Dependency" on page 5-3

"Troubleshoot Missing Dependencies" on page 5-3

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 analysis process cannot resolve overloaded methods at package time. MATLAB Compiler dependency analysis also processes `include`/`exclude` files on each pass.

Tip To improve package time performance and lessen application size, prune the path with the `mcc` command's `-N` and `-p` flags. You can also specify files to include using the `mcc -a` flag, the **Custom Requirements** section in a compiler app, or the `AdditionalFiles` option in a `compiler.build` function.

Function Dependency

The dependency analysis process 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 cannot determine the p-file's dependencies.
- `.fig` files
- MEX-files

MATLAB Compiler is not able to discover functions called through an `feval`, `eval`, Handle Graphics® callback, or objects loaded from MAT files. To explicitly include the function(s), use the `%#function` pragma. For example:

```
function foo
  %#function bar

     feval('bar');

  end %function foo
```

The line `%#function bar` notifies MATLAB Compiler that function `bar` will be included in the compilation and is called through `feval`.

For more information on including files in your deployable artifact, see "Include and Access Files in Packaged Applications" on page 5-15.

Include MEX Files, DLLs, or Shared Libraries

When you compile MATLAB functions containing MEX files, ensure that the dependency analysis process can find them. In particular, note that:

- Since the dependency analysis function cannot examine MEX files, DLLs, or shared libraries to determine their dependencies, explicitly include all executable files these files require.
- If you have any doubts that a MATLAB function called by a MEX file, DLL, or shared library can be found during dependency analysis, then manually include that function.
- Not all functions are compatible with the compiler. Check the file `mccExcludedFiles.log` after your build completes. This file lists all functions called from your application that you cannot deploy.

License and Toolbox Dependencies

For details on determining required toolboxes, see the MATLAB answers post How do I determine the required toolboxes and licenses for my MATLAB code?.

Data File Dependency

In addition to executable content, MATLAB Compiler can detect and automatically include files that your MATLAB functions access by calling any of these functions: `audioinfo`, `audioread`, `csvread`, `daqread`, `dlmread`, `fileread`, `fopen`, `imfinfo`, `importdata`, `imread`, `load`, `matfile`, `mmfileinfo`, `open`, `readtable`, `type`, `VideoReader`, `xlsfinfo`, `xlsread`, `xmlread`, and `xslt`.

The compiler app automatically adds detected data files to the deployable archive.

To ensure that a specific file is included and can be accessed in the compiled application, specify the file without a path. This means the file should be accessible from the current directory or any directory listed in the MATLAB path. For example:

```
fileread('myfile.ext')
```

Dependency analysis relies on the MATLAB path to find the file. If you specify a full path to the file in your MATLAB code, the dependency analysis process may not be able to find the file.

Exclude Files From Package

To ignore data files during dependency analysis, use one or more of the following options. For examples on how to use these options together, see `%#exclude`.

- Use the `%#exclude` pragma in your MATLAB code to ignore a file or function during dependency analysis.
- Use the `-X` flag in your `mcc` command to ignore all data files detected during dependency analysis.
- Use the `AutoDetectDataFiles` option in a `compiler.build` function to control whether data files are automatically included in the package. Setting this to `false`/`'off'`/`0` is equivalent to using `-X`.

Troubleshoot Missing Dependencies

If one or more dependencies cannot be found, MATLAB Compiler populates the `unresolvedSymbols.txt` file with a list of the missing items. If this file is not empty, you must locate the required dependencies and ensure they are available by placing them in the search path. Then, repackage your project. If any dependencies are not deployable, you can still use the functionality in your code before it is deployed by using the `isdeployed` boolean.

Some functionality requires a specific MathWorks product. You can use the function `matlab.codetools.requiredFilesAndProducts` to display a list of MATLAB files and MathWorks products that may be required to run the specified MATLAB program files.

For more information on MATLAB Compiler limitations and troubleshooting help, see “Limitations” on page 13-2.

See Also

`compiler.build.standaloneApplication` | Standalone Application Compiler | `mcc` | `isdeployed`

More About

- “About Deployable Archives” on page 5-5
- “Steps for Deployment with MATLAB Compiler” on page 1-3
- “Write Deployable MATLAB Code” on page 5-10
- “Include and Access Files in Packaged Applications” on page 5-15
- “Limitations” on page 13-2

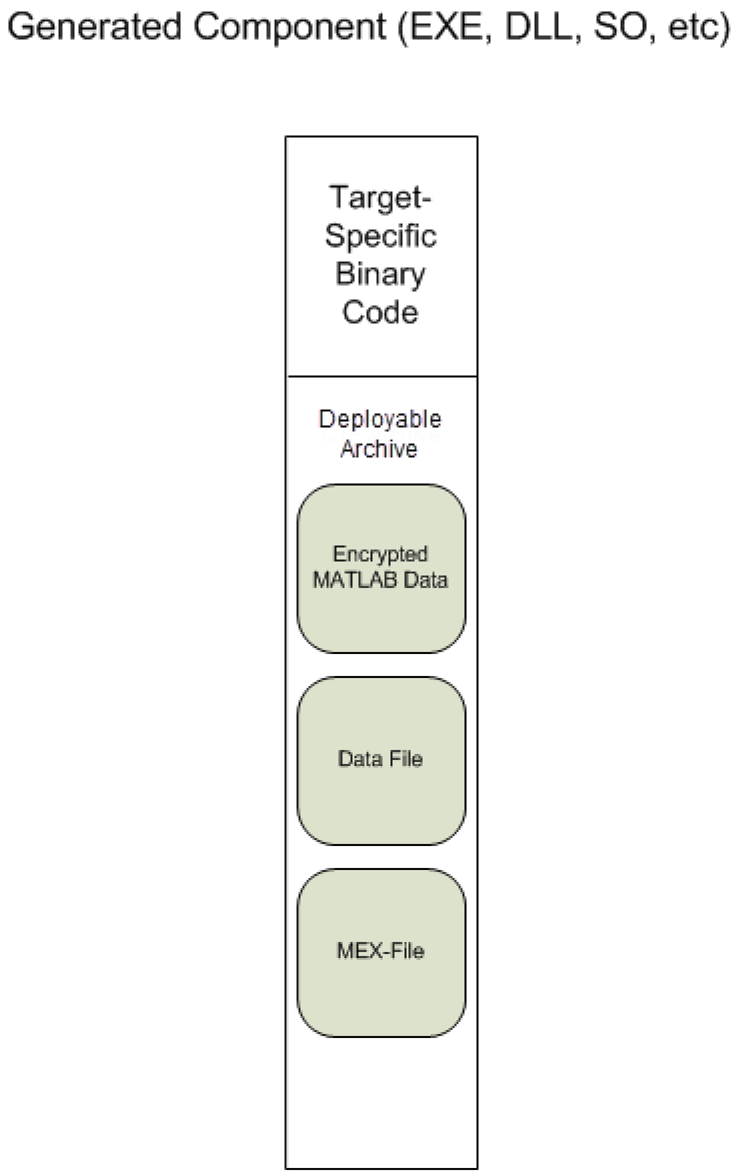
About Deployable Archives

When MATLAB Compiler or MATLAB Compiler SDK creates an application or shared library, it bundles the content into a deployable archive, which is a CTF file that contains all the MATLAB based content (MATLAB files, MEX-files, and so on) included in the application.

All MATLAB files (`.m` and `.p` files) included in the deployable archive are encrypted using the Advanced Encryption Standard (AES) cryptosystem. By default, the names of files and the directory structure are not obscured and other file types, including MEX files, MAT files, FIG files, Java JAR or class files, are not encrypted. Every other type of file is copied, unchanged, into the archive. When the deployable application runs, the files in the CTF archive are extracted onto the disk, and any files that were encrypted in the archive remain encrypted on the disk. If you choose to extract the deployable archive as a separate file, the files also remain encrypted.

Caution Do not extract the files within the `.ctf` file and place them individually under version control. Since the `.ctf` file contains interdependent MATLAB functions and data, the files within it must be accessed only by accessing the `.ctf` file. For best results, place the entire `.ctf` file under version control.

There are several packaging options to increase the security of a packaged application. For details, see “Protect Code and Data in Deployable Archive” on page 3-12.


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

Additional Details

Multiple deployable archives, such as those generated with COM components, .NET assemblies, or Excel add-ins, 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 deployable archives into another deployable archive and distribute them.

All the MATLAB files from a given deployable archive associate with a unique cryptographic key. MATLAB files with different keys, placed in the same deployable 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 deployable archive.

The compiler deletes the deployable 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 deployable archives. If you do, you can possibly strip the deployable archive from the binary, resulting in run time errors for the driver application.

See Also

Related Examples

- “MATLAB Runtime Component Cache and Deployable Archive Embedding” on page 5-8
- “About MATLAB Runtime” on page 7-2
- “Protect Code and Data in Deployable Archive” on page 3-12

MATLAB Runtime Component Cache and Deployable Archive Embedding

For all deployment targets except for Java, compiled application data is embedded directly in the deployable archive by default and extracted to a temporary folder at run time. This temporary folder is known as the MATLAB Runtime component cache. When a compiled application runs, the files in the deployable archive (CTF file) are extracted to the MATLAB Runtime component cache. For details on data extraction with Java packages, see “Define Embedding and Extraction Options for Deployable Java Archive” (MATLAB Compiler SDK).

Cache Size and Location

The default maximum cache size is 32 MB in releases prior to R2024b, or 1024 MB starting in R2024b. The default component cache location is listed in the following table.

Platform	Before R2024b	Since R2024b
Windows	`%LOCALAPPDATA%\Temp\%USERNAME%\mcrCache`*`<version>`*	`%LOCALAPPDATA%\MathWorks\MatlabRuntimeCache\`*`<release_version>`*
Linux	`$HOME/.mcrCache`*`<version>`*	`$HOME/.MathWorks/MatlabRuntimeCache/`*`<release_version>`*
Mac	A cache is not used in MATLAB R2020a and later. The app bundle contains the files necessary for run time.	

Change Options Using Environment Variables

Using environment variables, you can specify the following options for the MATLAB Runtime component cache:

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

Use the following system environment variables to change these settings.

Environment Variable	Purpose	Notes
MCR_CACHE_ROOT	When set to the location of where you want the deployable archive to be extracted, this variable overrides the default per-user component cache location. This is true for embedded `.ctf` files only.	On macOS, this variable is ignored in MATLAB R2020a and later. The app bundle contains the files necessary for run time.

Environment Variable	Purpose	Notes
MCR_CACHE_SIZE	When set, this variable overrides the default component cache size.	The initial limit for this variable is 32M (megabytes), or 1024M starting in R2024b. 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.

Override Default Behavior

For targets besides Java, you can direct `mcc` to not embed the deployable archive in generated binaries using the `mcc -C` option. You can use this option to troubleshoot problems with the deployable archive, as the log and diagnostic messages are much more visible.

You can also implement this override by adding the `-c` flag in the **Settings** section of the compiler app.

If you are using a `compiler.build` function, use the `EmbedArchive=false` option.

Note If you run `mcc` specifying conflicting wrapper and target types, the archive will not be embedded into the generated component, as if you had specified the `-C` option to `mcc`. For example:

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

See Also

Related Examples

- “About Deployable Archives” on page 5-5
- “About MATLAB Runtime” on page 7-2
- “Define Embedding and Extraction Options for Deployable Java Archive” (MATLAB Compiler SDK)

Write Deployable MATLAB Code

In this section...

“Accepted File Types for Packaging” on page 5-10
“Packaged Applications Do Not Process MATLAB Files at Run Time” on page 5-10
“Get Proper Licenses for Toolbox Functionality You Want to Deploy” on page 5-11
“Use isdeployed Functions to Execute Deployment-Specific Code Paths” on page 5-11
“Do Not Rely on Changing Directory or Path to Control the Execution of MATLAB Files” on page 5-11
“Gradually Refactor Applications That Depend on Non-Deployable Functions” on page 5-12
“Do Not Create or Use Nonconstant Static State Variables” on page 5-12

In order to package and deploy MATLAB code, your code must follow certain guidelines to avoid errors.

Accepted File Types for Packaging

The valid and invalid file types for packaging using deployment apps are as follows.

Target Application	Valid File Types	Invalid File Types
Standalone applications	MATLAB MEX files, MATLAB scripts, MATLAB functions, and MATLAB class files. These files must have a single entry point.	Protected function files (`.p` files), Java functions, COM or .NET components, and data files.
C/C++ shared libraries, .NET assemblies, Java classes, Python packages, and COM components	MATLAB MEX files, MATLAB functions, and MATLAB class files. These files must have a single entry point.	MATLAB scripts, protected function files (`.p` files), Java functions, COM or .NET components, and data files.
MATLAB Production Server archives	MATLAB MEX files and MATLAB functions. These files must have a single entry point.	MATLAB scripts, MATLAB class files, protected function files (`.p` files), Java functions, COM or .NET components, and data files. MATLAB class files can be dependent files.

You can add other types of files to the packaged code archive, such as data files. For more information, see “Include and Access Files in Packaged Applications” on page 5-15.

Packaged Applications Do Not Process MATLAB Files at Run Time

The compiler secures your code against unauthorized changes. Deployable MATLAB files are suspended or frozen at the time of compilation. 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 deployable archive.

MATLAB Runtime only works on MATLAB code that was encrypted when the deployable archive was built. Any function or process that dynamically generates new MATLAB code will not work against MATLAB Runtime.

Some MATLAB toolboxes, such as the Deep Learning Toolbox™ product, generate MATLAB code dynamically. Because MATLAB Runtime only executes encrypted MATLAB files, and the Deep Learning Toolbox generates unencrypted MATLAB files, some functions in the Deep Learning 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. Package the MATLAB code and include 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.

Get Proper Licenses for Toolbox Functionality You Want to Deploy

You must have a valid MathWorks license for toolboxes you use to create deployable MATLAB code. Your end users do not require any licenses to run packaged toolbox code.

Use 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 packaged and executed.

```
if ~(ismcc || isdeployed)
    addpath(mypath);
end
```

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 attempt 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`. For details, see “Use isdeployed Functions to Execute Deployment-Specific Code Paths” on page 5-11.

Gradually Refactor Applications That Depend on Non-Deployable Functions

Over time, refactor, streamline, and modularize MATLAB code containing non-compilable or non-deployable functions that use `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 Deep Learning 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 non-deployable 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 MATLAB Runtime 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 against packaged MATLAB code, you should be aware that an instance of MATLAB Runtime 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 MATLAB Runtime instance 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 MATLAB Runtime 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`.

See Also

`isdeployed` | `ismcc`

More About

- MATLAB Compiler support for MATLAB and toolboxes
- “Limitations” on page 13-2

Calling Shared Libraries in Deployed Applications

The `loadlibrary` function in MATLAB allows you to load shared library into MATLAB.

Loading libraries using header files is not supported in compiled applications. Therefore, to create an application that uses the `loadlibrary` function with a header file, follow these steps:

1. Create a prototype MATLAB file. Suppose that you call `loadlibrary` with the following syntax.

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

   Run the following command in MATLAB only once to create the prototype file:

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

   This creates *`mylibrarymfile`*`.m` in the current folder. If you are on Windows, another file named `library_thunk_pcwin64.dll` is also created in the current folder.
2. Change the call to `loadlibrary` in your MATLAB to the following:

   ```
   loadlibrary(library, @mylibrarymfile)
   ```
3. Compile and deploy the application.

   - If you are integrating the library into a deployed application, specify the library's `.dll` along with `library_thunk_pcwin64.dll`, if created, using the `-a` option of `mcc` command. If you are using a compiler app, add the `.dll` files to the **Files required for your application to run** section of the app.
   - If you are providing the library as an external file that is not integrated with the deployed application, place the library `.dll` file in the same folder as the compiled application. If you are on Windows, you must integrate `library_thunk_pcwin64.dll` into your compiled application.

     The benefit of this approach is that you can replace the library with an updated version without recompiling the deployed application. Replacing the library with a different version works only if the function signatures of the function in the library are not altered. This is because *`mylibrarymfile`*`.m` and `library_thunk_pcwin64.dll` are tied to the function signatures of the functions in the library.

Note You cannot use `loadlibrary` inside MATLAB to load a shared library built with MATLAB. For more information on `loadlibrary`, see "Limitations to Shared Library Support".

Note Operating systems have a `loadlibrary` function, which loads specified Windows operating system module into the address space of the calling process.

See Also

`loadlibrary`

Related Examples

- "Call Functions in C Library Loaded with loadlibrary"

Include and Access Files in Packaged Applications

In this section...

"Include Files in Deployable Archive" on page 5-15
"Access Files from Deployed Functions" on page 5-16
"Example Processing MATLAB Data for Deployed Applications" on page 5-16

In addition to MATLAB script files, you can add other types of files to deployable archives such as data files, DLLs, and files from other programming languages. Access the additional files from your deployed code by using the `which` function or referencing the file location relative to the deployable archive root `ctfroot`.

For more information about deployable archives, see "About Deployable Archives" on page 5-5.

Include Files in Deployable Archive

MATLAB Compiler uses a dependency analysis function to determine the list of necessary files to include in the generated package. For details, see "Dependency Analysis Using MATLAB Compiler" on page 5-2.

You can include additional files in the deployable archive using the `-a` flag with the `mcc` command or the `'AdditionalFiles'` option using a `compiler.build` function, such as `compiler.build.standaloneApplication`.

Alternatively, you can add files to the **Custom Requirements** section in a compiler app.

Explicitly Include MATLAB Data Files Using %#function Pragma

The compiler excludes MATLAB data files (MAT files) during dependency analysis by default. You can include data files by adding them manually.

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

For example, if you want to include a dependency on the `ClassificationSVM` class loaded from a MAT file, use the `%#function` pragma.

```
function foo
    %#function ClassificationSVM
        load('svm-classifier.mat');
        num_dimensions = size(svm_model.PredictorNames, 2);
end %function foo
```

Include MEX Files, DLLs, or Shared Libraries

When you compile MATLAB functions containing MEX files, ensure that the dependency analysis process can find them. In particular, note that:

- Since the dependency analysis function cannot examine MEX files, DLLs, or shared libraries to determine their dependencies, explicitly include all executable files these files require.
- If you have any doubts that a MATLAB function called by a MEX file, DLL, or shared library can be found during dependency analysis, then manually include that function.

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

Access Files from Deployed Functions

To access files from your deployed MATLAB code, check if the code is running in deployed mode using `isdeployed`. Then, locate the file either by using the `which` function or by specifying the file location relative to `ctfroot`.

Use which function

The simplest way to obtain the path to a file is to locate the file by using the `which` function.

```
if isdeployed
    locate_externapp = which('extern_app.exe');
end
```

The `which` function returns the path to the file `extern_app.exe` if it is located within the deployable archive.

Specify File Location in ctfroot

When you include files that are in a folder other than the current MATLAB working folder, the partial file path is preserved in the deployable archive relative to `ctfroot`.

- Files within the current MATLAB working folder or subfolders retain the relative path from the current folder to the file.

  For example, if the folder open in MATLAB during packaging is `D:\Documents\Work\MyProj`, then the file `D:\Documents\Work\MyProj\exfiles\data1.mat` will be located at *`ctfroot`*`\`*`mfilename`*`\exfiles\data1.mat` in the deployable archive, where *`mfilename`* is the name of the main MATLAB script file.
- Files outside of the current folder retain the full folder structure from the root of the disk drive.

  For example, the file `C:\Users\mwuser\Documents\External\externdata\extern_app.exe` will be located at *`ctfroot`*`\Users\mwuser\Documents\External\externdata\extern_app.exe` in the deployable archive.

Use the `fullfile` function to ensure that file paths use the correct file separators for your system.

```
if isdeployed
    locate_data1 = fullfile(ctfroot,'exfiles','data1.mat');
    locate_data2 = fullfile(ctfroot,'Users','mwuser','Documents',...
        'External','externdata','extern_app.exe');
end
```

Example Processing MATLAB Data for Deployed Applications

This example shows how to include data files in a packaged application and use the `load` and `save` functions to manipulate MATLAB data.

1 Navigate to your work folder in MATLAB. For this example, the work folder is `C:\Users\mwuser\Documents\Work\exfiles`.

2 Copy the `Data_Handling` and `externdata` folders that ship with MATLAB to your work folder.

```
copyfile(fullfile(matlabroot,'extern','examples','compiler','Data_Handling'),'Data_Handling')
copyfile(fullfile(matlabroot,'extern','examples','compiler','externdata'),'externdata');
```

At the MATLAB command prompt, navigate into the new `Data_Handling` folder in your work folder.

3 Examine `ex_loadsave.m`.

The `ex_loadsave` script loads three MATLAB data files, each located in a different folder:

- `user_data.mat` — In the current folder
- `userdata\extra_data.mat` — In a subfolder of the current folder
- `..\externdata\extern_data.mat` — Outside of the current folder

ex_loadsave.m

```
function ex_loadsave
% This example shows how to work with the "load/save" functions
% on data files in deployed mode. This example contains three
% source data files.
%    user_data.mat
%    userdata/extra_data.mat
%    ../externdata/extern_data.mat
%
% Compile this example with mcc command:
%     mcc -m ex_loadsave.m -a 'user_data.mat'
%         -a './userdata/extra_data.mat'
%         -a '../externdata/extern_data.mat'
%
% In this example, the output data file is written to the path:
%    output/saved_data.mat
% relative to the application's run time current working directory.
% When writing data files to local disk, do not save any files under $ctfroot,
% as $ctfroot may be deleted and/or refreshed for each application run.
%
%==== load data file =============================
if isdeployed
    % In deployed mode, all files in or under the main file's directory
    % may be loaded by full path, by path relative to ctfroot, or by
    % filename only, since their directories are on the MATLAB path.
    % Here we load 'user_data.mat' by full path.
    LOADFILENAME1=which(fullfile(ctfroot,mfilename,'user_data.mat'));
    % These alternate methods also work:
    % LOADFILENAME1=which(fullfile(mfilename,'user_data.mat'));
    % LOADFILENAME1=which(fullfile('user_data.mat'));

    % Here we load 'extra_data.mat' by relative path:
    LOADFILENAME2=which(fullfile('userdata','extra_data.mat'));
    % These alternate methods also work:
    % LOADFILENAME2=which(fullfile(ctfroot,'userdata','extra_data.mat'));
    % LOADFILENAME2=which(fullfile('extra_data.mat'));

    % For a data file external to the main MATLAB file's directory tree,
    % its full compile time path is appended to $ctfroot, so it is
    % best to simply load it by filename (since it is on the path):
    LOADFILENAME3=which(fullfile('extern_data.mat'));
else
    %running the code in MATLAB
    LOADFILENAME1=fullfile(pwd,'user_data.mat');
    LOADFILENAME2=fullfile(pwd,'userdata','extra_data.mat');
    LOADFILENAME3=fullfile(pwd,'..','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 subdirectory
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);

%==== multiply two data matrices together ==============
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');
```

4 Create a cell array that lists the data files.

```
datafiles = {'user_data.mat','./userdata/extra_data.mat','../externdata/extern_data.mat'};
```

5 Compile `ex_loadsave.m` using the `compiler.build.standaloneApplication` function.

```
compiler.build.standaloneApplication('ex_loadsave.m','AdditionalFiles',datafiles)
```

6 Run the compiled application.

```
!ex_loadsavestandaloneApplication\ex_loadsave.exe
```

```
Load A from : C:\Users\mwuser\AppData\Local\Temp\mwuser\mcrCache9.13\ex_loa0\ex_loadsave\user_data.mat
A=
   21.4669   15.7255   15.6930   11.8122
   19.6691   17.0570   17.4689   22.2803
   20.3894   17.2548   17.3474   17.7316
   19.3062   15.1321   16.0573   25.4584

Load B from : C:\Users\mwuser\AppData\Local\Temp\mwuser\mcrCache9.13\ex_loa0\ex_loadsave\userdata\extra_data.mat
B=
   15.3970   20.5682   13.8388   26.5186
   14.2255   24.6506   18.9545   24.8117
   14.9904   22.8211   16.4942   25.3533
   13.1022   26.0567   21.2197   24.8940

Load extern data from : C:\Users\mwuser\AppData\Local\Temp\mwuser\mcrCache9.13\ex_loa0\Users\mwuser\Documents\Work\e
ext_data=
   27.6923   69.4829   43.8744   18.6873
    4.6171   31.7099   38.1558   48.9764
    9.7132   95.0222   76.5517   44.5586
   82.3458    3.4446   79.5200   64.6313

A * B =
   1.0e+03 *

    0.9442    1.4951    1.1046    1.6514
    1.0993    1.8042    1.3564    1.9424
    1.0518    1.7026    1.2716    1.8500
    1.0868    1.7999    1.3591    1.9283

Save the A * B result to : C:\Users\mwuser\Documents\Work\exfiles\Data_Handling\output\saved_data.mat
```

7 Compare the results to the output of `ex_loadsave.m`.

See Also

`ctfroot` | `which`

Related Examples

- "About Deployable Archives" on page 5-5
- "Dependency Analysis Using MATLAB Compiler" on page 5-2

Standalone Application Creation

Dependency Analysis Function and User Interaction with the Compilation Path

addpath and rmpath in MATLAB

MATLAB Compiler uses the MATLAB search path to analyze dependencies. See `addpath`, `rmpath`, `savepath` for information on working with the search path.

Note `mcc` does not use the MATLAB startup folder and will not find any path information saved there.

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.

- “About MATLAB Runtime” on page 7-2
- “Download and Install MATLAB Runtime” on page 7-4
- “Deploy Applications and MATLAB Runtime on Network Drives” on page 7-10
- “MATLAB Runtime on Big Data Platforms” on page 7-12
- “Install Deployed Application” on page 7-14

About MATLAB Runtime

In this section...

"Differences Between MATLAB Runtime and MATLAB" on page 7-2
"Size and Performance Considerations for MATLAB Runtime" on page 7-2

MATLAB Runtime (MCR) is a freely-available set of shared libraries, MATLAB code, and other files that enables the execution of MATLAB files on computers without an installed version of MATLAB. Applications that use artifacts built with MATLAB Compiler or MATLAB Compiler SDK require access to a matching version of MATLAB Runtime at the same update level or newer to run.

End users of compiled artifacts without access to MATLAB must install MATLAB Runtime. For details, see "Download and Install MATLAB Runtime" on page 7-4. For information on using a network installation of MATLAB Runtime, see "Deploy Applications and MATLAB Runtime on Network Drives" on page 7-10.

To download a Docker image that contains MATLAB Runtime, see "MATLAB Runtime Container" on page 2-2.

The installers generated by the compiler apps or functions such as `compiler.package.installer` may include the MATLAB Runtime installer.

After you install MATLAB Runtime, you may need to add the directories to the system library path depending on your operating system and shell environment. For details, see "Set MATLAB Runtime Path for Deployment" on page 15-2.

Differences Between MATLAB Runtime and MATLAB

The MATLAB Runtime differs from MATLAB in several important ways:

- MATLAB Runtime is freely available to download on the MathWorks website. MATLAB requires an active license to use.
- In MATLAB Runtime, MATLAB files are encrypted and immutable.
- MATLAB has a desktop graphical interface. MATLAB Runtime has all the MATLAB functionality without the graphical interface.
- MATLAB Runtime is version-specific. You must run your applications with the version of MATLAB Runtime associated with the version of MATLAB Compiler with which it was created. For example, if you compiled an application using release R2023a of MATLAB, end users must install version R2023a at the same update level or later of MATLAB Runtime. Use `mcrversion` to return the version number of MATLAB Runtime.
- The MATLAB paths in a MATLAB Runtime instance are fixed and cannot be changed. To change them, you must first customize them within MATLAB.

Size and Performance Considerations for MATLAB Runtime

MATLAB Compiler SDK 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 MATLAB Runtime 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 MATLAB Runtime is necessary in order to retain the power and functionality of a full version of MATLAB.

Calls into MATLAB Runtime are serialized, so calls into MATLAB Runtime are thread-safe. This can impact performance.

Reduce MATLAB Runtime Size

Starting in R2024b, you can reduce the size of a MATLAB Runtime installation. Create a custom MATLAB Runtime installer with a minimal size footprint using the `compiler.runtime.customInstaller` function.

Additionally, GPU libraries are no longer required dependencies for MATLAB Runtime. Packaged MATLAB code that does not display graphical output can be run using a MATLAB Runtime installation without GPU libraries. You can use MATLAB Runtime without GPU support by using the `compiler.runtime.customInstaller` function or the MATLAB Runtime Docker image hosted on the MathWorks Docker repository. For more information, see “MATLAB Runtime Container” on page 2-2.

See Also
`mcrversion` | `compiler.runtime.download`

Related Examples
- “Download and Install MATLAB Runtime” on page 7-4
- “Set MATLAB Runtime Path for Deployment” on page 15-2
- “MATLAB Runtime Container” on page 2-2
- “Deploy Applications and MATLAB Runtime on Network Drives” on page 7-10

Download and Install MATLAB Runtime

Supported Platforms: Windows, Linux, macOS

MATLAB Runtime contains the libraries needed to run compiled MATLAB applications on a target system without a licensed copy of MATLAB. For more information, see About MATLAB Runtime on page 7-2.

Download MATLAB Runtime Installer

Download the MATLAB Runtime installer using one of the following options:

- Download the MATLAB Runtime installer at the latest update level for the selected release from the website at https://www.mathworks.com/products/compiler/matlab-runtime.html. This option is best for end users who want to run deployed applications.
- Use the MATLAB function `compiler.runtime.download` to download the MATLAB Runtime installer matching the version and update level of MATLAB from where the command is executed. If the installer has already been downloaded to the machine, it returns the path to the MATLAB Runtime installer. If the machine is offline, it returns a URL to the MATLAB Runtime installer. This option is best for developers who want to create application installers that contain MATLAB Runtime.
- Use the MATLAB function `compiler.runtime.customInstaller` to create a MATLAB Runtime installer that installs only the MATLAB Runtime components required to run the specified artifacts created with MATLAB Compiler or MATLAB Compiler SDK. This option is best for developers who require an installer with a minimal size footprint that can only run specific applications.

Note If you want to install MATLAB Runtime to a shared network drive, see “Deploy Applications and MATLAB Runtime on Network Drives” on page 7-10.

To download a Docker image that contains MATLAB Runtime, see “MATLAB Runtime Container” on page 2-2.

Install MATLAB Runtime Interactively

To install MATLAB Runtime:

1. Extract the archive containing the MATLAB Runtime installer. The release part of the installer file name (for instance, *`_R2025b_`*) changes from one release to the next.

Platform	Steps
Windows	Unzip the MATLAB Runtime installer. Right-click the ZIP file `MATLAB_Runtime_R2025b_win64.zip` and select **Extract All**.

Platform	**Steps**
Linux	Unzip the MATLAB Runtime installer at the terminal using the `unzip` command. For example, if you are unzipping the R2025b MATLAB Runtime installer, at the terminal, type: `unzip MATLAB_Runtime_R2025b_glnxa64.zip`

macOS	Unzip the MATLAB Runtime installer at the terminal using the `unzip` command.
	
	For Intel® processor-based macOS, type:
	
	`unzip MATLAB_Runtime_R2025b_maci64.zip`
	
	For Apple silicon-based macOS, type:
	
	`unzip MATLAB_Runtime_R2025b_maca64.zip`

2 Start the MATLAB Runtime installer.

Platform	**Steps**
Windows	Double-click the file `setup.exe` from the extracted files to start the installer.

Linux	At the terminal, type:
	
	`sudo -H ./install`
	
	`sudo` is only required if you install to a directory that you do not have write access to.
	
	Note You may need to allow the root user to access the running X server:
	
	`xhost +SI:localuser:root`
	`sudo -H ./install`
	`xhost -SI:localuser:root`

macOS	Double-click the DMG file to start the installer.

3 When the MATLAB Runtime installer starts, it displays a dialog box. Read the information and then click **Next** to proceed with the installation.

4 In the **Folder Selection** dialog box, specify the folder where you want to install MATLAB Runtime.

You can have multiple versions of MATLAB Runtime on your computer, but only one installation for any particular version. If you have an existing installation of the same version, the MATLAB Runtime installer does not display the **Folder Selection** dialog box because it overwrites the existing installation in the same folder.

Caution The installation path must not contain any non-ASCII characters.

5 Confirm your choices and click **Next**.

The MATLAB Runtime installer starts copying files into the installation folder.

6. On Linux and macOS platforms, after copying files to your disk, the MATLAB Runtime 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, save it to a text file, and then click **Next**.

   For instructions on setting the path environment variables, see “Set MATLAB Runtime Path for Deployment” on page 15-2.
7. Click **Finish** to exit the installer.

Default Install Folder

The default MATLAB Runtime installation folders for R2025b are specified in the following table:

Platform	MATLAB Runtime Installation Folder
Windows	`C:\Program Files\MATLAB\MATLAB Runtime\R2025b`
Linux	`/usr/local/MATLAB/MATLAB_Runtime/R2025b`
macOS	`/Applications/MATLAB/MATLAB_Runtime/R2025b`

Install MATLAB Runtime Noninteractively

Supported platforms: Windows, Linux

If you have many installations to perform, you can specify installation arguments as command-line arguments or in an installer control file to save time and prevent errors. When you specify installation arguments, the MATLAB Runtime installer runs as a background task and does not display any dialog boxes.

When running noninteractively, the installer overwrites the installation location.

Caution On Linux, the installer displays information necessary for setting your environment variables in the **Product Configuration Notes** dialog box. If you use the installer noninteractively, you must locate your MATLAB Runtime installation directory in order to set the library path after installation. For more information, see “Set MATLAB Runtime Path for Deployment” on page 15-2.

Run Installer in Silent Mode

To install MATLAB Runtime in silent mode:

1. Extract the contents of the MATLAB Runtime installer archive to a temporary folder.
2. In your system command prompt, navigate to the folder where you extracted the installer.
3. Run the MATLAB Runtime installer, specifying the `-agreeToLicense yes` option on the command line. If you do not include `-agreeToLicense yes` as the first option, the installer will not install MATLAB Runtime.

   Note On most platforms, the installer is located at the root of the folder into which the archive was extracted. On 64-bit Windows, the installer is located in the archive `bin` folder.

Platform	Command
Windows	`setup -agreeToLicense yes`
Linux	`sudo ./install -agreeToLicense yes` **Note** `sudo` is only required if you install to a directory you do not have write access to.

Note To install MATLAB Runtime R2022a and earlier, you must also specify `-mode silent` in the command to run the installer in silent mode.

4 View a log of the installation.

- On Windows systems, the installer creates a log file named `mathworks_`*`username`*`.log`, where *`username`* is your Windows login name, in the location defined by your `TEMP` environment variable.
- On Linux, the installer displays the log information at the command prompt. It also saves it to a file if you use the `-outputFile` option.

Customize Noninteractive Installation

When run noninteractively, the installer uses the default values unless you specify otherwise. Like the MATLAB installer, the MATLAB Runtime installer accepts a number of options that modify the default installation properties.

Create an installer control text file that contains your command-line options and values. Put each option and value pair on a separate line. For example:

```
agreeToLicense=yes
destinationFolder=/usr/MATLAB/MATLAB_Runtime
outputFile=myapp_log.txt
```

Then, specify the file using the `-inputfile` argument at the command line. For example, on Linux:

```
./install -inputfile installer_input.txt
```

You can specify the following options in the installer input file.

Option	Description
`agreeToLicense`	Agree to the MATLAB Runtime license.
`destinationFolder`	Specifies where MATLAB Runtime is installed. If the user does not have write access to the folder, you must run the installer with administrator privileges.
`outputFile`	Specifies where the installation log file is written.

Note The MATLAB installer archive includes an example installer control file called `installer_input.txt`, which contains all of the options available for a full MATLAB installation. However, the MATLAB Runtime installer only accepts the options listed in this section.

Install MATLAB Runtime Without Administrator Rights

On Linux, to install MATLAB Runtime without `sudo` privileges, select a folder that you have write access to during installation.

On Windows, to install MATLAB Runtime as a user without administrator rights:

1. Install MATLAB Runtime on a Windows machine where you have administrator rights.
2. Copy the folder where MATLAB Runtime was installed to the machine without administrator rights. You can compress the folder into a zip file for distribution.
3. On the machine without administrator rights, add the *`<MATLAB_RUNTIME_INSTALL_DIR>`* `\runtime\`*`arch`* directory to the user's `PATH` environment variable. For more information, see "Set MATLAB Runtime Path for Deployment" on page 15-2.

Install Multiple MATLAB Runtime Versions on Single Machine

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

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

Note Installing multiple versions of MATLAB Runtime on the same machine is not supported on macOS.

MATLAB and MATLAB Runtime on Same Machine

To test your deployed component on your development machine, you do not need an installation of MATLAB Runtime. The MATLAB installation that you use to compile the component can act as a MATLAB Runtime replacement.

You can install MATLAB Runtime for debugging purposes.

Modify Path

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

To run deployed MATLAB code against MATLAB Runtime rather than MATLAB, ensure that your library path lists the MATLAB Runtime directories before any MATLAB directories. For information on setting environment variables, see "Set MATLAB Runtime Path for Deployment" on page 15-2.

Alternatively, you can specify the location of MATLAB Runtime using the generated shell script for your compiled application.

Uninstall MATLAB Runtime

Windows

1. Start the uninstaller.

   From the Windows Start menu, search for the **Add or Remove Programs** control panel, and double-click MATLAB Runtime in the list.

   You can also start the MATLAB Runtime uninstaller from the `<MATLAB_RUNTIME_INSTALL_DIR>\bin\<arch>` folder, where `<MATLAB_RUNTIME_INSTALL_DIR>` is your MATLAB Runtime installation folder and `<arch>` is an architecture-specific folder, such as `win32` or `win64`.
2. Select MATLAB Runtime from the list of products in the Uninstall Products dialog box and click **Next**.
3. Click **Finish**.

Linux

1. Close all instances of MATLAB and MATLAB Runtime.
2. Enter this command at the Linux terminal:

   ```
   rm -rf <MATLAB_RUNTIME_INSTALL_DIR>
   ```

   Caution Be careful when using the `rm` command, as deleted files cannot be recovered.

macOS

1. Close all instances of MATLAB and MATLAB Runtime.
2. Navigate to your MATLAB Runtime installation folder. For example, the installation folder might be named `MATLAB_Compiler_Runtime.app` in your Applications folder.
3. Drag your MATLAB Runtime installation folder to the trash, and then select **Empty Trash** from the Finder menu.

See Also

`compiler.runtime.download`

More About

- About MATLAB Runtime on page 7-2
- “MATLAB Runtime Startup Options” on page 8-2
- “Set MATLAB Runtime Path for Deployment” on page 15-2
- “MATLAB Runtime Container” on page 2-2

Deploy Applications and MATLAB Runtime on Network Drives

You can deploy a MATLAB Compiler or MATLAB Compiler SDK generated application to a network drive so that it can be accessed by all network users. You can also install MATLAB Runtime onto a network drive so that local clients do not need to install MATLAB Runtime on their individual machines.

Install MATLAB Runtime on Network Drives

On Linux systems, distributing to a network file system is the same as distributing to a local file system. After installing MATLAB Runtime on the network drive, set the `LD_LIBRARY_PATH` environment variable on all client machines or use shell scripts that point to the MATLAB Runtime installation. For details, see “Set MATLAB Runtime Path for Deployment” on page 15-2.

On Windows systems, complete the following steps to use a network install of MATLAB Runtime.

1. Install MATLAB Runtime onto a machine with the same system architecture as the network drive. For details, see “Download and Install MATLAB Runtime” on page 7-4.
2. Copy the entire MATLAB Runtime installation folder onto the network drive.

   Caution The installation path must not contain any non-ASCII characters.
3. Add the directory `<MATLAB_RUNTIME_INSTALL_DIR>\<VERSION>\runtime\<ARCH>` to the path on all client machines. For details, see “Set MATLAB Runtime Path for Deployment” on page 15-2. All network clients can then execute compiled applications.

Deploy Applications on Network Drives

On Linux systems and for deployment targets other than Excel or COM, distributing to a network file system is the same as distributing to a local file system.

On Windows systems, in order to run Excel add-ins and COM components, you must register the `MWComUtil` utility library on client machines.

The following table specifies which DLLs to register on each client machine to deploy specific applications.

Application Deployed	DLLs to Register

Excel Add-Ins	`mwcomutil.dll`
	
	`mwcommgr.dll`

COM Components	`mwcomutil.dll`

To register these DLLs:

1. Open a system command prompt
2. Navigate to *`matlabroot`*`\bin\`*`version`*, where *`matlabroot`* represents the location of MATLAB or MATLAB Runtime that corresponds to the MATLAB release that you used to compile your application.
3. Run one or both of the following commands:

```
mwregsvr mwcomutil.dll
```

```
mwregsvr mwcommgr.dll
```

See Also

Related Examples

- “Download and Install MATLAB Runtime” on page 7-4
- “Set MATLAB Runtime Path for Deployment” on page 15-2
- “Distribute Add-Ins and Integrate into Microsoft Excel”
- “Register COM Component” (MATLAB Compiler SDK)

MATLAB Runtime on Big Data Platforms

MATLAB Runtime can be downloaded and installed on big data platforms such as CLOUDERA®, Apache® Ambari™, and Azure® HDInsight.

CLOUDERA

MATLAB Runtime can be downloaded as a parcel from within CLOUDERA Manager.

Download URL

```
https://ssd.mathworks.com/supportfiles/downloads/R2025b/deployment_files/
R2025b/cdhparcels
```

After downloading the parcel, you can and distribute and activate it across the cluster. For more information on how to work with CLOUDERA Manager and parcels, see the CLOUDERA documentation.

Apache Ambari

Warning MATLAB Runtime support for Apache Ambari will be removed in a future release.

You can download MATLAB Runtime as an Apache Ambari stack for distribution across a cluster using the following URLs:

Download URLs

```
https://ssd.mathworks.com/supportfiles/downloads/R2025b/deployment_files/
R2025b/ambari/matlab-runtime-2025b-service.tgz
```

```
https://ssd.mathworks.com/supportfiles/downloads/R2025b/deployment_files/
R2025b/ambari/matlab-runtime-2025b-service.sha1
```

For more information, see the Apache Ambari documentation.

Azure HDInsight

You can download MATLAB Runtime onto an Azure HDInsight cluster using the following URLs:

Download URLs

```
https://ssd.mathworks.com/supportfiles/downloads/R2025b/deployment_files/
R2025b/hdinsight/runtime_install_R2025b_hdinsight.sh
```

```
https://ssd.mathworks.com/supportfiles/downloads/R2025b/deployment_files/
R2025b/hdinsight/matlab-runtime-2025b-glnxa64.tgz
```

Use the script action URL within the Azure interface to download and deploy MATLAB Runtime across the cluster.

See Also

External Websites

- Customize Azure HDInsight clusters by using script actions

Install Deployed Application

After you create an installer for your compiled application, you can install it interactively using a graphical interface or noninteractively using command-line arguments.

Install Application Interactively

Complete the following steps according to your operating system to install the application `my_app` interactively using `MyAppInstaller`.

1. Start the installer.

Platform	Steps
Windows	Double-click the file `MyAppInstaller.exe`.

Linux	In the terminal, type:
	`sudo -H ./MyAppInstaller.install`
	Note You may need to allow the root user to access the running X server:
	`xhost +SI:localuser:root`
	`sudo -H ./install`
	`xhost -SI:localuser:root`
	`sudo` is only required if you install to a directory that you do not have write access to.

macOS	Double-click the `MyAppInstaller` file to start the installer. Alternatively, in the terminal, type: `./MyAppInstaller` **Note** You may need to enter an administrator username and password after you run `MyAppInstaller`.

2. If you want to connect to the Internet using a proxy server, click **Connection Settings**. Enter the proxy server settings in the provided window and click **OK**. Click **Next**.
3. Choose the installation folder for the application. To create a desktop shortcut, check the box labeled **Add a shortcut to the desktop**. Click **Next**.
4. If MATLAB Runtime is not already installed on your machine, choose the installation folder for the MATLAB Runtime libraries and click **Next**.
5. Select **Yes** to accept the terms of the MATLAB Runtime license agreement and click **Next**.
6. Click **Install >** to begin the installation.
7. On Linux and macOS platforms, after copying files to your disk, the 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, save it to a text file, and then click **Next**. For information on setting environment variables, see “Set MATLAB Runtime Path for Deployment” on page 15-2.
8. Click **Finish** to exit the installer.

9. If you accepted the default settings, you can find the installed application in one of the following locations:

Windows	`C:\Program Files\my_app`
macOS	`/Applications/my_app`
Linux	`/usr/my_app`

Install Application Noninteractively

If you have many installations to perform, you can specify installation arguments as command-line arguments or in an installer control file to save time and prevent errors. When you specify installation arguments, the installer runs as a background task and does not display any dialog boxes.

When running noninteractively, the installer overwrites the installation location.

Caution On Linux and macOS systems, the installer displays information necessary for setting your environment variables in the **Product Configuration Notes** dialog box. If you use the installer noninteractively, you must locate your MATLAB Runtime installation directory in order to set the library path after installation. For more information, see “Set MATLAB Runtime Path for Deployment” on page 15-2.

To install the application in noninteractively:

1. Run the installer on the command line and specify the option `-agreeToLicense yes`. If you do not include `-agreeToLicense yes` as the first option, the installer will not install the application.

Platform	**Command**
Windows	`MyAppInstaller.exe -agreeToLicense yes`
Linux	`sudo ./MyAppInstaller.install -agreeToLicense yes` **Note** `sudo` is only required if you install to a directory you do not have write access to.
macOS	`./MyAppInstaller -agreeToLicense yes`

2. View a log of the installation.

   On Windows systems, the installer creates a log file named `mathworks_`*`username`*`.log`, where *`username`* is your Windows login name, in the location defined by your `TEMP` environment variable. You can specify a log file using the `-outputFile` option.

   On Linux and macOS systems, the installer displays the log information at the command prompt. If you specify a file using the `-outputFile` option, it also saves the log information to the file.

Customize Noninteractive Installation

When run noninteractively, the installer uses the default values for installation options unless you specify otherwise. Like the MATLAB installer, the application installer accepts a number of command-line options that modify the default installation properties.

To specify options on the command line, separate each option and its value with a space. For example, on Linux:

```
./MyAppInstaller.install -agreeToLicense yes \
-outputFile myapp_log.txt -applicationFolder ~/Apps/magicsquare
```

Option	Description	Comment

`-inputFile`	Specifies an installer control file that contains your command-line options and values.	Omit the dash before each option and put each option and value pair on a separate line. For example:
		
		`agreeToLicense=yes`
		`startMenuShortcut=true`

`-applicationFolder`	Specifies where the application is installed.	Do not specify this option with the `-destinationFolder` option.
`-runtimeFolder`	Specifies where MATLAB Runtime is installed.	In the destination folder, MATLAB Runtime is installed in a folder named after the corresponding MATLAB release, for example, R2025b. Do not specify this option with the `-destinationFolder` option.
`-destinationFolder`	Specifies where both the application and MATLAB Runtime are installed.	In the destination folder, MATLAB Runtime is installed in a folder named after the corresponding MATLAB release, for example, R2023b.
`-outputFile`	Specifies where the installation log file is written.	On Windows, the log file is written to the location defined by your `TEMP` environment variable by default. On Linux and macOS, log information is displayed at the command prompt. If you specify a file using this option, it saves the log information to the file.

Option	Description	Comment
`-desktopShortcut true|false`	Specifies whether to create a desktop shortcut icon for the installed application.	This option must be specified in an installer control file provided by `-inputFile`. The default value is false. This option is only used on Windows.
`-startMenuShortcut true|false`	Specifies whether to create a Start Menu shortcut icon for the installed application.	This option must be specified in an installer control file provided by `-inputFile`. The default value is false. This option is only used on Windows.

See Also

More About

- “Create Standalone Application from MATLAB” on page 18-2
- “Download and Install MATLAB Runtime” on page 7-4
- “Set MATLAB Runtime Path for Deployment” on page 15-2

Work with the MATLAB Runtime

- “MATLAB Runtime Startup Options” on page 8-2
- “Using MATLAB Runtime User Data Interface” on page 8-3
- “Display MATLAB Runtime Initialization Messages” on page 8-5
- “Ensure MATLAB Runtime Compatibility for MATLAB Compiler and MATLAB Compiler SDK Artifacts” on page 8-6

MATLAB Runtime Startup Options

Set MATLAB Runtime Options

For a standalone executable, set MATLAB Runtime options by using the `mcc` command and specifying the `-R` flag and arguments. For example, specify a log file.

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

Not all options are available for all compilation targets. For full details, see `mcc -R`.

Set Multiple Options Using -R

You can specify multiple `-R` options. When you specify multiple `-R` options, they are processed from left to right. For example, specify initialization start and end messages.

```
mcc -R '-startmsg,MATLAB Runtime initialized' -R '-completemsg,Initialization complete'
```

See Also

`mcc`

Using MATLAB Runtime User Data Interface

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

- You need to supply MATLAB Runtime 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. For more information, see “Use Parallel Computing Toolbox in Deployed Applications” on page 3-6.
- You want to initialize MATLAB Runtime with constant values that can be accessed by all your MATLAB applications.
- 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.

MATLAB Functions

The API consists of two MATLAB functions callable from within deployed MATLAB code. Use the MATLAB functions `getmcruserdata` and `setmcruserdata` from deployed MATLAB applications. They are loaded by default only in applications created with MATLAB Compiler or MATLAB Compiler SDK.

Tip `getmcruserdata` and `setmcruserdata` produce an `Unknown function` error when called in MATLAB if the MCLMCR module cannot be located. You can avoid this situation by calling `isdeployed` before calling `getmcruserdata` and `setmcruserdata`. For more information, see `isdeployed`.

Set and Retrieve MATLAB Runtime Data for Shared Libraries

There are many possible scenarios for working with MATLAB Runtime data. The most general scenario involves setting the MATLAB Runtime with specific data for later retrieval, as follows:

1. In your code, include the MATLAB Runtime header file and the library header generated by MATLAB Compiler SDK.
2. Properly initialize your application using `mclInitializeApplication`.
3. After creating your input data, write or *set* it to the MATLAB Runtime with `setmcruserdata`.
4. After calling functions or performing other processing, retrieve the new MATLAB Runtime 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`.

See Also

`setmcruserdata` | `getmcruserdata`

More About

- “Use Parallel Computing Toolbox in Deployed Applications” on page 3-6
- “Specify Parallel Computing Toolbox Profile in .NET Application” (MATLAB Compiler SDK)
- “Specify Parallel Computing Toolbox Profile in Java Application” (MATLAB Compiler SDK)

Display MATLAB Runtime Initialization Messages

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

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

You have the following options:

- Use the default startup message only (`Initializing MATLAB runtime version` *`x.xx`*)
- Customize the startup or completion message with text of your choice. The default startup message will also display prior to displaying your customized startup message.

Some examples of different ways to invoke this option follow:

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

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

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

Ensure MATLAB Runtime Compatibility for MATLAB Compiler and MATLAB Compiler SDK Artifacts

Artifacts created with MATLAB Compiler and MATLAB Compiler SDK require the same version of the MATLAB Runtime as the version of MATLAB used to create them. The artifacts are tied to the major MATLAB release that created them but can run on any update level of the MATLAB Runtime. However, it is recommended you use a MATLAB Runtime version that matches or is later than the update level of the MATLAB release used for creation.

For example, let’s say you created an artifact using MATLAB R2022a. This artifact requires the MATLAB Runtime version corresponding to R2022a. While it can run on any update of MATLAB Runtime for R2022a such as R2022a Update 3, it’s recommended you use a MATLAB Runtime version that matches the update level of the MATLAB version used for creation. So, if you developed the artifact with R2022a Update 2, it’s best to use MATLAB Runtime R2022a Update 2 or later to ensure compatibility.

See Also

Related Examples

- “Download and Install MATLAB Runtime” on page 7-4

Distributing Code to an End User

Distribute MATLAB Code Using the MATLAB Runtime

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

MATLAB Runtime

MATLAB Runtime 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.

MATLAB Runtime is available to download from the web to simplify the distribution of your applications created using the MATLAB Compiler or the MATLAB Compiler SDK. Download the MATLAB Runtime from the MATLAB Runtime product page or use the `compiler.runtime.download` MATLAB function.

The MATLAB Runtime installer performs the following actions:

1. Install the MATLAB Runtime.
2. Install the component assembly in the folder from which the installer is run.
3. Copy the `MWArray` assembly to the Global Assembly Cache (GAC).

MATLAB Runtime Prerequisites

1. The MATLAB Runtime installer requires administrator privileges to run.
2. The version of MATLAB Runtime that runs your application on the target computer must be the same as the version of MATLAB Compiler or MATLAB Compiler SDK that built the deployed code, at the same update level or newer.
3. Do not install the MATLAB Runtime in MATLAB installation directories.
4. The MATLAB Runtime installer requires approximately 2 GB of disk space.

Add the MATLAB Runtime Installer to the Installer

This example shows how to include the MATLAB Runtime 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 or MATLAB Compiler SDK and properly lays them 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 downloads the MATLAB Runtime installer from the MathWorks website.
   - **Runtime included in package** — The option includes the MATLAB Runtime installer in the generated installer.
2. Click **Package**.
3. Distribute the installer to end users.

Install the MATLAB Runtime

For instructions on how to install the MATLAB Runtime on a system, see “Download and Install MATLAB Runtime” on page 7-4.

If you are given an installer containing the compiled artifacts, then MATLAB Runtime is installed along with the application or shared library. If you are given just the raw binary files, you must download and run the MATLAB Runtime installer.

Note On Windows, paths are set automatically by the installer. If you are running on a platform other than Windows, you must either modify the path on the target machine or use a shell script to launch the compiled application. Setting the paths enables your application executable to find MATLAB Runtime. For more information on setting the path, see “Set MATLAB Runtime Path for Deployment” on page 15-2.

Compiler Commands

This chapter describes `mcc`, which is the command that invokes the compiler.

Compiler Tips

In this section...
"Deploying Applications That Call the Java Native Libraries" on page 10-2
"Using the VER Function in a Compiled MATLAB Application" on page 10-2

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 MATLAB Runtime 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.

Using the VER Function in a Compiled MATLAB Application

When you use the VER function in a compiled MATLAB application, it will perform with the same functionality as if you had called it from MATLAB. However, be aware that when using VER in a compiled MATLAB application, only version information for toolboxes which the compiled application uses will be displayed.

Standalone Applications

This chapter describes how to use MATLAB Compiler to code and build standalone applications. You can distribute standalone applications to users who do not have MATLAB software on their systems.

Deploying Standalone Applications

In this section...

"Compiling the Application" on page 11-2
"Testing the Application" on page 11-2
"Deploying the Application" on page 11-3
"Running the Application" on page 11-4

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.

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 deployable 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 "Set MATLAB Runtime Path for Deployment" on page 15-2.
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 4    (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
MATLAB Runtime installer	Self-extracting MATLAB 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
MATLAB Runtime installer	MATLAB 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
MATLAB Runtime installer	MATLAB 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

Component	Description

`magicsquare.app`	Application bundle
	
	Assuming `foo` is a folder within your current folder:
	
	• Distribute by copying:
	`cp -R myapp.app foo`
	• Distribute by tarring:
	`tar -cvf myapp.tar myapp.app`
	`cd foo`
	`tar -xvf../ myapp.tar`
	• Distribute by zipping:
	`zip -ry myapp myapp.app`
	`cd foo`
	`unzip ..\myapp.zip`

Running the Application

These steps describe the process that end users must follow to install and run the application on their machines.

Preparing Your Machines

Install the MATLAB Runtime by running the `mcrinstaller` command to obtain name of the executable or binary. For more information on running the MATLAB Runtime installer utility and modifying your system paths, see “MATLAB Runtime” on page 9-2.

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.

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
```

Troubleshooting

- "Testing Failures" on page 12-2
- "Investigate Deployed Application Failures" on page 12-4

Testing Failures

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 MATLAB Runtime to be installed. A distribution includes all of the files that are required by your application to run, which include the executable, deployable archive, and MATLAB Runtime.

Test the application on the development machine by running the application against the MATLAB Runtime shipped with MATLAB Compiler. This will verify that library dependencies are correct, that the deployable 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 following questions 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.

Occasionally, there is a version mismatch between a DLL included with both MATLAB Runtime and Microsoft Windows. You can investigate which DLLs are called by your application using the Process Monitor tool. For information on using Process Monitor with your deployed application, see How can I use Process Monitor to troubleshoot the execution of my program?.

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\win64` of the version of MATLAB in which you are compiling appears ahead of *`matlabroot`*`\runtime\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 MATLAB Runtime?

All shared libraries required for your standalone executable or shared library are contained in MATLAB Runtime. Installing MATLAB Runtime 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 `mclmcrrt`*`X_XX`*`.dll` or `mclmcrrt`*`X_XX`*`.so` are generally caused by an incorrect installation of MATLAB Runtime. For information on installing MATLAB Runtime, see “Download and Install MATLAB Runtime” on page 7-4.

It is also possible that MATLAB Runtime is installed correctly, but the `PATH`,`LD_LIBRARY_PATH`, or `DYLD_LIBRARY_PATH` variable is set incorrectly. For information on setting environment variables, see “Set MATLAB Runtime Path for Deployment” on page 15-2.

Caution Do not solve these problems by moving libraries or other files within the MATLAB Runtime folder structure. The MATLAB Runtime system is designed to accommodate different MATLAB Runtime versions operating on the same machine. The folder structure is an important part of this feature.

Investigate Deployed Application Failures

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 MATLAB Runtime on their machines. MATLAB Runtime includes a set of shared libraries that provides support for all features of MATLAB.

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 5-10

Install MATLAB Runtime

All shared libraries required for your standalone executable or shared library are contained in MATLAB Runtime. Installing MATLAB Runtime is required for all deployment targets. For more information, see "Download and Install MATLAB Runtime" on page 7-4.

Update Dynamic Library Path on Linux or macOS

For information on setting the path on a deployment machine after installing MATLAB Runtime, see "Set MATLAB Runtime Path for Deployment" on page 15-2.

Error Message for Missing DLL

Error messages indicating missing DLLs such as `mclmcrrt`*`X_XX`*`.dll` or `mclmcrrt`*`X_XX`*`.so` are generally caused by an incorrect installation of MATLAB Runtime. For information on installing MATLAB Runtime, see "Download and Install MATLAB Runtime" on page 7-4.

Occasionally, there is a version mismatch between a DLL included with both MATLAB Runtime and Microsoft Windows. You can investigate which DLLs are called by your application using the Process Monitor tool. For information on using Process Monitor with your deployed application, see How can I use Process Monitor to troubleshoot the execution of my program?.

It is also possible that MATLAB Runtime is installed correctly, but the `PATH`,`LD_LIBRARY_PATH`, or `DYLD_LIBRARY_PATH` variable is set incorrectly. For information on setting environment variables, see "Set MATLAB Runtime Path for Deployment" on page 15-2.

Caution Do not solve these problems by moving libraries or other files within the MATLAB Runtime folder structure. The MATLAB Runtime system is designed to accommodate different MATLAB Runtime versions operating on the same machine. The folder structure is an important part of this feature.

Obtain Write Access to Install Directory

The first operation attempted by a compiled application is extraction of the deployable 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.

Deploy Newer Version of Application

When deploying a newer version of an executable, the executable needs to be redeployed, since it also contains the embedded deployable code archive. The deployable archive is keyed to a specific compilation session. Each time an application is recompiled, a new, matched deployable archive is created. Delete the existing application folder and run the new executable to ensure that the application can expand the new deployable archive. As above, write access is required to expand the new deployable archive.

Limitations and Restrictions

- "Limitations" on page 13-2
- "Ensure Multiplatform Portability for Compiled Applications" on page 13-8
- "Functions Not Supported for Compilation by MATLAB Compiler and MATLAB Compiler SDK " on page 13-10

Limitations

Packaging MATLAB and Toolboxes

MATLAB Compiler supports the full MATLAB language and almost all toolboxes based on MATLAB except:

- Most of the prebuilt graphical user interfaces included in MATLAB and its companion toolboxes.
- Functionality that cannot be called directly from the command line.

Compiled applications can run only on operating systems that run MATLAB. However, components generated by the MATLAB Compiler cannot be used in MATLAB. Also, since MATLAB Runtime 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, see the MATLAB System Requirements.

For information on multiplatform compatibility, see “Ensure Multiplatform Portability for Compiled Applications” (MATLAB Compiler SDK).

To see the full list of MATLAB Compiler limitations, visit: `https://www.mathworks.com/products/compiler/compiler_support.html`.

Note For a list of functions not supported by the MATLAB Compiler See “Functions Not Supported for Compilation by MATLAB Compiler and MATLAB Compiler SDK” on page 13-10.

Callback Problems Due to Missing Functions

When MATLAB Compiler creates an application, it packages the MATLAB files that you specify. In addition, it includes any other MATLAB files that your packaged MATLAB files call. MATLAB Compiler uses a dependency analysis function that 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 cannot discover the p-file dependencies.

The dependency analysis cannot locate a function if the only place the function is called in your MATLAB file is a call to the function in either of the following:

- Callback string
- Character array passed as an argument to the `feval` function or an ODE solver

Tip Dependent functions can also be hidden from the dependency analysis process in `.mat` files that are loaded by compiled applications. Use the `mcc -a` argument or the `%#function` pragma to identify `.mat` file classes or functions that are supported by the `load` command.

MATLAB Compiler does not look in these text character arrays for the names of functions to package.

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 character arrays
- Specifying callbacks with function handles
- Include the MATLAB function in the **Custom Requirements** area of a **Compiler** app, the `AdditionalFiles` option using a `compiler.build` function, or the `-a` flag using `mcc`.

Specifying Callbacks as Character Arrays

Create a list of all the functions that are specified only in callback character arrays and pass these functions using separate `%#function` pragma statements. This overrides the product 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.

Finding Missing Functions in MATLAB File

To find functions in your application that need to be listed in a `%#function` pragma, search your MATLAB file source code for text specified as callback character arrays or as arguments to the `feval`, `fminbnd`, `fminsearch`, `funm`, and `fzero` functions or any ODE solvers.

To find text used as callback character array, search for the characters “Callback” or “fcn” in your MATLAB file. This search finds all the `Callback` properties defined by graphics objects, such as `uicontrol` and `uimenu`. In addition, it finds the properties of figures and axes that end in `Fcn`, such as `CloseRequestFcn`, that also support callbacks.

Suppressing Warnings on UNIX System

Several warnings might appear when you run a standalone application on UNIX systems.

To suppress the `libjvm.so` warning, set the dynamic library path properly for your platform. For details, see “Set MATLAB Runtime Path for Deployment” on page 15-2.

You can also use the compiler option `-R -nojvm` to set your application's `nojvm` runtime option, if the application is capable of running without Java.

Cannot Use Graphics with -nojvm Option

If your program uses graphics and you compile with the `-nojvm` option, you get a runtime error.

Cannot Create Output File

If you receive this error, there are several possible causes to consider.

```
Can't create the output file filename
```

Possible causes include:

- 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 Packaged Functions

If you create a MATLAB file with self-documenting online help and package it, the results of following command are unintelligible:

```
help filename
```

Note For performance reasons, MATLAB file comments are stripped out before MATLAB Runtime encryption.

No MATLAB Runtime Versioning on Mac OS X

The feature that allows you to install multiple versions of MATLAB Runtime 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 your applications and components. Also, when you install a new version of MATLAB

Runtime on a target machine, you must delete the old version of MATLAB Runtime before installing the new one. You can have only one version of MATLAB Runtime on the target machine.

Older Neural Networks Not Deployable with MATLAB Compiler

Loading networks saved from older Deep Learning Toolbox versions requires some initialization routines that are not deployable. Therefore, these networks cannot be deployed without first being updated.

For example, deploying with Deep Learning 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 Packaged Mode

In compiled mode, only one argument can be present in a call to the MATLAB `printdlg` function (for example, `printdlg(gcf)`).

You cannot receive an error when making at call to `printdlg` with multiple arguments. However, when an application containing the multiple-argument call is packaged, the action fails with the following error message:

```
Error using = => printdlg at 11
PRINTDLG requires exactly one argument
```

Opening File Using which and open Does Not Search Current Working Folder

Using `which`, as in this example, does not cause the current working folder to be searched in deployed applications. In addition, it may cause unpredictable behavior of the `open` function.

```
function pathtest
which myFile.mat
open('myFile.mat')
```

Use one of the following solutions as an alternative:

- 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
  ```
- Use the `ctfroot` function to explicitly point to the file location relative to the deployable archive.

  ```
  load(fullfile(ctfroot,'..','datafiles','data1.mat'));
  ```

- Include your file in the **Files required for your application to run** area of the **Compiler** app, the `AdditionalFiles` option using a `compiler.build` function, or the `-a` flag using `mcc`. You can then locate the file using the `which` function.

For more information on including data files with your application, see "Include and Access Files in Packaged Applications" on page 5-15.

Restrictions on Using C++ SetData to Dynamically Resize an mwArray

You cannot use the C++ `SetData` method to dynamically resize `mwArray`s.

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.

Accepted File Types for Packaging

The valid and invalid file types for packaging using deployment apps are as follows.

Target Application	Valid File Types	Invalid File Types
Standalone applications	MATLAB MEX files, MATLAB scripts, MATLAB functions, and MATLAB class files. These files must have a single entry point.	Protected function files (`.p` files), Java functions, COM or .NET components, and data files.
C/C++ shared libraries, .NET assemblies, Java classes, Python packages, and COM components	MATLAB MEX files, MATLAB functions, and MATLAB class files. These files must have a single entry point.	MATLAB scripts, protected function files (`.p` files), Java functions, COM or .NET components, and data files.
MATLAB Production Server archives	MATLAB MEX files and MATLAB functions. These files must have a single entry point.	MATLAB scripts, MATLAB class files, protected function files (`.p` files), Java functions, COM or .NET components, and data files. MATLAB class files can be dependent files.

You can add other types of files to the packaged code archive, such as data files. For more information, see "Include and Access Files in Packaged Applications" on page 5-15.

See Also

More About

- “Functions Not Supported for Compilation by MATLAB Compiler and MATLAB Compiler SDK” on page 13-10

Ensure Multiplatform Portability for Compiled Applications

Compiled MATLAB code that contains only MATLAB files is platform independent, with some exceptions. You can run these files on any platform provided that the platform has either MATLAB or MATLAB Runtime installed. For more information on MATLAB Runtime, see “About MATLAB Runtime” on page 7-2.

The following components can only run on the same platform on which they were packaged:

- Components that contain platform-specific files, unless you also include files for the additional platforms and specify the additional platforms using `mcc` with the `-A` option
- Standalone applications
- Excel add-ins and COM components, which can only run on Windows
- Docker and microservice Docker images, which can only run in Linux containers
- C++ libraries compiled using the `mwArray` API
- .NET assemblies, except those compiled using .NET 6.0 or .NET Core
- Deployable archives compiled for MATLAB Production Server that include operating system-specific dependencies or content, such as MEX files or Simulink simulations
- Simulink Compiler artifacts

MEX Files

If your compiled MATLAB code contains MEX files which are platform dependent, you can do the following to allow them to run on other platforms. For this example, use the file `yprime.c` located in *`matlabroot`*`\extern\examples\mex`.

1. Compile your MEX file once on each platform where you want to run your application.

   For example, if you want to run the application on the Windows 64-bit platform as well as the Linux 64-bit platform, compile `yprime.c` twice: once on a PC to get `yprime.mexw64` and then again on a Linux 64-bit machine to get `yprime.mexa64`.
2. Create a simple MATLAB function named `callyprime.m` that calls `yprime`.

   ```
   function callyprime
   disp(yprime(1,1:4));
   ```
3. Compile the package on one platform and use the `-a` option of `mcc` or the `AdditionalFiles` option of a `compiler.build` function to include the MEX file compiled on the other platform(s). Ensure that the Linux MEX file is in the same folder as the Windows MEX file.

   For example, if you are creating a Java package on a Windows machine and you want to ensure the package can run on the Linux 64-bit platform, include the Linux MEX file `yprime.mexa64`.

   ```
   mcc -W 'java:myComp,myClass' callyprime.m -a yprime.mexa64
   ```

   It is not necessary to explicitly include the `glnxa64` architecture here using `-A`, as the dependency analysis process will detect the Linux MEX file and enable the platform.

Tip If you are unsure if your application contains MEX files, do the following:

1. Run `mcc` with the `-v` option to list the names of the MEX files, or enable the `Verbose` option in a `compiler.build` function.
2. Obtain appropriate versions of these files from the version of MATLAB installed on your target operating system.
3. Include these versions in the archive by running `mcc` with the `-a` option, or use the `AdditionalFiles` options in a `compiler.build` function.

MATLAB Toolboxes

Toolbox functionality that runs seamlessly across platforms when executed from within the MATLAB desktop environment will continue to run seamlessly across platforms when deployed. However, if a particular toolbox functionality is designed to run on a specific platform, then that functionality will run only on that specific platform when deployed. For example, functionality from the Data Acquisition Toolbox™ runs only on Windows.

Java JAR Files

Java JAR files are platform independent. However, if you add operating system-specific dependencies or content to your package, such as MEX files or Simulink simulations, the generated archive is limited to the compatible system. You can override the restriction using `mcc` with the `-A` option.

JAR files produced by MATLAB Compiler SDK are tested and qualified to run on platforms supported by MATLAB. For more information, see the Platform Roadmap for MATLAB.

Web Apps

In most cases, you can generate a web app archive (`.ctf` file) on one platform and deploy to a server running on any other supported platform. Unless you add operating system-specific dependencies or content, such as MEX files or Simulink simulations to your applications, the generated archives are platform-independent.

See Also

`mcc`

Related Examples

- "Include and Access Files in Packaged Applications" (MATLAB Compiler SDK)
- "Limitations" (MATLAB Compiler SDK)
- "Testing Failures" (MATLAB Compiler SDK)

External Websites

- Platform Roadmap for MATLAB

Functions Not Supported for Compilation by MATLAB Compiler and MATLAB Compiler SDK

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 cannot 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. For an updated list of such functions, see Support for MATLAB and Toolboxes.

Functions that cannot be compiled fall into the following categories:

- Functions that print or report MATLAB code from a function, such as the MATLAB `help` function or debug functions.
- Simulink functions, in general.
- Functions that require a command line, such as the MATLAB `lookfor` function.
- `clc`, `home`, and `savepath`, which do not do anything in deployed mode.

In addition, there are functions and programs that have been identified as non-deployable due to licensing restrictions.

Only certain tools that allow run-time manipulation of figures are supported for compilation, for example, adding legends, selecting data points, zooming in and out, etc.

`mccExcludedFiles.log` lists all the functions and files excluded by `mcc`. It is created after each attempted build.

Functions from unsupported products are reported in a warning, and listed in `unresolvedSymbols.txt`. For more information on product support for MATLAB Compiler, see https://www.mathworks.com/products/compiler/compiler_support.html

List of Unsupported Functions and Programs

`add_block`
`add_line`
`checkcode`
`close_system`
`colormapeditor`
`commandwindow`
Control System Toolbox™ prescale GUI
`dbclear`
`dbcont`
`dbdown`
`dbquit`
`dbstack`
`dbstatus`
`dbstep`
`dbstop`
`dbtype`
`dbup`
`delete_block`
`delete_line`
`depfun`
`doc`
`echo`
`edit`
`export`
`fields`
`get_param`
`help`
`home`
`inmem`
`inspect`
`keyboard`
`linkdata`
`linmod`
`load_system`
`matlab.unittest.TestSuite.fromProject`
`mislocked`
`mlock`
`more`

```
munlock
new_system
open
open_system
pack
pcode
plotedit
profile
profsave
publish
quit
rehash
restoredefaultpath
run
segment
set_param
sldebug
type
```

Note The `diary` function is supported. However, when called from a compiled Python package it will produce an empty text file. For information on logging command window text with a compiled Python package, see "Redirect Standard Output and Error to Python".

14

Package to Docker

Package MATLAB Standalone Applications into Docker Images

Supported Platform: Linux only.

This example shows how to package a MATLAB standalone application into a Docker image.

This option is best for developers who want to distribute an application in a standardized format with all dependencies included, or to run batch jobs in an orchestrator. To create a microservice Docker image that provides an HTTP/HTTPS endpoint, see "Create Microservice Docker Image" (MATLAB Compiler SDK).

Prerequisites

1. Verify that you have Docker installed on your Linux machine by typing `docker` in the terminal. If you do not have Docker installed, you can follow the instructions on the Docker website to install and set up Docker.

   `https://docs.docker.com/engine/install/`
2. Test your Docker installation by typing the following at the system terminal:

   ```
   docker run hello-world
   ```

   If your Docker installation is working correctly, you see the following message:

   ```
   Hello from Docker!
   This message shows that your installation appears to be working correctly.
   ```
3. Verify that MATLAB Runtime installer is available on your machine. You can verify its existence by executing the `compiler.runtime.download` function at the MATLAB command prompt. If there is an existing installer on the machine, the function returns its location. Otherwise, it downloads the MATLAB Runtime installer matching the version and update level of MATLAB from where the command is executed.

   If the computer you are using is not connected to the Internet, you must download the MATLAB Runtime installer from a computer that is connected to the Internet. After downloading the MATLAB Runtime installer, you need to transfer the installer to the offline computer. You can download the installer from the MathWorks website.

   `https://www.mathworks.com/products/compiler/matlab-runtime.html`

Create Function in MATLAB

Write a MATLAB function called `mymagic` and save it as `mymagic.m`.

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

Test the function at the MATLAB command prompt.

```
mymagic(5)

    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 Standalone Application

Package the `mymagic` function into a standalone application using the `compiler.build.standaloneApplication` function.

```
res = compiler.build.standaloneApplication('mymagic.m','TreatInputsAsNumeric',true)
```

```
res = 
  Results with properties:

    BuildType: 'standaloneApplication'
        Files: {3×1 cell}
      Options: [1×1 compiler.build.StandaloneApplicationOptions]
  RuntimeDependencies: [1×1 compiler.runtime.Dependencies]
```

The `Results` object `res` returned at the MATLAB command prompt contains information about the build.

Once the build is complete, the function creates a folder named `mymagicstandaloneApplication` in your current directory to store the standalone application.

Package Standalone Application into Docker Image

Create DockerOptions Object

Prior to creating a Docker image, create a `DockerOptions` object using the `compiler.package.DockerOptions` function and pass the `Results` object `res` and an image name `mymagic-standalone-app` as input arguments. The `compiler.package.DockerOptions` function lets you customize Docker image packaging.

```
opts = compiler.package.DockerOptions(res,'ImageName','mymagic-standalone-app')
```

```
opts = 
  DockerOptions with properties:

                EntryPoint: 'mymagic'
    AdditionalInstructions: {}
        AdditionalPackages: {}
        ExecuteDockerBuild: on
                 ImageName: 'mymagic-standalone-app'
             DockerContext: './mymagic-standalone-appdocker'
```

Create Docker Image

Create a Docker image using the `compiler.package.docker` function and pass the `Results` object `res` and the `DockerOptions` object `opts` as input arguments.

```
compiler.package.docker(res,'Options',opts)
```

```
Generating Runtime Image
Cleaning MATLAB Runtime installer location. It may take several minutes...
Copying MATLAB Runtime installer. It may take several minutes...
...
```

```
...
...
Successfully built 6501fa2bc057
Successfully tagged mymagic-standalone-app:latest

DOCKER CONTEXT LOCATION:

/home/user/MATLAB/work/mymagic-standalone-appdocker

SAMPLE DOCKER RUN COMMAND:

docker run --rm -e "DISPLAY=:0" -v /tmp/.X11-unix:/tmp/.X11-unix mymagic-standalone-app
```

Once packaging is complete, the function creates a folder named `mymagic-standalone-appdocker` in your current directory. This folder is the Docker context and contains the Dockerfile. The `compiler.package.docker` function also returns the location of the Docker context and a sample Docker run command. You can use the sample Docker run command to test whether your image executes correctly. If the application requires input arguments, append them to the sample command.

During the packaging process, the necessary bits for MATLAB Runtime are packaged as a parent Docker image and the standalone application is packaged as a child Docker image.

Test Docker Image

Open a Linux terminal and navigate to the Docker context folder. Verify that the `mymagic-standalone-app` Docker image is listed in your list of Docker images.

```
$ docker images
```

```
REPOSITORY                                  TAG         IMAGE ID            CREATED
mymagic-standalone-app                      latest      6501fa2bc057        23 seconds ago
matlabruntime/r2025b/update0/4000000000000000   latest      c6eb5ba4ae69        24 hours ago
```

After verifying that the `mymagic-standalone-app` Docker image is listed in your list of Docker images, execute the sample run command with the input argument `5`:

```
$ docker run --rm -e "DISPLAY=:0" -v /tmp/.X11-unix:/tmp/.X11-unix mymagic-standalone-app 5
```

```
No protocol specified

out =

    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
```

The standalone application is packaged and can now be run as a Docker image.

Note When running applications that generate plots or graphics, execute the `xhost` program with the `+` option prior to running your Docker image.

```
xhost +
```

The `xhost` program controls access to the X display server, thereby enabling plots and graphics to be displayed. The `+` option indicates that everyone has access to the X display server. If you run the `xhost` program with the `+` option prior to running applications that do not generate plots or graphics, the message `No protocol specified` is no longer displayed.

Share Docker Image

You can share your Docker image in various ways.

- Push your image to the Docker's central registry DockerHub, or to your private registry. This is the most common workflow.
- Save your image as a tar archive and share it with others. This workflow is suitable for immediate testing.

For details about pushing your image to Docker's central registry or your private registry, consult the Docker documentation.

Save Docker Image as Tar Archive

To save your Docker image as a tar archive, open a Linux terminal, navigate to the Docker context folder, and type the following.

```
$ docker save mymagic-standalone-app -o mymagic-standalone-app.tar
```

A file named `mymagic-standalone-app.tar` is created in your current folder. Set the appropriate permissions using `chmod` prior to sharing the tarball with other users.

Load Docker Image from Tar Archive

Load the image contained in the tarball on the end-user's machine and then run it.

```
$ docker load --input mymagic-standalone-app.tar
```

Verify that the image is loaded.

```
$ docker images
```

Run Docker Image

```
$ xhost +
$ docker run --rm -e "DISPLAY=:0" -v /tmp/.X11-unix:/tmp/.X11-unix mymagic-standalone-app 5
```

See Also

`compiler.package.docker` | `compiler.package.DockerOptions` | `compiler.build.standaloneApplication` | `compiler.runtime.download`

Related Examples

- "Create Microservice Docker Image" (MATLAB Compiler SDK)

Reference Information

- “Set MATLAB Runtime Path for Deployment” on page 15-2
- “MATLAB Compiler Licensing” on page 15-6
- “Deployment Product Terms” on page 15-7

Set MATLAB Runtime Path for Deployment

In this section...

"Library Path Environment Variables and MATLAB Runtime Folders" on page 15-2
"Windows" on page 15-3
"Linux" on page 15-3
"macOS" on page 15-4
"Set Path Permanently on UNIX" on page 15-4

Applications generated with MATLAB Compiler or MATLAB Compiler SDK use the system library path to locate the MATLAB Runtime libraries. The MATLAB Runtime installer for Windows automatically sets the library path during installation, but on Linux or macOS you must add the libraries manually. After you install MATLAB Runtime, add the run-time folders to the system library path according to the instructions for your operating system and shell environment.

Alternatively, you can pass the location of MATLAB Runtime as an input to the associated shell script (`run_`*`application`*`.sh`) on Linux or macOS to launch an application.

Note

- Your library path may contain multiple versions of MATLAB Runtime. Applications launched without using the shell script use the first version listed in the path.
- Save the value of your current library path as a backup before modifying it.
- If you are using a network install of MATLAB Runtime, see "Deploy Applications and MATLAB Runtime on Network Drives" on page 7-10.

Library Path Environment Variables and MATLAB Runtime Folders

Operating System	Environment Variable	Directories
Windows	`PATH`	`<MATLAB_RUNTIME_INSTALL_DIR>\runtime\<arch>`

Linux	`LD_LIBRARY_PATH`	`<MATLAB_RUNTIME_INSTALL_DIR>/runtime/glnxa64`
		`<MATLAB_RUNTIME_INSTALL_DIR>/bin/glnxa64`
		`<MATLAB_RUNTIME_INSTALL_DIR>/sys/os/glnxa64`
		`<MATLAB_RUNTIME_INSTALL_DIR>/extern/bin/glnxa64`

macOS (Intel processor)	`DYLD_LIBRARY_PATH`	`<MATLAB_RUNTIME_INSTALL_DIR>/runtime/maci64`
		`<MATLAB_RUNTIME_INSTALL_DIR>/bin/maci64`
		`<MATLAB_RUNTIME_INSTALL_DIR>/sys/os/maci64`
		`<MATLAB_RUNTIME_INSTALL_DIR>/extern/bin/maci64`

Operating System	Environment Variable	Directories
macOS (Apple silicon)	DYLD_LIBRARY_PATH	*<MATLAB_RUNTIME_INSTALL_DIR>*/runtime/maca64 *<MATLAB_RUNTIME_INSTALL_DIR>*/bin/maca64 *<MATLAB_RUNTIME_INSTALL_DIR>*/sys/os/maca64 *<MATLAB_RUNTIME_INSTALL_DIR>*/extern/bin/maca64

Windows

The MATLAB Runtime installer for Windows automatically sets the library path during installation. If you do not use the installer, complete the following steps to set the PATH environment variable permanently.

1. Run C:\Windows\System32\SystemPropertiesAdvanced.exe and click the **Environment Variables...** button.
2. Select the system variable Path and click **Edit...**.

   Note If you do not have administrator rights on the machine, select the user variable Path instead of the system variable.
3. Click **New** and add the folder *<MATLAB_RUNTIME_INSTALL_DIR>*\runtime*<arch>*.

   For example, if you are using MATLAB Runtime R2025b located in the default installation folder on 64-bit Windows, add C:\Program Files\MATLAB\MATLAB Runtime\R2025b\runtime\win64.
4. Click **OK** to apply the change.

Note If the path contains multiple versions of MATLAB Runtime, applications use the first version listed in the path.

Linux

For information on setting environment variables in shells other than Bash, see your shell documentation.

Bash Shell

1. Display the current value of LD_LIBRARY_PATH in the terminal.

   ```
   echo $LD_LIBRARY_PATH
   ```
2. Append the MATLAB Runtime folders to the LD_LIBRARY_PATH variable for the current session.

   For example, if you are using MATLAB Runtime R2025b located in the default installation folder, use the following command.

   ```
   export LD_LIBRARY_PATH="${LD_LIBRARY_PATH:+${LD_LIBRARY_PATH}:}\
   /usr/local/MATLAB/MATLAB_Runtime/R2025b/runtime/glnxa64:\
   /usr/local/MATLAB/MATLAB_Runtime/R2025b/bin/glnxa64:\
   ```

```
/usr/local/MATLAB/MATLAB_Runtime/R2025b/sys/os/glnxa64:\
/usr/local/MATLAB/MATLAB_Runtime/R2025b/extern/bin/glnxa64"
```

Note If you require Mesa Software OpenGL® rendering to resolve low level graphics issues, add the folder *<MATLAB_RUNTIME_INSTALL_DIR>*`/sys/opengl/lib/glnxa64` to the path.

3 Display the new value of `LD_LIBRARY_PATH` to ensure the path is correct.

```
echo $LD_LIBRARY_PATH
```

4 Type `ldd --version` to check your version of GNU® C library (`glibc`). If the version displayed is 2.17 or lower, add *<MATLAB_RUNTIME_INSTALL_DIR>*`/bin/glnxa64/glibc-2.17_shim.so` to the `LD_PRELOAD` environment variable using the following command.

```
export LD_PRELOAD="${LD_PRELOAD:+${LD_PRELOAD}:}\
/usr/local/MATLAB/MATLAB_Runtime/R2025b/bin/glnxa64/glibc-2.17_shim.so"
```

5 To make these changes permanent, see “Set Path Permanently on UNIX” on page 15-4.

macOS

1 Display the current value of `DYLD_LIBRARY_PATH` in the terminal.

```
echo $DYLD_LIBRARY_PATH
```

2 Append the MATLAB Runtime folders to the `DYLD_LIBRARY_PATH` variable for the current session.

For example, if you are using MATLAB Runtime R2025b located in the default installation folder, use one of the following commands depending on your system architecture.

For Intel processor-based macOS, use the `maci64` directories.

```
export DYLD_LIBRARY_PATH="${DYLD_LIBRARY_PATH:+${DYLD_LIBRARY_PATH}:}\
/Applications/MATLAB/MATLAB_Runtime/R2025b/runtime/maci64:\
/Applications/MATLAB/MATLAB_Runtime/R2025b/bin/maci64:\
/Applications/MATLAB/MATLAB_Runtime/R2025b/sys/os/maci64:\
/Applications/MATLAB/MATLAB_Runtime/R2025b/extern/bin/maci64"
```

For Apple silicon-based macOS, use the `maca64` directories.

```
export DYLD_LIBRARY_PATH="${DYLD_LIBRARY_PATH:+${DYLD_LIBRARY_PATH}:}\
/Applications/MATLAB/MATLAB_Runtime/R2025b/runtime/maca64:\
/Applications/MATLAB/MATLAB_Runtime/R2025b/bin/maca64:\
/Applications/MATLAB/MATLAB_Runtime/R2025b/sys/os/maca64:\
/Applications/MATLAB/MATLAB_Runtime/R2025b/extern/bin/maca64"
```

3 Display the value of `DYLD_LIBRARY_PATH` to ensure the path is correct.

```
echo $DYLD_LIBRARY_PATH
```

4 To make these changes permanent, see “Set Path Permanently on UNIX” on page 15-4.

Set Path Permanently on UNIX

Caution The MATLAB Runtime libraries may conflict with other applications that use the library path. In this case, set the path only for the current session, or run MATLAB Compiler applications using the generated shell script.

To set an environment variable at login on Linux or macOS, append the `export` command to the shell configuration file `~/.bash_profile` in a `Bash` shell or `~/.zprofile` in a `Zsh` shell.

To determine your current shell environment, type `echo $SHELL`.

See Also

More About

- “Download and Install MATLAB Runtime” on page 7-4
- “Deploy Applications and MATLAB Runtime on Network Drives” on page 7-10
- “Change Environment Variable for Shell Command”

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 has different behavior in MATLAB mode and standalone mode.

Running MATLAB Compiler in MATLAB Mode

When you run MATLAB Compiler from “inside” of the MATLAB environment, that is, you run `mcc` from the MATLAB command prompt, you hold the MATLAB Compiler license as long as MATLAB remains open. To give up the MATLAB Compiler license, exit MATLAB.

Running MATLAB Compiler in Standalone Mode

If you run MATLAB Compiler from a DOS or UNIX prompt, you are running from “outside” of MATLAB. In this case, MATLAB Compiler

- Does not require MATLAB to be running on the system where MATLAB Compiler is running
- Gives the user a dedicated 30-minute time allotment during which the user has complete ownership over a license to MATLAB Compiler

Each time a user requests MATLAB Compiler, the user begins a 30-minute time period as the sole owner of the MATLAB Compiler license. Anytime during the 30-minute segment, if the same user requests MATLAB Compiler , the user gets a new 30-minute allotment. When the 30-minute interval has elapsed, if a different user requests MATLAB Compiler , the new user gets the next 30-minute interval.

When a user requests MATLAB Compiler and a license is not available, the user receives the message

```
Error: Could not check out a Compiler License.
```

This message is given when no licenses are available. As long as licenses are available, the user gets the license and no message is displayed. The best way to guarantee that all MATLAB Compiler users have constant access to MATLAB Compiler is to have an adequate supply of licenses for your users.

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 function 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.

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 and MATLAB Compiler SDK, to compile MATLAB code involves generating a binary that wraps around MATLAB code, enabling it to execute in various computing environments. For example, when MATLAB code is compiled into a Java package, a Java wrapper provides Java code that enables the MATLAB code to execute in a Java environment.

COM component — In MATLAB Compiler, the executable back-end code behind a Microsoft Excel add-in. In MATLAB Compiler SDK, an executable component, to be integrated with Microsoft COM applications.

Console application — Any application that is executed from a system command prompt window.

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 MATLAB code into a larger-scale computing environment, usually to an enterprise application, and often to end users.

Deployable archive — The deployable archive is embedded by default in each binary generated by MATLAB Compiler or MATLAB Compiler SDK. It houses the deployable package. All MATLAB-based content in the deployable archive uses the Advanced Encryption Standard (AES) cryptosystem.

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.

I

Integration — Combining deployed MATLAB code'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.

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 product which provides the environment required for programming in Java.

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 Runtime — 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.

MATLAB Runtime singleton — See *Shared MATLAB Runtime instance*.

MATLAB Runtime workers — A MATLAB Runtime session. Using MATLAB Production Server software, you have the option of specifying more than one MATLAB Runtime session, using the `--num-workers` options in the server configurations file.

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`. For more details, see Server Configuration Properties.

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 SDK 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 the compiler. It is the command-line equivalent of using the compiler apps.

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 MATLAB code, along with the MATLAB Runtime and other files, into an installer that can be distributed to others. The compiler apps place the installer in the `package` 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 for Microsoft Windows.

Shared MATLAB Runtime instance — When using MATLAB Compiler SDK, you can create a shared MATLAB Runtime instance, also known as a singleton. When you invoke MATLAB Compiler with the `-S` option through the compiler (using either `mcc` or a compiler app), a single MATLAB Runtime instance is created for each COM component or Java package in an application. You reuse this instance by sharing it among all subsequent class instances. Such sharing results in more efficient memory usage and eliminates the MATLAB Runtime startup cost in each subsequent class instantiation. All class instances share a single MATLAB workspace and share global variables in the deployed MATLAB files. MATLAB Compiler SDK creates singletons by default for .NET assemblies. MATLAB Compiler creates singletons by default for the COM components used by the Excel add-ins.

State — The present condition of MATLAB, or the MATLAB 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.

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 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 website 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.

16

Functions

%#exclude

Ignore file or function dependencies during MATLAB Compiler dependency analysis

Syntax

`%#exclude fileOrFunction1 [fileOrFunction2 ... fileOrFunctionN]`

Description

`%#exclude fileOrFunction1 [fileOrFunction2 ... fileOrFunctionN]` pragma informs the compiler that the specified files or functions need to be excluded from dependency analysis during compilation. The pragma also suppresses the compile-time warning that the files or functions cannot be compiled.

Examples

Use `%#exclude` and `isdeployed` for Non-Deployable Function

Use `isdeployed` with the `%#exclude` pragma to suppress compile-time warnings for the non-deployable function `edit`.

```
if ~isdeployed
    %#exclude edit
    edit('readme.txt');
end
```

The `~isdeployed` statement prevents the code from being invoked in the deployed component. The `%#exclude` pragma suppresses the warning that `edit` cannot be compiled.

Use %#exclude for Data File

Create a MATLAB function that uses pragmas to include and exclude files.

1. Write a function named `testExclusion` that uses two pragmas.

   ```
   function testExclusion()

   %#exclude foo.mat
   load foo.mat
   load bar.mat

   %#function foo.txt
   fid = fopen('foo.txt');
   fclose(fid)
   ```

   The `%#exclude` pragma informs the compiler to exclude the file `foo.mat` during compilation.

   The `%#function` pragma informs the compiler that the file `foo.txt` should be included in the compilation.

2 Compile the function into a standalone application using `mcc`. The `-m` option builds a standalone executable. The `-a` option adds files to the deployable archive. The `-X` option instructs `mcc` to ignore data files during dependency analysis.

   Executing `mcc -m testExclusion.m` results in:

   - `bar.mat` and `foo.txt` being included during dependency analysis
   - `foo.mat` being excluded

   Executing `mcc -m testExclusion.m -X` results in:

   - `foo.txt` being included during dependency analysis
   - `bar.mat` and `foo.mat` being excluded

   Executing `mcc -m testExclusion.m -X -a foo.mat` results in:

   - `foo.mat` and `foo.txt` being included during dependency analysis
   - `bar.mat` being excluded

   In the last case, the `-a` option takes precedence over the `%#exclude` pragma.

Version History

Introduced in R2020a

See Also

`mcc` | `%#function` | `isdeployed`

%#function

Pragma to help MATLAB Compiler locate functions called through `feval`, `eval`, Handle Graphics callback, or objects loaded from MAT files

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`, Handle Graphics callback, or objects loaded from MAT files.

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`.

Example 3

```
function foo
   %#function ClassificationSVM

      load('svm-classifier.mat');
      num_dimensions = size(svm_model.PredictorNames, 2);

    end %function foo
```

In this example, an object from the class `ClassificationSVM` is loaded from a MAT file. For more information, see "Include and Access Files in Packaged Applications" on page 5-15.

Version History

Introduced before R2006a

compiler.build.Results

Compiler build results object

Description

A `compiler.build.Results` object contains information about the build type, generated files, support packages, build options, and dependencies of a `compiler.build` function.

All `Results` properties are read-only. You can use dot notation to query these properties.

With MATLAB Compiler, you can create standalone applications, Excel add-ins, or web app archives.

With MATLAB Compiler SDK, you can create C/C++ shared libraries, .NET assemblies, COM components, Java packages, Python packages, MATLAB Production Server deployable archives, or Excel add-ins for MATLAB Production Server.

Creation

There are several ways to create a `compiler.build.Results` object.

- Create a standalone application using `compiler.build.standaloneApplication` (example on page 16-10).
- Create a standalone Windows application using `compiler.build.standaloneWindowsApplication` (example on page 16-10).
- Create a web app archive using `compiler.build.webAppArchive` (example on page 16-11).
- Create an Excel add-in using `compiler.build.excelAddIn` (example on page 16-11).

If you have a MATLAB Compiler SDK license, you can also create the following objects.

- Create a C shared library using `compiler.build.cSharedLibrary` (example (MATLAB Compiler SDK)).
- Create a C++ shared library using `compiler.build.cppSharedLibrary` (example (MATLAB Compiler SDK)).
- Create a .NET assembly using `compiler.build.dotNETAssembly` (example (MATLAB Compiler SDK)).
- Create a Java package using `compiler.build.javaPackage` (example (MATLAB Compiler SDK)).
- Create a Python package using `compiler.build.pythonPackage` (example (MATLAB Compiler SDK)).
- Create a production server archive using `compiler.build.productionServerArchive` (example (MATLAB Compiler SDK)).
- Create an Excel add-in for MATLAB Production Server using `compiler.build.excelClientForProductionServer` (example (MATLAB Compiler SDK)).
- Create a COM component using `compiler.build.comComponent` (example (MATLAB Compiler SDK)).

Properties

`BuildType` — Build type

`'standaloneApplication'` | `'standaloneWindowsApplication'` | `'webAppArchive'` | `'productionServerArchive'` | `'excelAddIn'` | `'comComponent'` | `'cSharedLibrary'` | `'cppSharedLibrary'` | `'dotNETAssembly'` | `'javaPackage'` | `'pythonPackage'` | `'excelClientForProductionServer'`

This property is read-only.

The build type of the `compiler.build` function used to generate the results, specified as a character vector:

`compiler.build` Function	Build Type
`compiler.build.standaloneApplication`	`'standaloneApplication'`
`compiler.build.standaloneWindowsApplication`	`'standaloneWindowsApplication'`
`compiler.build.webAppArchive`	`'webAppArchive'`
`compiler.build.productionServerArchive`	`'productionServerArchive'`
`compiler.build.excelAddIn`	`'excelAddIn'`
`compiler.build.comComponent`	`'comComponent'`
`compiler.build.cSharedLibrary`	`'cSharedLibrary'`
`compiler.build.cppSharedLibrary`	`'cppSharedLibrary'`
`compiler.build.dotNETAssembly`	`'dotNETAssembly'`
`compiler.build.javaPackage`	`'javaPackage'`
`compiler.build.pythonPackage`	`'pythonPackage'`
`compiler.build.excelClientForProductionServer`	`'excelClientForProductionServer'`

Data Types: `char`

`Files` — Paths to compiled files

cell array of character vectors

This property is read-only.

Paths to the compiled files of the `compiler.build` function used to generate the results, specified as a cell array of character vectors.

Build Type	Files

`'standaloneApplication'`	2×1 cell array
	`{'path\to\`*`ExecutableName`*`.exe'}`
	`{'path\to\readme.txt'}`

Build Type	Files

`'standaloneWindowsApplication'`	3×1 cell array
	
	`{'path\to\`*`ExecutableName`*`.exe'}`
	`{'path\to\splash.png'}`
	`{'path\to\readme.txt'}`

`'webAppArchive'`	1×1 cell array `{'path\to\`*`ArchiveName`*`.ctf'}`
`'productionServerArchive'`	1×1 cell array `{'path\to\`*`ArchiveName`*`.ctf'}`

`'excelAddIn'`	2×1 or 4×1 cell array
	
	`{'path\to\`*`AddInName_AddInVersion`*`.dll'}`
	`{'path\to\`*`AddInName`*`.bas'}`
	`{'path\to\`*`AddInName`*`.xla'}`
	`{'path\to\GettingStarted.html'}`
	
	Note The files *`AddInName`*`.bas` and *`AddInName`*`.xla` are included only if you enable the `'GenerateVisualBasicFile'` option.

`'comComponent'`	2×1 cell array
	
	`{'path\to\`*`ComponentName_ComponentVersion`*`.dll'}`
	`{'path\to\GettingStarted.html'}`

`'cSharedLibrary'`	4×1 cell array
	
	`{'path\to\`*`LibraryName`*`.h'}`
	`{'path\to\`*`LibraryName`*`.dll'}`
	`{'path\to\`*`LibraryName`*`.lib'}`
	`{'path\to\GettingStarted.html'}`

`'cppSharedLibrary'`	2×1 or 4×1 cell array
	
	Using the `matlab-data` interface:
	
	`{'path\to\v2\'}`
	`{'path\to\GettingStarted.html'}`
	
	Using the `mwArray` interface:
	
	`{'path\to\`*`LibraryName`*`.h'}`
	`{'path\to\`*`LibraryName`*`.dll'}`
	`{'path\to\`*`LibraryName`*`.lib'}`
	`{'path\to\GettingStarted.html'}`

`'dotNETAssembly'`	4×1 cell array
	
	`{'path\to\`*`AssemblyName`*`.dll'}`
	`{'path\to\`*`AssemblyName`*`Native.dll'}`
	`{'path\to\`*`AssemblyName`*`_overview.html'}`
	`{'path\to\GettingStarted.html'}`

Build Type	Files

`'javaPackage'`	3×1 cell array
	`{'path\to\`*`PackageName`*`.jar'}`
	`{'path\to\doc\'}`
	`{'path\to\GettingStarted.html'}`

`'pythonPackage'`	3×1 cell array
	`{'path\to\example\'}`
	`{'path\to\setup.py'}`
	`{'path\to\GettingStarted.html'}`

`'excelClientForProductionServer'`	1×1 or 3×1 cell array
	`{'path\to\`*`AddInName`*`.dll'}`
	`{'path\to\`*`AddInName`*`.bas'}`
	`{'path\to\`*`AddInName`*`.xla'}`
	Note The files *`AddInName`*`.bas` and *`AddInName`*`.xla` are included only if you enable the `'GenerateVisualBasicFile'` option.

Example: `{'D:\Documents\MATLAB\work\MagicSquarewebAppproductionServerArchive\MagicSquare.ctf'}`

Data Types: `cell`

`IncludedSupportPackages` — Support packages
cell array of character vectors

This property is read-only.

Support packages included in the generated component, specified as a cell array of character vectors.

`Options` — Build options
`StandaloneApplicationOptions` | `WebAppArchiveOptions` | `ProductionServerArchiveOptions` | `ExcelAddInOptions` | `COMComponentOptions` | `CSharedLibraryOptions` | `CppSharedLibraryOptions` | `DotNETAssemblyOptions` | `JavaPackageOptions` | `PythonPackageOptions` | `ExcelClientForProductionServerOptions`

This property is read-only.

Build options of the `compiler.build` function used to generate the results, specified as an options object of the corresponding build type.

Build Type	Options
`'standaloneApplication'`	`StandaloneApplicationOptions`
`'standaloneWindowsApplication'`	`StandaloneApplicationOptions`
`'webAppArchive'`	`WebAppArchiveOptions`
`'productionServerArchive'`	`ProductionServerArchiveOptions`
`'excelAddIn'`	`ExcelAddInOptions`

Build Type	Options
'comComponent'	COMComponentOptions
'cSharedLibrary'	CSharedLibraryOptions
'cppSharedLibrary'	CppSharedLibraryOptions
'dotNETAssembly'	DotNETAssemblyOptions
'javaPackage'	JavaPackageOptions
'pythonPackage'	PythonPackageOptions
'excelClientForProductionServer'	ExcelClientForProductionServerOptions

`RuntimeDependencies` — Optional and required dependencies
object containing two tables

This property is read-only.

Runtime dependencies for the generated component, specified as a `Dependencies` object containing two tables, `Required` and `Optional`. Each table contains a collection of dependencies.

Examples

Get Build Information from Standalone Application

Create a standalone application and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

```
results = compiler.build.standaloneApplication('magicsquare.m')
```

```
results = 

              BuildType: 'standaloneApplication'
                  Files: {2×1 cell}
IncludedSupportPackages: {}
                Options: [1×1 compiler.build.StandaloneApplicationOptions]
    RuntimeDependencies: [1×1 compiler.runtime.Dependencies]
```

The `Files` property contains the paths to the `magicsquare` standalone executable and `readme.txt` files.

Get Build Information from Standalone Windows Application

Create a standalone Windows application on a Windows system and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `Mortgage.mlapp`.

```
results = compiler.build.standaloneWindowsApplication('Mortgage.mlapp')
```

```
results = 
  Results with properties:

              BuildType: 'standaloneWindowsApplication'
                  Files: {3×1 cell}
IncludedSupportPackages: {}
                Options: [1×1 compiler.build.StandaloneApplicationOptions]
    RuntimeDependencies: [1×1 compiler.runtime.Dependencies]
```

The `Files` property contains the paths to the following files:

- `Mortgage.exe`
- `splash.png`
- `readme.txt`

Get Build Information from Web App Archive

Create a web app archive and save information about the build type, archive file, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `Mortgage.mlapp`.

```
results = compiler.build.webAppArchive('Mortgage.mlapp')
```

```
results = 

  Results with properties:

              BuildType: 'webAppArchive'
                  Files: {'D:\Documents\MATLAB\work\MortgagewebAppArchive\Mortgage.ctf'}
IncludedSupportPackages: {}
                Options: [1×1 compiler.build.WebAppArchiveOptions]
    RuntimeDependencies: [1×1 compiler.runtime.Dependencies]
```

The `Files` property contains the path to the deployable archive file `Mortgage.ctf`.

Get Build Information from Production Server Archive

Create a production server archive and save information about the build type, archive file, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

```
results = compiler.build.productionServerArchive('magicsquare.m')
```

```
results = 

  Results with properties:

                  BuildType: 'productionServerArchive'
                      Files: {'D:\Documents\MATLAB\work\magicsquareproductionServerArchive\magicsquare.ctf'}
    IncludedSupportPackages: {}
                    Options: [1×1 compiler.build.ProductionServerArchiveOptions]
        RuntimeDependencies: [1×1 compiler.runtime.Dependencies]
```

The `Files` property contains the path to the deployable archive file `magicsquare.ctf`.

Get Build Information from Excel Add-In

Create an Excel add-in and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

```
results = compiler.build.excelAddIn('magicsquare.m')
```

```
results = 

  Results with properties:

              BuildType: 'excelAddIn'
                  Files: {2×1 cell}
IncludedSupportPackages: {}
                Options: [1×1 compiler.build.ExcelAddInOptions]
    RuntimeDependencies: [1×1 compiler.runtime.Dependencies]
```

The `Files` property contains the paths to the following compiled files:

- `magicsquare_1_0.dll`
- `GettingStarted.html`

Note The files `magicsquare.bas` and `magicsquare.xla` are included in `Files` only if you enable the `'GenerateVisualBasicFile'` option in the build command.

Get Build Information from COM Component

Create a COM component on a Windows system and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

```
results = compiler.build.comComponent('magicsquare.m')
```

```
results = 

  Results with properties:

              BuildType: 'comComponent'
                  Files: {2×1 cell}
IncludedSupportPackages: {}
                Options: [1×1 compiler.build.COMComponentOptions]
    RuntimeDependencies: [1×1 compiler.runtime.Dependencies]
```

The `Files` property contains the paths to the following compiled files:

- `magicsquare_1_0.dll`
- `GettingStarted.html`

Get Build Information from C Library

Create a C library and save information about the build type, compiled files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

```
results = compiler.build.cSharedLibrary('magicsquare.m')
```

```
results = 

  Results with properties:

              BuildType: 'cSharedLibrary'
                  Files: {4×1 cell}
IncludedSupportPackages: {}
                Options: [1×1 compiler.build.CSharedLibraryOptions]
    RuntimeDependencies: [1×1 compiler.runtime.Dependencies]
```

The `Files` property contains the paths to the following files:

- `magicsquare.dll`
- `magicsquare.lib`
- `magicsquare.h`
- `GettingStarted.html`

Get Build Information from C++ Library

Create a C++ library and save information about the build type, compiled files, support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

```
results = compiler.build.cppSharedLibrary('magicsquare.m')
```

```
results = 

  Results with properties:

              BuildType: 'cppSharedLibrary'
                  Files: {2×1 cell}
IncludedSupportPackages: {}
                Options: [1×1 compiler.build.CppSharedLibraryOptions]
    RuntimeDependencies: [1×1 compiler.runtime.Dependencies]
```

The `Files` property contains the paths to the `v2` folder and `GettingStarted.html`.

Get Build Information from .NET Assembly

Create a .NET assembly on a Windows system and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

```
results = compiler.build.dotNETAssembly('magicsquare.m')
```

```
results = 

  Results with properties:

              BuildType: 'dotNETAssembly'
                  Files: {4×1 cell}
IncludedSupportPackages: {}
                Options: [1×1 compiler.build.DotNETAssemblyOptions]
    RuntimeDependencies: [1×1 compiler.runtime.Dependencies]
```

The `Files` property contains the paths to the following compiled files:

- `magicsquare.dll`
- `magicsquareNative.dll`
- `magicsquare_overview.dll`
- `GettingStarted.html`

Get Build Information from Java Package

Create a Java package and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

```
results = compiler.build.javaPackage('magicsquare.m')
```

```
results = 

  Results with properties:

              BuildType: 'javaPackage'
                  Files: {3×1 cell}
IncludedSupportPackages: {}
                Options: [1×1 compiler.build.JavaPackageOptions]
    RuntimeDependencies: [1×1 compiler.runtime.Dependencies]
```

The `Files` property contains the paths to the following:

- `doc` folder
- `magicsquare.jar`
- `GettingStarted.html`

Get Build Information from Python Package

Create a Python package and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

```
results = compiler.build.pythonPackage('magicsquare.m');
results = 

  Results with properties:

            BuildType: 'pythonPackage'
                Files: {3×1 cell}
IncludedSupportPackages: {}
              Options: [1×1 compiler.build.PythonPackageOptions]
    RuntimeDependencies: [1×1 compiler.runtime.Dependencies]
```

The `Files` property contains the paths to the following:

- `example` folder
- `setup.py`
- `GettingStarted.html`

Get Build Information from Excel Add-In for MATLAB Production Server

Create an Excel add-in for MATLAB Production Server and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Build a MATLAB Production Server archive using the file `magicsquare.m`. Save the output as a `compiler.build.Results` object `serverBuildResults`.

```
serverBuildResults = compiler.build.productionServerArchive('magicsquare.m');
```

Build the Excel add-in using the `serverBuildResults` object.

```
results = compiler.build.excelClientForProductionServer(serverBuildResults)
```

```
results = 

  Results with properties:

              BuildType: 'excelClientForProductionServer'
                  Files: {1×1 cell}
IncludedSupportPackages: {}
                Options: [1×1 compiler.build.ExcelClientForProductionServerOptions]
    RuntimeDependencies: [1×1 compiler.runtime.Dependencies]
```

The `Files` property contains the paths to the following compiled files:

- `magicsquare.dll`
- `magicsquare.bas`
- `magicsquare.xla`

Note The files `magicsquare.bas` and `magicsquare.xla` are included in `Files` only if you enable the `'GenerateVisualBasicFile'` option in the `compiler.build.excelClientForProductionServer` command.

Version History

Introduced in R2020b

See Also

compiler.build.standaloneApplication | compiler.build.standaloneWindowsApplication | compiler.build.webAppArchive | compiler.build.productionServerArchive | compiler.build.excelAddIn | compiler.build.comComponent | compiler.build.cSharedLibrary | compiler.build.cppSharedLibrary | compiler.build.dotNETAssembly | compiler.build.javaPackage | compiler.build.pythonPackage | compiler.build.excelClientForProductionServer

compiler.build.standaloneApplication

Create standalone application for deployment outside MATLAB

Syntax

```
compiler.build.standaloneApplication(AppFile)
compiler.build.standaloneApplication(AppFile,Name,Value)
compiler.build.standaloneApplication(opts)
results = compiler.build.standaloneApplication( ___ )
```

Description

`compiler.build.standaloneApplication(AppFile)` creates a deployable standalone application using a MATLAB function, class, or app specified by `AppFile`. The executable type is determined by your operating system. The generated executable does not include MATLAB Runtime or an installer.

`compiler.build.standaloneApplication(AppFile,Name,Value)` creates a standalone application with additional options specified using one or more name-value arguments. Options include the executable name, help text, and icon image.

`compiler.build.standaloneApplication(opts)` creates a standalone application with additional options specified using a `compiler.build.StandaloneApplicationOptions` object `opts`. You cannot specify any other options using name-value arguments.

`results = compiler.build.standaloneApplication( ___ )` returns build information as a `compiler.build.Results` object using any of the argument combinations in previous syntaxes. The build information consists of the build type, paths to the compiled files, and build options.

Examples

Create Standalone Application

Create a standalone application using a function file that generates a magic square.

In MATLAB, locate the MATLAB code that you want to deploy as a standalone application. For this example, compile using the file `magicsquare.m`.

```
appFile = fullfile(which('magicsquare.m'));
```

Build a standalone application using the `compiler.build.standaloneApplication` command.

```
compiler.build.standaloneApplication(appFile);
```

This syntax generates the following files within a folder named `magicsquarestandaloneApplication` in your current working directory:

- `includedSupportPackages.txt` — Text file that lists all support files included in the application.
- `magicsquare.exe` or `magicsquare` — Executable file that has the `.exe` extension if compiled on a Windows system, or no extension if compiled on Linux or macOS systems.

- `run_magicsquare.sh` — Shell script file that sets the library path and executes the application. This file is only generated on Linux and macOS systems.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations on page 13-2.
- `readme.txt` — Text file that contains information on deployment prerequisites and the list of files to package for deployment.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

To run `magicsquare` from MATLAB with the input argument `4`, navigate to the `magicsquarestandaloneApplication` folder and execute one of the following commands based on your operating system:

Operating System	Test in MATLAB Command Window
Windows	`!magicsquare 4`
macOS	`system(['./run_magicsquare.sh ',matlabroot,' 4']);`
Linux	`!./magicsquare 4`

The application outputs a 4-by-4 magic square.

```
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

To run your standalone application outside of MATLAB, see “Run Standalone Application” on page 18-3.

Customize Standalone Application

Create a standalone application and customize it using name-value arguments.

Write a MATLAB function that uses a subfunction to compute the diagonal components of a magic square. Save the functions to files named `mymagicdiag.m` and `mydiag.m`.

```
function out = mymagicdiag(in)
X = magic(in);
out = mydiag(X);

function out = mydiag(in)
out = [diag(in)]';
```

Build the standalone application using `mymagicdiag.m`. Use name-value pair arguments to specify the executable name, add the `mydiag.m` function file, and interpret command line inputs as numeric doubles.

```
compiler.build.standaloneApplication('mymagicdiag.m', ...
'ExecutableName','MagicDiagApp', ...
'AdditionalFiles','mydiag.m', ...
'TreatInputsAsNumeric','On')
```

To run `MagicDiagApp` from MATLAB with the input argument `4`, navigate to the `MagicDiagAppstandaloneApplication` folder and execute one of the following commands based on your operating system:

Operating System	Test in MATLAB Command Window
Windows	`!MagicDiagApp 4`
macOS	`system(['./run_MagicDiagApp.sh ',matlabroot,' 4']);`
Linux	`!./MagicDiagApp 4`

The application outputs the diagonal entries of a 4-by-4 magic square.

```
    16    11     6     1
```

Create Multiple Applications Using Options Object

Create multiple standalone applications on a Windows system using a `compiler.build.StandaloneApplicationOptions` object.

Create a standalone application using the file `magicsquare.m`.

```
appFile = fullfile(which('magicsquare'));
```

Create a `StandaloneApplicationOptions` object using `appFile`. Use name-value arguments to specify a common output directory, interpret command line inputs as numeric doubles, and display progress information during the build process.

```
opts = compiler.build.StandaloneApplicationOptions(appFile,...
    'OutputDir','D:\Documents\MATLAB\work\MagicBatch',...
    'TreatInputsAsNumeric','On',...
    'Verbose','On')
```

```
opts = 

  StandaloneApplicationOptions with properties:

            CustomHelpTextFile: ''
              EmbedArchive: on
            ExecutableIcon: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\
            ExecutableName: 'magicsquare'
    ExecutableSplashScreen: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\
         ExecutableVersion: '1.0.0.0'
                   AppFile: 'C:\Program Files\MATLAB\R2025b\extern\examples\compiler\magicsquare
      TreatInputsAsNumeric: on
           AdditionalFiles: {}
        AutoDetectDataFiles: on
     ExternalEncryptionKey: [0×0 struct]
          ObfuscateArchive: off
           SecretsManifest: ''
           SupportPackages: {'autodetect'}
                   Verbose: on
                 OutputDir: 'D:\Documents\MATLAB\work\MagicBatch'
```

Build a standalone application by passing the `StandaloneApplicationOptions` object as an input to the build function.

```
compiler.build.standaloneApplication(opts);
```

To create a new standalone application using the function file `example2.m` with the same options, use dot notation to modify the `AppFile` of the existing `StandaloneApplicationOptions` object before running the build function again.

```
opts.AppFile = 'example2.m';
compiler.build.standaloneApplication(opts);
```

By modifying the `AppFile` argument and recompiling, you can create multiple applications using the same options object.

Get Build Information from Standalone Application

Create a standalone application and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

```
results = compiler.build.standaloneApplication('magicsquare.m')
```

```
results = 

              BuildType: 'standaloneApplication'
                  Files: {2×1 cell}
IncludedSupportPackages: {}
                Options: [1×1 compiler.build.StandaloneApplicationOptions]
    RuntimeDependencies: [1×1 compiler.runtime.Dependencies]
```

The `Files` property contains the paths to the `magicsquare` standalone executable and `readme.txt` files.

Input Arguments

`AppFile` — Path to main file
character vector | string scalar

Path to the main file used to build the application, specified as a row character vector or string scalar. The file must be a MATLAB function, class, or app of one of the following types: `.m`, `.p`, `.mlx`, `.mlapp`, or a valid MEX file.

Example: `'mymagic.m'`

Data Types: `char` | `string`

`opts` — Standalone application build options
`StandaloneApplicationOptions` object

Standalone application build options, specified as a `compiler.build.StandaloneApplicationOptions` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `OutputDir='D:\work\myproject'`

`AdditionalFiles` — Additional files
character vector | string scalar | cell array of character vectors | string array

Additional files and folders to include in the standalone application, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute.

Example: `'AdditionalFiles',["myvars.mat","myfunc.m"]`

Data Types: `char` | `string` | `cell`

`AutoDetectDataFiles` — Flag to automatically include data files
`'on'` (default) | on/off logical value

Flag to automatically include data files, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then data files that you provide as inputs to certain functions (such as `load` and `fopen`) are automatically included in the standalone application. This is the default behavior.
- If you set this property to `'off'`, then you must add data files to the application using the `AdditionalFiles` property.

Example: `'AutoDetectDataFiles','Off'`

Data Types: `logical`

`CustomHelpTextFile` — Path to help file
character vector | string scalar

Path to a help file containing help text for the end user of the application, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

Example: `'CustomHelpTextFile','D:\Documents\MATLAB\work\help.txt'`

Data Types: `char` | `string`

`EmbedArchive` — Flag to embed deployable archive
`'on'` (default) | on/off logical value

Flag to embed the deployable archive, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then the function embeds the archive in the deployable executable.
- If you set this property to `'off'`, then the function generates the deployable archive as a separate file.

Example: `'EmbedArchive','Off'`

Data Types: `logical`

`ExecutableIcon` — Path to icon image
character vector | string scalar

Path to the icon image, specified as a character vector or a string scalar. The image is used as the icon for the standalone executable. The path can be relative to the current working directory or absolute. Accepted image types are `.jpg`, `.jpeg`, `.png`, `.bmp`, and `.gif`.

The default path is:

`'`*`matlabroot`*`\toolbox\compiler\packagingResources\default_icon_48.png'`

Example: `'ExecutableIcon','D:\Documents\MATLAB\work\images\myIcon.png'`

Data Types: `char` | `string`

`ExecutableName` — Name of generated application
character vector | string scalar

Name of the generated application, specified as a character vector or a string scalar. The default value is the file name of `AppFile`. Target output names must begin with a letter or underscore character and contain only alpha-numeric characters or underscores.

Example: `'ExecutableName','MagicSquare'`

Data Types: `char` | `string`

`ExecutableSplashScreen` — Path to splash screen image
character vector | string scalar

Path to the splash screen image, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute. Accepted image types are `.jpg`, `.jpeg`, `.png`, `.bmp`, and `.gif`. The image is resized to 400 pixels by 400 pixels.

The default path is:

`'`*`matlabroot`*`\toolbox\compiler\packagingResources\default_splash.png'`

Note This is only used in Windows applications built using `compiler.build.standaloneWindowsApplication`.

Example: `'ExecutableSplashScreen','D:\Documents\MATLAB\work\images\mySplash.png'`

Data Types: `char` | `string`

`ExecutableVersion` — Executable version
`'1.0.0.0'` (default) | character vector | string scalar

Executable version, specified as a character vector or a string scalar.

Note This is only used on Windows operating systems.

Example: `'ExecutableVersion','4.0'`

Data Types: `char` | `string`

`ExternalEncryptionKey` — Paths to encryption key and loader files
scalar struct

Paths to the external AES encryption key and MEX key loader files, specified as a scalar struct with exactly two row char vector or string scalar fields named `EncryptionKeyFile` and `RuntimeKeyLoaderFile`, respectively. Both struct fields are required. File paths can be relative to the current working directory or absolute.

For example, specify the encryption key as `encrypt.key` and loader file as `loader.mexw64` using struct `keyValueStruct`.

```
keyValueStruct.EncryptionKeyFile='encrypt.key'; keyValueStruct.RuntimeKeyLoaderFile='loader.mexw64'
```

The encryption key file must be in one of the following supported formats:

- Binary 256-bit AES key, with a 32 byte file size
- Hex encoded AES key, with a 64 byte file size

The MEX file loader retrieves the decryption key at runtime and must be an interface with the following arguments:

- `prhs[0]` — Input, char array specified as the static value `'get'`
- `prhs[1]` — Input, char array specified as the CTF component UUID
- `plhs[0]` — Output, 32 byte UINT8 numeric array or 64 byte hex encoded char array, depending on the key format

Avoid sharing the same key across multiple CTFs.

Example: `'ExternalEncryptionKey',keyValueStruct`

Data Types: `struct`

`ObfuscateArchive` — Flag to obfuscate deployable archive
`'off'` (default) | on/off logical value

Flag to obfuscate the deployable archive, specified as `'on'`/`1`/`true` or `'off'`/`0`/`false`. The value is stored as an on/off logical value of type `matlab.lang.onoffSwitchState`.

If you set this property to `'on'`, then folder structures and file names in the deployable archive are obfuscated from the end user, and user code and data contained in `.m`, `.mlapp`, `.p`, `.mat`, MLX, SFX, and MEX files are placed into a user package within the archive. Additionally, all `.m` files are converted to P-files before packaging.

During runtime, MATLAB code and data is decrypted and loaded directly from the user package rather than extracted to the file system. MEX files are temporarily extracted from the user package before being loaded.

To manually include additional file types in the user package, add each file type in a separate extension tag to the file *`matlabroot`*`/toolbox/compiler/advanced_package_supported_files.xml`.

The following are **not** supported:

- `ver` function
- Calling external libraries such as DLLs
- Out-of-process MATLAB Runtime (C++ shared library for MATLAB Data Array)
- Out-of-process MEX file execution (`mexhost`, `feval`, `matlab.mex.MexHost`)
- `.mat` files other than v7.3

Enabling this option is equivalent to using `mcc` with `-j` and `-s` specified.

If you set this property to `'off'`, then the deployable archive is not obfuscated. This is the default behavior.

Example: `'ObfuscateArchive','on'`

Data Types: `logical`

`OutputDir` — Path to output directory
character vector | string scalar

Path to the output directory where the build files are saved, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

The default name of the build folder is the executable name appended with `standaloneApplication`.

Example: `'OutputDir','D:\Documents\MATLAB\work\MagicSquarestandaloneApplication'`

Data Types: `char` | `string`

`SecretsManifest` — Path to JSON manifest file
character vector | string scalar

Path to a secret manifest JSON file that specifies the secret keys to be embedded in the deployable archive, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

If your MATLAB code calls the `getSecret`, `getSecretMetadata`, or `isSecret` function, you must specify the secret keys to embed in the deployable archive in a JSON secret manifest file. If your code calls `getSecret` and you do not specify the `SecretsManifest` option, MATLAB Compiler issues a warning and generates a template JSON file in the output folder named `<`*`component_name`*`>_secrets_manifest.json`. Modify this file by specifying the secret key names in the **Embedded** field.

The `setSecret` function is not deployable. To embed secret keys in a deployable archive, you must call `setSecret` in MATLAB before you build the archive.

For more information on deployment using secrets, see "Handle Sensitive Information in Deployed Applications" on page 18-17.

Example: `'SecretsManifest','D:\Documents\MATLAB\work\mycomponent\mycomponent_secrets_manifest.json'`

Data Types: `char` | `string`

`SupportPackages` — Support packages
`'autodetect'` (default) | `'none'` | string scalar | cell array of character vectors | string array

Support packages to include, specified as one of the following options:

- `'autodetect'` (default) — The dependency analysis process detects and includes the required support packages automatically.
- `'none'` — No support packages are included. Using this option can cause runtime errors.
- A string scalar, character vector, or cell array of character vectors — Only the specified support packages are included. To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

Example: `'SupportPackages',{'Deep Learning Toolbox Converter for TensorFlow Models','Deep Learning Toolbox Model for Places365-GoogLeNet Network'}`

Data Types: `char` | `string` | `cell`

`TreatInputsAsNumeric` — Flag to interpret command line inputs
`'off'` (default) | on/off logical value

Flag to interpret command line inputs as numeric values, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then command line inputs are treated as numeric MATLAB doubles.
- If you set this property to `'off'`, then command line inputs are treated as MATLAB character vectors. This is the default behavior.

Example: `'TreatInputsAsNumeric','on'`

Data Types: `logical`

`Verbose` — Flag to control build verbosity
`'off'` (default) | on/off logical value

Flag to control build verbosity, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then the MATLAB command window displays progress information indicating compiler output during the build process.
- If you set this property to `'off'`, then the command window does not display progress information. This is the default behavior.

Example: `'Verbose','on'`

Data Types: `logical`

Output Arguments

`results` — Build results
`compiler.build.Results` object

Build results, returned as a `compiler.build.Results` object. The `Results` object contains:

- The build type, which is `'standaloneApplication'`
- Paths to the compiled files
- A list of included support packages
- Build options, specified as a `StandaloneApplicationOptions` object
- A list of required and optional dependencies, specified as a `Dependencies` object

Tips

- To create a standalone application from the system command prompt using this function, use the `matlab` function with the `-batch` option. For example:

  ```
  matlab -batch compiler.build.standaloneApplication('mymagic.m')
  ```

Version History

Introduced in R2020b

R2021b: Specify support packages

Use the `SupportPackages` option to specify support packages to include in the deployable code archive.

R2024b: Specify encryption key

Use the `ExternalEncryptionKey` option to specify a 256-bit AES encryption key and a MEX-file loader interface to retrieve the decryption key at runtime. This option is equivalent to the `mcc -k` option.

R2024b: Package code with secrets

Use the `SecretsManifest` option to include a JSON file that specifies secrets to embed within your deployable code archive. This option is equivalent to the `mcc -J` option.

R2023a: Obfuscate file structure and MATLAB code

Use the `ObfuscateArchive` option to obfuscate folder structures and file names, and place MATLAB file data and user code into a user package within the archive. Additionally, all `.m` files are converted to P-files before packaging. This option is equivalent to using `mcc` with `-j` and `-s` specified.

See Also

`compiler.build.StandaloneApplicationOptions` | `compiler.package.installer` | `compiler.build.standaloneWindowsApplication` | Standalone Application Compiler | `mcc`

Topics

"Create Standalone Application from MATLAB" on page 18-2

compiler.build.StandaloneApplicationOptions

Options for building standalone applications

Syntax

```
opts = compiler.build.StandaloneApplicationOptions(AppFile)
opts = compiler.build.standaloneApplicationOptions(AppFile,Name,Value)
```

Description

`opts = compiler.build.StandaloneApplicationOptions(AppFile)` creates a default standalone application options object using a MATLAB function, class, or app specified using `AppFile`. Use the `StandaloneApplicationOptions` object as an input to the `compiler.build.standaloneApplication` and `compiler.build.standaloneWindowsApplication` functions.

`opts = compiler.build.standaloneApplicationOptions(AppFile,Name,Value)` creates a standalone application options object with options specified using one or more name-value arguments.

Examples

Create Standalone Application Options Object

Create a `StandaloneApplicationOptions` object using file input.

For this example, use the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
opts = compiler.build.StandaloneApplicationOptions(appFile)
```

```
opts = 

  StandaloneApplicationOptions with properties:

        CustomHelpTextFile: ''
              EmbedArchive: on
            ExecutableIcon: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\
            ExecutableName: 'magicsquare'
    ExecutableSplashScreen: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\
         ExecutableVersion: '1.0.0.0'
                   AppFile: 'C:\Program Files\MATLAB\R2025b\extern\examples\compiler\magicsquare
     TreatInputsAsNumeric: off
           AdditionalFiles: {}
      AutoDetectDataFiles: on
    ExternalEncryptionKey: [0×0 struct]
          ObfuscateArchive: off
           SecretsManifest: ''
           SupportPackages: {'autodetect'}
                   Verbose: off
                 OutputDir: '.\magicsquarestandaloneApplication'
```

You can modify the property values of an existing `StandaloneApplictionOptions` object using dot notation. For example, enable verbose output.

```
opts.Verbose = 'on'
```

```
opts = 

  StandaloneApplicationOptions with properties:

        CustomHelpTextFile: ''
              EmbedArchive: on
            ExecutableIcon: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\
            ExecutableName: 'magicsquare'
    ExecutableSplashScreen: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\
         ExecutableVersion: '1.0.0.0'
                   AppFile: 'C:\Program Files\MATLAB\R2025b\extern\examples\compiler\magicsquare
      TreatInputsAsNumeric: off
           AdditionalFiles: {}
       AutoDetectDataFiles: on
     ExternalEncryptionKey: [0×0 struct]
          ObfuscateArchive: off
           SecretsManifest: ''
           SupportPackages: {'autodetect'}
                   Verbose: on
                 OutputDir: '.\magicsquarestandaloneApplication'
```

Use the `StandaloneApplicationOptions` object as an input to the `compiler.build.standaloneApplication` function to build a standalone application.

```
compiler.build.standaloneApplication(opts);
```

Customize a Standalone Application Options Object Using Name-Value Arguments

Create a `StandaloneApplictionOptions` object and customize it using name-value arguments.

Create a `StandaloneApplicationOptions` object using the function file `mymagic.m`. Use name-value arguments to specify the output directory, set the executable version and icon, and treat inputs as numeric values.

```
opts = compiler.build.StandaloneApplicationOptions('mymagic.m', ...
'OutputDir','D:\Documents\MATLAB\work\MagicApp', ...
'ExecutableIcon','D:\Documents\MATLAB\work\images\magicicon.png', ...
'ExecutableVersion','2.0','TreatInputsAsNumeric','On')
```

```
opts = 

  StandaloneApplicationOptions with properties:

        CustomHelpTextFile: ''
              EmbedArchive: on
            ExecutableIcon: 'D:\Documents\MATLAB\work\images\magicicon.png'
            ExecutableName: 'mymagic'
    ExecutableSplashScreen: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\
         ExecutableVersion: '2.0'
                   AppFile: 'D:\Documents\MATLAB\work\mymagic.m'
      TreatInputsAsNumeric: on
           AdditionalFiles: {}
       AutoDetectDataFiles: on
     ExternalEncryptionKey: [0×0 struct]
```

```
           ObfuscateArchive: off
            SecretsManifest: ''
            SupportPackages: {'autodetect'}
                    Verbose: off
                  OutputDir: 'D:\Documents\MATLAB\work\MagicApp'
```

You can modify the property values of an existing `StandaloneApplictionOptions` object using dot notation. For example, enable verbose output.

```
opts.Verbose = 'on';
```

Use the `StandaloneApplicationOptions` object as an input to the `compiler.build.standaloneApplication` function to build a standalone application.

```
compiler.build.standaloneApplication(opts);
```

Input Arguments

`AppFile` — Path to main file

character vector | string scalar

Path to the main file used to build the application, specified as a row character vector or a string scalar. The file must be a MATLAB function, class, or app of one of the following types: `.m`, `.p`, `.mlx`, `.mlapp`, or a valid MEX file.

Example: `'mymagic.m'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `OutputDir='D:\work\myproject'`

`AdditionalFiles` — Additional files

character vector | string scalar | cell array of character vectors | string array

Additional files and folders to include in the standalone application, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute.

Example: `'AdditionalFiles',["myvars.mat","myfunc.m"]`

Data Types: `char` | `string` | `cell`

`AutoDetectDataFiles` — Flag to automatically include data files

`'on'` (default) | on/off logical value

Flag to automatically include data files, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then data files that you provide as inputs to certain functions (such as `load` and `fopen`) are automatically included in the standalone application. This is the default behavior.
- If you set this property to `'off'`, then you must add data files to the application using the `AdditionalFiles` property.

Example: `'AutoDetectDataFiles','Off'`

Data Types: `logical`

`CustomHelpTextFile` — Path to help file
character vector | string scalar

Path to a help file containing help text for the end user of the application, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

Example: `'CustomHelpTextFile','D:\Documents\MATLAB\work\help.txt'`

Data Types: `char` | `string`

`EmbedArchive` — Flag to embed deployable archive
`'on'` (default) | on/off logical value

Flag to embed the deployable archive, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then the function embeds the archive in the deployable executable.
- If you set this property to `'off'`, then the function generates the deployable archive as a separate file.

Example: `'EmbedArchive','Off'`

Data Types: `logical`

`ExecutableIcon` — Path to icon image
character vector | string scalar

Path to the icon image, specified as a character vector or a string scalar. The image is used as the icon for the standalone executable. The path can be relative to the current working directory or absolute. Accepted image types are `.jpg`, `.jpeg`, `.png`, `.bmp`, and `.gif`.

The default path is:

```
'matlabroot\toolbox\compiler\packagingResources\default_icon_48.png'
```

Example: `'ExecutableIcon','D:\Documents\MATLAB\work\images\myIcon.png'`

Data Types: `char` | `string`

`ExecutableName` — Name of generated application
character vector | string scalar

Name of the generated application, specified as a character vector or a string scalar. The default value is the file name of `AppFile`. Target output names must begin with a letter or underscore character and contain only alpha-numeric characters or underscores.

Example: `'ExecutableName','MagicSquare'`

Data Types: `char` | `string`

`ExecutableSplashScreen` — Path to splash screen image
character vector | string scalar

Path to the splash screen image, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute. Accepted image types are `.jpg`, `.jpeg`, `.png`, `.bmp`, and `.gif`. The image is resized to 400 pixels by 400 pixels.

The default path is:

```
'matlabroot\toolbox\compiler\packagingResources\default_splash.png'
```

Note This is only used in Windows applications built using `compiler.build.standaloneWindowsApplication`.

Example: `'ExecutableSplashScreen','D:\Documents\MATLAB\work\images\mySplash.png'`

Data Types: `char` | `string`

`ExecutableVersion` — Executable version
`'1.0.0.0'` (default) | character vector | string scalar

Executable version, specified as a character vector or a string scalar.

Note This is only used on Windows operating systems.

Example: `'ExecutableVersion','4.0'`

Data Types: `char` | `string`

`ExternalEncryptionKey` — Paths to encryption key and loader files
scalar struct

Paths to the external AES encryption key and MEX key loader files, specified as a scalar struct with exactly two row char vector or string scalar fields named `EncryptionKeyFile` and `RuntimeKeyLoaderFile`, respectively. Both struct fields are required. File paths can be relative to the current working directory or absolute.

For example, specify the encryption key as `encrypt.key` and loader file as `loader.mexw64` using struct `keyValueStruct`.

```
keyValueStruct.EncryptionKeyFile='encrypt.key'; keyValueStruct.RuntimeKeyLoaderFile='loader.mexw64'
```

The encryption key file must be in one of the following supported formats:

- Binary 256-bit AES key, with a 32 byte file size

- Hex encoded AES key, with a 64 byte file size

The MEX file loader retrieves the decryption key at runtime and must be an interface with the following arguments:

- `prhs[0]` — Input, char array specified as the static value `'get'`
- `prhs[1]` — Input, char array specified as the CTF component UUID
- `plhs[0]` — Output, 32 byte UINT8 numeric array or 64 byte hex encoded char array, depending on the key format

Avoid sharing the same key across multiple CTFs.

Example: `'ExternalEncryptionKey',keyValueStruct`

Data Types: `struct`

`ObfuscateArchive` — Flag to obfuscate deployable archive
`'off'` (default) | on/off logical value

Flag to obfuscate the deployable archive, specified as `'on'/1/true` or `'off'/0/false`. The value is stored as an on/off logical value of type `matlab.lang.onoffSwitchState`.

If you set this property to `'on'`, then folder structures and file names in the deployable archive are obfuscated from the end user, and user code and data contained in `.m`, `.mlapp`, `.p`, `.mat`, MLX, SFX, and MEX files are placed into a user package within the archive. Additionally, all `.m` files are converted to P-files before packaging.

During runtime, MATLAB code and data is decrypted and loaded directly from the user package rather than extracted to the file system. MEX files are temporarily extracted from the user package before being loaded.

To manually include additional file types in the user package, add each file type in a separate extension tag to the file `*matlabroot*/toolbox/compiler/advanced_package_supported_files.xml`.

The following are **not** supported:

- `ver` function
- Calling external libraries such as DLLs
- Out-of-process MATLAB Runtime (C++ shared library for MATLAB Data Array)
- Out-of-process MEX file execution (`mexhost`, `feval`, `matlab.mex.MexHost`)
- `.mat` files other than v7.3

Enabling this option is equivalent to using `mcc` with `-j` and `-s` specified.

If you set this property to `'off'`, then the deployable archive is not obfuscated. This is the default behavior.

Example: `'ObfuscateArchive','on'`

Data Types: `logical`

`OutputDir` — Path to output directory
character vector | string scalar

Path to the output directory where the build files are saved, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

The default name of the build folder is the executable name appended with `standaloneApplication`.

Example: `'OutputDir','D:\Documents\MATLAB\work\MagicSquarestandaloneApplication'`

Data Types: `char` | `string`

`SecretsManifest` — Path to JSON manifest file
character vector | string scalar

Path to a secret manifest JSON file that specifies the secret keys to be embedded in the deployable archive, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

If your MATLAB code calls the `getSecret`, `getSecretMetadata`, or `isSecret` function, you must specify the secret keys to embed in the deployable archive in a JSON secret manifest file. If your code calls `getSecret` and you do not specify the `SecretsManifest` option, MATLAB Compiler issues a warning and generates a template JSON file in the output folder named `<`*`component_name`*`>_secrets_manifest.json`. Modify this file by specifying the secret key names in the **Embedded** field.

The `setSecret` function is not deployable. To embed secret keys in a deployable archive, you must call `setSecret` in MATLAB before you build the archive.

For more information on deployment using secrets, see “Handle Sensitive Information in Deployed Applications” on page 18-17.

Example: `'SecretsManifest','D:\Documents\MATLAB\work\mycomponent\mycomponent_secrets_manifest.json'`

Data Types: `char` | `string`

`SupportPackages` — Support packages
`'autodetect'` (default) | `'none'` | string scalar | cell array of character vectors | string array

Support packages to include, specified as one of the following options:

- `'autodetect'` (default) — The dependency analysis process detects and includes the required support packages automatically.
- `'none'` — No support packages are included. Using this option can cause runtime errors.
- A string scalar, character vector, or cell array of character vectors — Only the specified support packages are included. To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

Example: `'SupportPackages',{'Deep Learning Toolbox Converter for TensorFlow Models','Deep Learning Toolbox Model for Places365-GoogLeNet Network'}`

Data Types: `char` | `string` | `cell`

`TreatInputsAsNumeric` — Flag to interpret command line inputs
`'off'` (default) | on/off logical value

Flag to interpret command line inputs as numeric values, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then command line inputs are treated as numeric MATLAB doubles.
- If you set this property to `'off'`, then command line inputs are treated as MATLAB character vectors. This is the default behavior.

Example: `'TreatInputsAsNumeric','on'`

Data Types: `logical`

`Verbose` — Flag to control build verbosity
`'off'` (default) | on/off logical value

Flag to control build verbosity, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then the MATLAB command window displays progress information indicating compiler output during the build process.
- If you set this property to `'off'`, then the command window does not display progress information. This is the default behavior.

Example: `'Verbose','on'`

Data Types: `logical`

Output Arguments

`opts` — Standalone application options object
`StandaloneApplictionOptions` object

Standalone application build options, returned as a `StandaloneApplictionOptions` object.

Version History

Introduced in R2020b

R2021b: Specify support packages

Use the `SupportPackages` option to specify support packages to include in the deployable code archive.

R2024b: Specify encryption key

Use the `ExternalEncryptionKey` option to specify a 256-bit AES encryption key and a MEX-file loader interface to retrieve the decryption key at runtime. This option is equivalent to the `mcc -k` option.

R2024b: Package code with secrets

Use the `SecretsManifest` option to include a JSON file that specifies secrets to embed within your deployable code archive. This option is equivalent to the `mcc -J` option.

R2023a: Obfuscate file structure and MATLAB code

Use the `ObfuscateArchive` option to obfuscate folder structures and file names, and place MATLAB file data and user code into a user package within the archive. Additionally, all `.m` files are converted to P-files before packaging. This option is equivalent to using `mcc` with `-j` and `-s` specified.

See Also

`compiler.build.standaloneApplication` | `compiler.build.standaloneWindowsApplication` | Standalone Application Compiler | `mcc`

compiler.build.standaloneWindowsApplication

Create a standalone application for deployment outside MATLAB that does not launch a Windows command shell

Syntax

```
compiler.build.standaloneWindowsApplication(AppFile)
compiler.build.standaloneWindowsApplication(AppFile,Name,Value)
compiler.build.standaloneWindowsApplication(opts)
results = compiler.build.standaloneWindowsApplication( ___ )
```

Description

Caution This function is only supported on Windows operating systems.

`compiler.build.standaloneWindowsApplication(AppFile)` creates a standalone Windows only application using a MATLAB function, class, or app specified using `AppFile`. The application does not open a Windows command shell on execution, and as a result, no console output is displayed. The generated executable has a `.exe` file extension and does not include MATLAB Runtime or an installer.

`compiler.build.standaloneWindowsApplication(AppFile,Name,Value)` creates a standalone Windows application with additional options specified using one or more name-value arguments. Options include the executable name, version number, and icon and splash images.

`compiler.build.standaloneWindowsApplication(opts)` creates a standalone Windows application with additional options specified using a `compiler.build.StandaloneApplicationOptions` object `opts`. You cannot specify any other options using name-value arguments.

`results = compiler.build.standaloneWindowsApplication( ___ )` returns build information as a `compiler.build.Results` object using any of the argument combinations in previous syntaxes. The build information consists of the build type, paths to the compiled files, and build options.

Examples

Create Standalone Windows Application

Create a graphical standalone application on a Windows system that displays a plot.

Write a MATLAB function that plots the values `1` to `10`. Save the function in a file named `myPlot.m`.

```
function myPlot()
plot(1:10)
```

Build a standalone Windows application using the `compiler.build.standaloneWindowsApplication` command.

```
compiler.build.standaloneWindowsApplication('myPlot.m');
```

This syntax generates the following files within a folder named `myPlotstandaloneApplication` in your current working directory:

- `includedSupportPackages.txt` — Text file that lists all support files included in the application.
- `myPlot.exe` — Executable file.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For more information on non-supported functions, see MATLAB Compiler Limitations on page 13-2.
- `readme.txt` — Readme file that contains information on deployment prerequisites and the list of files to package for deployment.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `splash.png` — File that contains the splash image that displays when the application starts.
- `unresolvedSymbols.txt` — Text file that contains any unresolved symbols.

To run `myPlot.exe`, navigate to the `myPlotstandaloneApplication` folder and double-click `myPlot.exe` from the file browser, execute `!myPlot` in the MATLAB command window, or execute `myPlot.exe` in the Windows command shell.

The application displays a splash image followed by a MATLAB figure of a line plot.

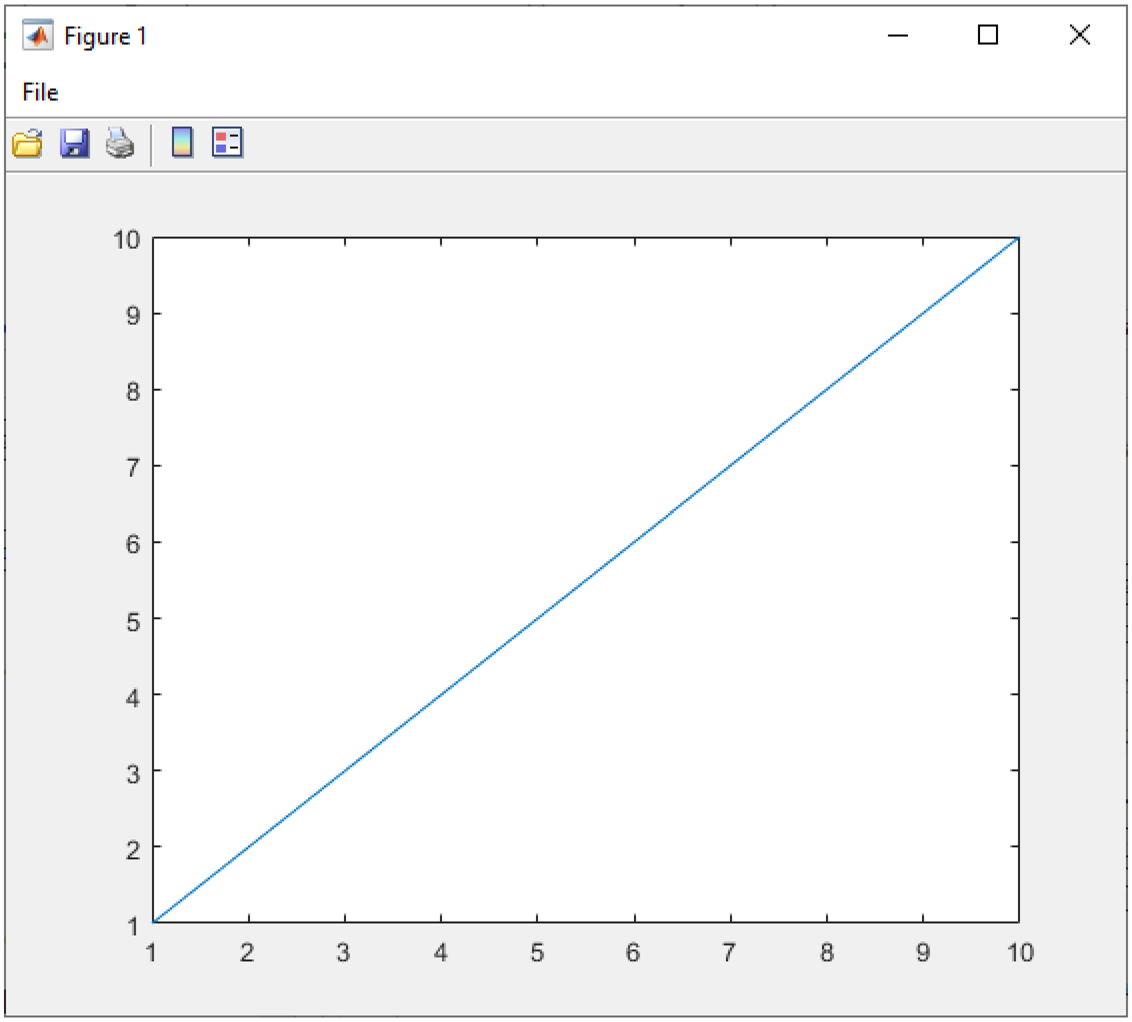


Figure 1 (myPlot.exe)

Customize Windows Application

Create a graphical standalone application on a Windows system and customize it using name-value arguments.

Create `xVal` as a vector of linearly spaced values between 0 and 2π. Use an increment of π/40 between the values. Create `yVal` as sine values of x. Save both variables in a MAT-file named `myVars.mat`.

```
xVal = 0:pi/40:2*pi;
yVal = sin(xVal);
save('myVars.mat','xVal','yVal');
```

Create a function file named `myPlot.m` to create a line plot of the `xVal` and `yVal` variables.

```
function myPlot()
load('myVars.mat');
plot(xVal,yVal)
```

Build the standalone application using the `compiler.build.standaloneWindowsApplication` function. Use name-value arguments to specify the executable name and version number.

```
compiler.build.standaloneWindowsApplication('myPlot.m', ...
'ExecutableName','SineWaveApp',...
'ExecutableVersion','2.0')
```

This syntax generates the following files within a folder named `SineWaveAppstandaloneApplication` in your current working directory:

- `includedSupportPackages.txt`
- `mccExcludedFiles.log`
- `readme.txt`
- `requiredMCRProducts.txt`
- `SineWaveApp.exe`
- `splash.png`
- `unresolvedSymbols.txt`

To run `SineWaveApp.exe`, navigate to the `myPlotstandaloneApplication` folder and double-click `SineWaveApp.exe` from the file browser, execute `!SineWaveApp.exe` in the MATLAB command window, or execute `SineWaveApp.exe` at the Windows command prompt.

The application displays a splash image followed by a MATLAB figure of a sine wave.

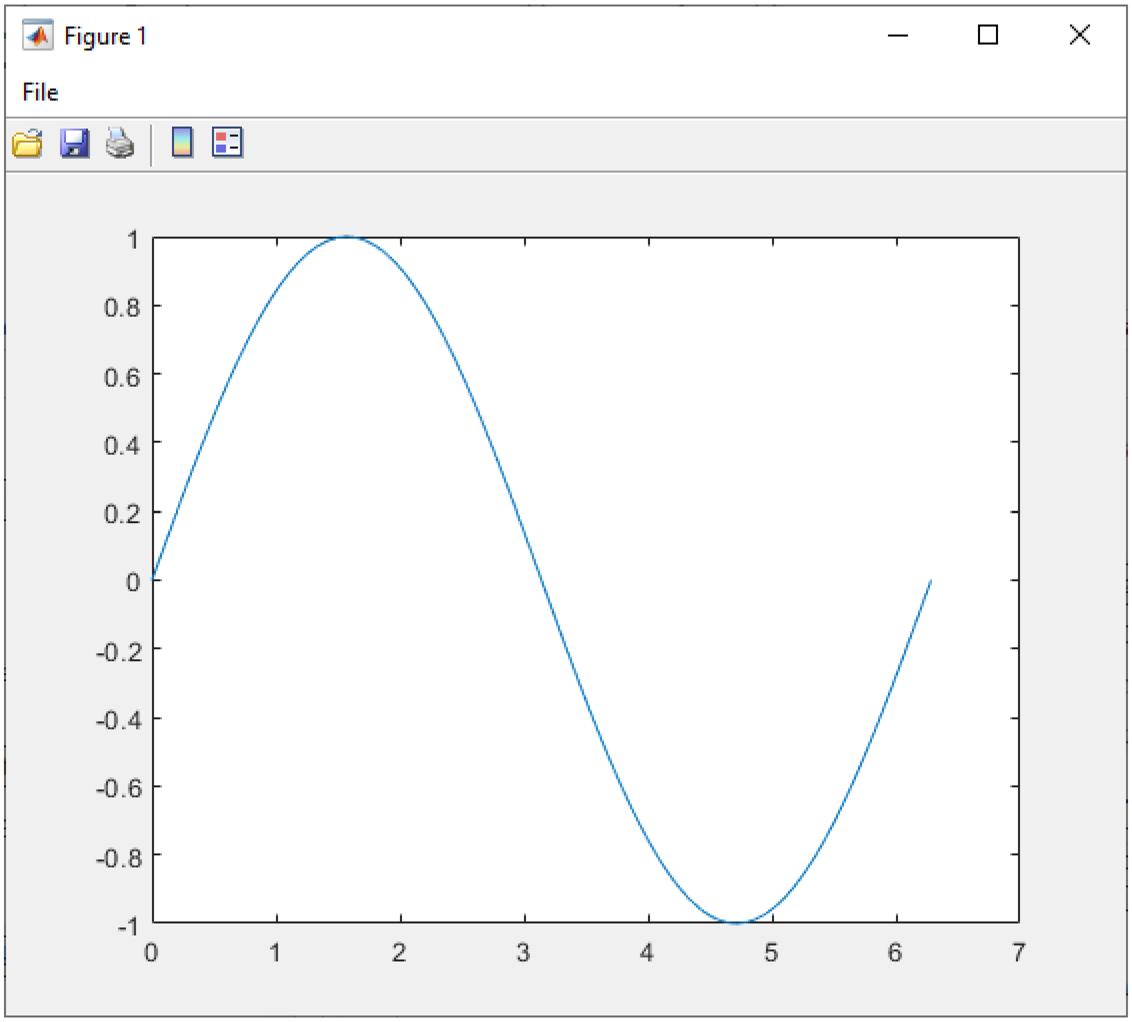


Figure 1 (SineWaveApp.exe)

Create Multiple Applications Using Options Object

Create multiple graphical standalone applications on a Windows system using a `compiler.build.StandaloneApplicationOptions` object.

Write a MATLAB function that plots the values `1` to `10`. Save the function in a file named `myPlot.m`.

```
function myPlot()
plot(1:10)
```

Create a `StandaloneApplicationOptions` object using `myPlot.m`. Use name-value arguments to specify a common output directory and display progress information during the build process.

```
opts = compiler.build.StandaloneApplicationOptions('myPlot.m', ...
'OutputDir','D:\Documents\MATLAB\work\WindowsApps', ...
'Verbose','On')
```

```
opts =
  StandaloneApplicationOptions with properties:

        CustomHelpTextFile: ''
              EmbedArchive: on
            ExecutableIcon: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\c
            ExecutableName: 'myPlot'
    ExecutableSplashScreen: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\c
         ExecutableVersion: '1.0.0.0'
                   AppFile: 'myPlot.m'
     TreatInputsAsNumeric: on
           AdditionalFiles: {}
        AutoDetectDataFiles: on
      ExternalEncryptionKey: [0×0 struct]
          ObfuscateArchive: off
           SecretsManifest: ''
           SupportPackages: {'autodetect'}
                   Verbose: on
                 OutputDir: 'D:\Documents\MATLAB\work\WindowsApps'
```

Build a graphical standalone application by passing the `StandaloneApplicationOptions` object as an input to the build function.

```
compiler.build.standaloneWindowsApplication(opts);
```

To create a new application using the function file `myPlot2.m` with the same options, use dot notation to modify the `AppFile` of the existing `StandaloneApplicationOptions` object before running the build function again.

```
opts.AppFile = 'example2.m';
compiler.build.standaloneWindowsApplication(opts);
```

By modifying the `AppFile` argument and recompiling, you can compile multiple applications using the same options object.

Get Build Information from Standalone Windows Application

Create a standalone Windows application on a Windows system and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `Mortgage.mlapp`.

```
results = compiler.build.standaloneWindowsApplication('Mortgage.mlapp')
```

```
results = 

  Results with properties:

              BuildType: 'standaloneWindowsApplication'
                  Files: {3×1 cell}
IncludedSupportPackages: {}
                Options: [1×1 compiler.build.StandaloneApplicationOptions]
    RuntimeDependencies: [1×1 compiler.runtime.Dependencies]
```

The `Files` property contains the paths to the following files:

- `Mortgage.exe`

- `splash.png`
- `readme.txt`

Input Arguments

`AppFile` — Path to main file
character vector | string scalar

Path to the main file used to build the application, specified as a row character vector or a string scalar. The file must be a MATLAB function, class, or app of one of the following types: `.m`, `.p`, `.mlx`, `.mlapp`, or a valid MEX file.

Example: `'mymagic.m'`

Data Types: `char` | `string`

`opts` — Standalone application build options
`StandaloneApplicationOptions` object

Standalone application build options, specified as a `compiler.build.StandaloneApplicationOptions` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `OutputDir='D:\work\myproject'`

`AdditionalFiles` — Additional files
character vector | string scalar | cell array of character vectors | string array

Additional files and folders to include in the standalone application, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute.

Example: `'AdditionalFiles',["myvars.mat","myfunc.m"]`

Data Types: `char` | `string` | `cell`

`AutoDetectDataFiles` — Flag to automatically include data files
`'on'` (default) | on/off logical value

Flag to automatically include data files, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then data files that you provide as inputs to certain functions (such as `load` and `fopen`) are automatically included in the standalone application. This is the default behavior.

- If you set this property to `'off'`, then you must add data files to the application using the `AdditionalFiles` property.

Example: `'AutoDetectDataFiles','Off'`

Data Types: `logical`

`CustomHelpTextFile` — Path to help file
character vector | string scalar

Path to a help file containing help text for the end user of the application, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

Example: `'CustomHelpTextFile','D:\Documents\MATLAB\work\help.txt'`

Data Types: `char` | `string`

`EmbedArchive` — Flag to embed deployable archive
`'on'` (default) | on/off logical value

Flag to embed the deployable archive, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then the function embeds the archive in the deployable executable.
- If you set this property to `'off'`, then the function generates the deployable archive as a separate file.

Example: `'EmbedArchive','Off'`

Data Types: `logical`

`ExecutableIcon` — Path to icon image
character vector | string scalar

Path to the icon image, specified as a character vector or a string scalar. The image is used as the icon for the standalone executable. The path can be relative to the current working directory or absolute. Accepted image types are `.jpg`, `.jpeg`, `.png`, `.bmp`, and `.gif`.

The default path is:

`'`*`matlabroot`*`\toolbox\compiler\packagingResources\default_icon_48.png'`

Example: `'ExecutableIcon','D:\Documents\MATLAB\work\images\myIcon.png'`

Data Types: `char` | `string`

`ExecutableName` — Name of generated application
character vector | string scalar

Name of the generated application, specified as a character vector or a string scalar. The default value is the file name of `AppFile`. Target output names must begin with a letter or underscore character and contain only alpha-numeric characters or underscores.

Example: `'ExecutableName','MagicSquare'`

Data Types: `char` | `string`

`ExecutableSplashScreen` — Path to splash screen image
character vector | string scalar

Path to the splash screen image, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute. Accepted image types are `.jpg`, `.jpeg`, `.png`, `.bmp`, and `.gif`. The image is resized to 400 pixels by 400 pixels.

The default path is:

'*matlabroot*\toolbox\compiler\packagingResources\default_splash.png'

Note This is only used in Windows applications built using `compiler.build.standaloneWindowsApplication`.

Example: `'ExecutableSplashScreen','D:\Documents\MATLAB\work\images\mySplash.png'`

Data Types: `char` | `string`

`ExecutableVersion` — Executable version
`'1.0.0.0'` (default) | character vector | string scalar

Executable version, specified as a character vector or a string scalar.

Note This is only used on Windows operating systems.

Example: `'ExecutableVersion','4.0'`

Data Types: `char` | `string`

`ExternalEncryptionKey` — Paths to encryption key and loader files
scalar struct

Paths to the external AES encryption key and MEX key loader files, specified as a scalar struct with exactly two row char vector or string scalar fields named `EncryptionKeyFile` and `RuntimeKeyLoaderFile`, respectively. Both struct fields are required. File paths can be relative to the current working directory or absolute.

For example, specify the encryption key as `encrypt.key` and loader file as `loader.mexw64` using struct `keyValueStruct`.

```
keyValueStruct.EncryptionKeyFile='encrypt.key'; keyValueStruct.RuntimeKeyLoaderFile='loader.mexw64'
```

The encryption key file must be in one of the following supported formats:

- Binary 256-bit AES key, with a 32 byte file size
- Hex encoded AES key, with a 64 byte file size

The MEX file loader retrieves the decryption key at runtime and must be an interface with the following arguments:

- `prhs[0]` — Input, char array specified as the static value `'get'`
- `prhs[1]` — Input, char array specified as the CTF component UUID
- `plhs[0]` — Output, 32 byte UINT8 numeric array or 64 byte hex encoded char array, depending on the key format

Avoid sharing the same key across multiple CTFs.

Example: `'ExternalEncryptionKey',keyValueStruct`

Data Types: `struct`

`ObfuscateArchive` — Flag to obfuscate deployable archive
`'off'` (default) | on/off logical value

Flag to obfuscate the deployable archive, specified as `'on'`/`1`/`true` or `'off'`/`0`/`false`. The value is stored as an on/off logical value of type `matlab.lang.onoffSwitchState`.

If you set this property to `'on'`, then folder structures and file names in the deployable archive are obfuscated from the end user, and user code and data contained in `.m`, `.mlapp`, `.p`, `.mat`, MLX, SFX, and MEX files are placed into a user package within the archive. Additionally, all `.m` files are converted to P-files before packaging.

During runtime, MATLAB code and data is decrypted and loaded directly from the user package rather than extracted to the file system. MEX files are temporarily extracted from the user package before being loaded.

To manually include additional file types in the user package, add each file type in a separate extension tag to the file `*matlabroot*/toolbox/compiler/advanced_package_supported_files.xml`.

The following are **not** supported:

- `ver` function
- Calling external libraries such as DLLs
- Out-of-process MATLAB Runtime (C++ shared library for MATLAB Data Array)
- Out-of-process MEX file execution (`mexhost`, `feval`, `matlab.mex.MexHost`)
- `.mat` files other than v7.3

Enabling this option is equivalent to using `mcc` with `-j` and `-s` specified.

If you set this property to `'off'`, then the deployable archive is not obfuscated. This is the default behavior.

Example: `'ObfuscateArchive','on'`

Data Types: `logical`

`OutputDir` — Path to output directory
character vector | string scalar

Path to the output directory where the build files are saved, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

The default name of the build folder is the executable name appended with `standaloneApplication`.

Example: `'OutputDir','D:\Documents\MATLAB\work\MagicSquarestandaloneApplication'`

Data Types: `char` | `string`

`SecretsManifest` — Path to JSON manifest file
character vector | string scalar

Path to a secret manifest JSON file that specifies the secret keys to be embedded in the deployable archive, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

If your MATLAB code calls the `getSecret`, `getSecretMetadata`, or `isSecret` function, you must specify the secret keys to embed in the deployable archive in a JSON secret manifest file. If your code calls `getSecret` and you do not specify the `SecretsManifest` option, MATLAB Compiler issues a warning and generates a template JSON file in the output folder named `<`*`component_name`*`>_secrets_manifest.json`. Modify this file by specifying the secret key names in the **Embedded** field.

The `setSecret` function is not deployable. To embed secret keys in a deployable archive, you must call `setSecret` in MATLAB before you build the archive.

For more information on deployment using secrets, see “Handle Sensitive Information in Deployed Applications” on page 18-17.

Example: `'SecretsManifest','D:\Documents\MATLAB\work\mycomponent\mycomponent_secrets_manifest.json'`

Data Types: `char` | `string`

`SupportPackages` — Support packages
`'autodetect'` (default) | `'none'` | string scalar | cell array of character vectors | string array

Support packages to include, specified as one of the following options:

- `'autodetect'` (default) — The dependency analysis process detects and includes the required support packages automatically.
- `'none'` — No support packages are included. Using this option can cause runtime errors.
- A string scalar, character vector, or cell array of character vectors — Only the specified support packages are included. To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

Example: `'SupportPackages',{'Deep Learning Toolbox Converter for TensorFlow Models','Deep Learning Toolbox Model for Places365-GoogLeNet Network'}`

Data Types: `char` | `string` | `cell`

`TreatInputsAsNumeric` — Flag to interpret command line inputs
`'off'` (default) | on/off logical value

Flag to interpret command line inputs as numeric values, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then command line inputs are treated as numeric MATLAB doubles.
- If you set this property to `'off'`, then command line inputs are treated as MATLAB character vectors. This is the default behavior.

Example: `'TreatInputsAsNumeric','on'`

Data Types: `logical`

`Verbose` — Flag to control build verbosity
`'off'` (default) | on/off logical value

Flag to control build verbosity, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then the MATLAB command window displays progress information indicating compiler output during the build process.
- If you set this property to `'off'`, then the command window does not display progress information. This is the default behavior.

Example: `'Verbose','on'`

Data Types: `logical`

Output Arguments

`results` — Build results
`compiler.build.Results` object

Build results, returned as a `compiler.build.Results` object. The `Results` object contains:

- The build type, which is `'standaloneApplication'`
- Paths to the following files:
  - `*ExecutableName*.exe`
  - `splash.png`
  - `readme.txt`
- A list of included support packages
- Build options, specified as a `StandaloneApplicationOptions` object

Limitations

- This function is only supported on Windows operating systems.
- The application does not open a Windows command shell on execution, and as a result, no console output is displayed.

Tips

- To create a Windows standalone application from the system command prompt using this function, use the `matlab` function with the `-batch` option. For example:

```
matlab -batch compiler.build.standaloneWindowsApplication('myapp.mlapp')
```

Version History

Introduced in R2020b

R2021b: Specify support packages

Use the `SupportPackages` option to specify support packages to include in the deployable code archive.

R2024b: Specify encryption key

Use the `ExternalEncryptionKey` option to specify a 256-bit AES encryption key and a MEX-file loader interface to retrieve the decryption key at runtime. This option is equivalent to the `mcc -k` option.

R2024b: Package code with secrets

Use the `SecretsManifest` option to include a JSON file that specifies secrets to embed within your deployable code archive. This option is equivalent to the `mcc -J` option.

R2023a: Obfuscate file structure and MATLAB code

Use the `ObfuscateArchive` option to obfuscate folder structures and file names, and place MATLAB file data and user code into a user package within the archive. Additionally, all `.m` files are converted to P-files before packaging. This option is equivalent to using `mcc` with `-j` and `-s` specified.

See Also

`compiler.build.standaloneApplication` | `compiler.build.StandaloneApplicationOptions` | `compiler.package.installer` | Standalone Application Compiler | `mcc`

compiler.codetools.deployableSupportPackages

Determine support packages used by files

Syntax

```
spstr = compiler.codetools.deployableSupportPackages
spstr = compiler.codetools.deployableSupportPackages(Files)
```

Description

`spstr = compiler.codetools.deployableSupportPackages` returns a list of all support packages usable by MATLAB Compiler.

`spstr = compiler.codetools.deployableSupportPackages(Files)` returns a list of all support packages used by `Files`.

Examples

List Installed Deployable Support Packages

List all the installed deployable support packages on the system.

Run the function with no arguments.

```
spstr = compiler.codetools.deployableSupportPackages
```

```
spstr = 

  4×1 string array

    "Aerospace Ephemeris Data"
    "Aerospace Geoid Data"
    "Communications Toolbox Support Package for RTL-SDR Radio"
    "MATLAB Support Package for Raspberry Pi Hardware"
```

List Deployable Support Packages Used by File

List all deployable support packages that are used by the specified file.

Install the support packages required by your MATLAB file. For this example, the **Ephemeris Data for Aerospace Toolbox** add-on is installed.

Run the function using `planetEphemeris.m`, which is located in `*matlabroot*\toolbox\aero\aero`.

```
spstr = compiler.codetools.deployableSupportPackages(...
   fullfile(matlabroot,'toolbox','aero','aero','planetEphemeris.m'))
```

```
spstr = 

    "Aerospace Ephemeris Data"
```

Input Arguments

`Files` — List of files
string scalar | cell array of character vectors | string array

List of files, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Each file must be a MATLAB function, class, or app of one of the following types: `.m`, `.p`, `.mlx`, `.mlapp`, or a valid MEX file.

Example: `{'function1.m',function2.m'}`

Data Types: `char` | `string` | `cell`

Output Arguments

`spstr` — List of support package names
string array

A list of support package names, specified as a string array.

Data Types: `string`

Version History

Introduced in R2021b

See Also

`compiler.build.Results`

compiler.runtime.customInstaller

Create a MATLAB Runtime installer for the specified files generated by MATLAB Compiler

Syntax

```
compiler.runtime.customInstaller(installerName, results)
compiler.runtime.customInstaller(installerName, filepath)
compiler.runtime.customInstaller( ___ ,Name=Value)
```

Description

`compiler.runtime.customInstaller(installerName, results)` creates a MATLAB Runtime installer with a minimal size footprint that installs only the MATLAB Runtime components required to run the specified artifacts created with MATLAB Compiler or MATLAB Compiler SDK. You specify the artifacts using `results`, which is a vector of one or more `compiler.build.Results` objects created by any `compiler.build` function.

`compiler.runtime.customInstaller(installerName, filepath)` creates a MATLAB Runtime installer using a list of `buildresult.json` files generated by MATLAB Compiler.

`compiler.runtime.customInstaller( ___ ,Name=Value)` creates a MATLAB Runtime installer with additional options specified as one or more name-value arguments. Options include the output folder, package type, and runtime delivery method.

Examples

Create MATLAB Runtime Installer Using Build Results

Create a minimal MATLAB Runtime installer that installs all MATLAB Runtime components required to run a standalone application.

Write a MATLAB function to package into a standalone application. For this example, compile using the file `magicsquare.m`.

```
copyfile(fullfile(matlabroot,"extern","examples","compiler","magicsquare.m"));
```

Create a standalone application using `compiler.build.standaloneApplication`. Save the output from the function as a `compiler.build.Results` object.

```
results = compiler.build.standaloneApplication("magicsquare.m");
```

Create a MATLAB Runtime installer that contains the components required to run the standalone application.

```
compiler.runtime.customInstaller("magicsquareInstaller",results);
```

Deploy the standalone application and generated MATLAB Runtime installer on the target machine.

The installed MATLAB Runtime takes up less space than a full MATLAB Runtime installation.

Create MATLAB Runtime Installer on Offline Machine

Create a minimal MATLAB Runtime installer on an offline machine by using the full MATLAB Runtime installer.

On the offline machine, use the `compiler.runtime.download` function to obtain the URL to the MATLAB Runtime installer at the matching version and update level.

```
compiler.runtime.download
```

```
Downloading MATLAB Runtime installer. It may take several minutes...

Error using compiler.runtime.download
A connection could not be established to download the Runtime Installer.
Download the runtime from:
https://ssd.mathworks.com/supportfiles/downloads/R2025b/Release/0/deployment_files/installer/comp
and update the runtime location in Compiler Settings.
```

Using a machine that is connected to the Internet, download the installer using the URL provided and transfer the installer to the offline machine.

On the offline machine, in MATLAB, open the **Settings** menu and select **MATLAB Compiler**. Specify the location of the MATLAB Runtime installer.

On the offline machine, write a MATLAB function to package into a standalone application. For this example, compile using the file `magicsquare.m`.

```
copyfile(fullfile(matlabroot,"extern","examples","compiler","magicsquare.m"));
```

Create a standalone application using `compiler.build.standaloneApplication`. Save the output from the function as a `compiler.build.Results` object.

```
results = compiler.build.standaloneApplication("magicsquare.m");
```

Create a MATLAB Runtime installer that contains the components required to run the standalone application.

```
compiler.runtime.customInstaller("magicsquareInstaller",results);
```

Deploy the standalone application and generated MATLAB Runtime installer on the target machine.

Create MATLAB Runtime Installer Using Files

Create a MATLAB Runtime installer on Windows using the `buildresult.json` files generated by MATLAB Compiler.

Write a MATLAB function to package into a standalone Windows application. For this example, copy the MATLAB script `flames.m` and data file `flames.mat` to your current working directory.

```
copyfile(fullfile(matlabroot,"extern","examples","compiler","flames.*"));
```

Use `mcc` to create a standalone Windows application using `flames.m`.

```
mcc("flames.m","-e","-a","flames.mat","-d","flames")
```

Create an Excel add-in using the file `houdini.m`.

```
copyfile(fullfile(matlabroot,"extern","examples","compiler","houdini.m"));
compiler.build.excelAddIn("houdini.m","GenerateVisualBasicFile","on","OutputDir","houdini");
```

Create a MATLAB Runtime installer that contains the components required to run both the standalone application and the Excel add-in.

```
compiler.runtime.customInstaller("flames_houdiniInstaller",...
["flames\buildresult.json","houdini\buildresult.json"]);
```

A MATLAB Runtime installer named `flames_houdiniInstaller.exe` is generated in the folder `flames_houdiniInstaller`.

Customize MATLAB Runtime Installer Using Name-Value Arguments

Create a MATLAB Runtime installer that installs all MATLAB Runtime components required to run multiple C++ shared libraries.

For this example, copy the matrix folder that ships with MATLAB to your work folder.

```
copyfile(fullfile(matlabroot,"extern","examples","compilersdk","c_cpp","matrix"),"matrix")
```

Navigate to the new `matrix` subfolder in your work folder.

Create two C++ shared libraries using `compiler.build.cppSharedLibrary`.

```
functionfiles = {"addmatrix.m", "multiplymatrix.m", "eigmatrix.m"};

results1 = compiler.build.cppSharedLibrary(functionfiles,...
"OutputDir","matrixLibraries");

results2 = compiler.build.cppSharedLibrary("subtractmatrix.m",...
"OutputDir","matrixLibraries");
```

Create a MATLAB Runtime installer that can be used for all libraries. Use name-value arguments to specify the build folder and embed MATLAB Runtime in the installer for offline deployment.

```
compiler.runtime.customInstaller("matrixInstaller",[results1,results2],...
OutputDir="customInstallers",RuntimeDelivery="installer")
```

The installer `matrixInstaller.zip` is generated in the `customInstallers` folder in the current working directory.

Deploy the MATLAB Runtime installer and shared library files (`.ctf` and `.hpp`) to the target machine.

Input Arguments

`installerName` — Name of installer file
character vector | string scalar

Name of the installer file, specified as a character vector or a string scalar. The extension is determined by the operating system in which the function is executed.

Example: `"MagicSquare_Installer"`

`results` — Build results
vector of `compiler.build.Results` objects

One or more build results created by a `compiler.build` function, such as `compiler.build.standaloneApplication` or `compiler.build.productionServerArchive`, specified as a vector of `compiler.build.Results` objects.

Example: `[results1,results2]`

`filepath` — Paths to `buildresult.json`
character vector | string scalar | string array | cell array of strings

One or more paths to `buildresult.json`, which is generated by MATLAB Compiler or MATLAB Compiler SDK.

Example: `["fun1/buildresult.json","fun2/buildresult.json"]`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `PackageType="zip"`

`OptionalDependencies` — Optional dependencies to include
`"all"` (default) | `"none"`

Optional MATLAB Runtime dependencies to include in the installer, specified as one of the following:

- `"all"` — Option to include all of the optional dependencies in the installer, including graphical support. This option is the default behavior.
- `"none"` — Option to not include any of the optional dependencies in the installer. Use this option to minimize the size of the generated installer.

Example: `OptionalDependencies="none"`

`OutputDir` — Path to output directory
character vector | string scalar

Path to the output directory where the installer is saved, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

The default name of the output folder is the value of `installerName`.

Example: `OutputDir="D:\Documents\MATLAB\work\myappInstaller"`

Data Types: `char` | `string`

`PackageType` — Installer file type
`"auto"` (default) | `"zip"`

Installer file type, specified as one of the following options.

- `"auto"`—Option for installer to automatically select the suitable file type when creating the installer. If the installer size is 2GB or greater, the installer is packaged as a ZIP file. This is the default option.
- `"zip"`—Option to always generate a ZIP file as an output when creating a new installer. This option is only supported when `RuntimeDelivery` is set to `"installer"`. This option is only supported on Windows systems.

Example: `PackageType="zip"`

Data Types: `char` | `string`

`RuntimeDelivery` — MATLAB Runtime delivery method
`"web"` (default) | `"installer"`

MATLAB Runtime delivery method, specified as one of the following options.

- `"web"`—Option for installer to download MATLAB Runtime from the MathWorks website during application installation. This is the default option.
- `"installer"`—Option to include MATLAB within the installer so that it can be installed during application installation without connecting to the MathWorks website. Use this option if you think your end user may not have access to the Internet.

Example: `RuntimeDelivery="installer"`

Data Types: `char` | `string`

Tips

- The installer created by this function installs a subset of MATLAB Runtime components with a minimal size footprint. To download the full MATLAB Runtime installer, see `compiler.runtime.download`.
- If you use multiple installers created by this function, MATLAB Runtime is updated to contain the minimal subset of components across all of the installers. This means that you can run multiple MATLAB Compiler generated artifacts with the same instance of MATLAB Runtime by using custom installers created from each build.
- The installer created by this function does not install your MATLAB Compiler generated artifacts. To create an installer for packaged MATLAB code, see `compiler.package.installer`.

Version History

Introduced in R2024b

R2025a: Optional dependencies

You can use the `OptionalDependencies` option to control whether to include all of the optional MATLAB Runtime dependencies in the installer.

R2025a: Build using `buildresult.json`

You can use the JSON file named `buildresult.json` in place of `requiredMCRProducts.txt` in deployment workflows. The `requiredMCRProducts.txt` file will be removed in a future release.

See Also

`compiler.package.installer` | `compiler.build.Results` | `mcc`

Topics

"About MATLAB Runtime" on page 7-2

compiler.package.docker

Create a Docker image for files generated by MATLAB Compiler on Linux operating systems

Note

- This function is only supported on Linux® operating systems.
- Only standalone applications can be packaged into Docker® images as of R2020b.

Syntax

```
compiler.package.docker(results)
compiler.package.docker(results,Name,Value)
compiler.package.docker(results,'Options',opts)
compiler.package.docker(files,filepath,'ImageName',imageName)
compiler.package.docker(files,filepath,'ImageName',imageName,Name,Value)
compiler.package.docker(files,filepath,'Options',opts)
```

Description

`compiler.package.docker(results)` creates a Docker image for files generated by the MATLAB Compiler using the `compiler.build.Results` object `results`. The results object is created by a `compiler.build` function.

`compiler.package.docker(results,Name,Value)` creates a Docker image using the `compiler.build.Results` object `results` and additional options specified as one or more name-value pairs. Options include the build folder, entry point command, and image name.

`compiler.package.docker(results,'Options',opts)` creates a Docker image using the `compiler.build.Results` object `results` and additional options specified by a `DockerOptions` object `opts`. If you use a `DockerOptions` object, you cannot specify any other options using name-value pairs.

`compiler.package.docker(files,filepath,'ImageName',imageName)` creates a Docker image using `files` that are generated by the MATLAB Compiler. The Docker image name is specified by `imageName`.

`compiler.package.docker(files,filepath,'ImageName',imageName,Name,Value)` creates a Docker image using `files` that are generated by the MATLAB Compiler. The Docker image name is specified by `imageName`. Additional options are specified as one or more name-value pairs.

`compiler.package.docker(files,filepath,'Options',opts)` creates a Docker image using `files` that are generated by the MATLAB Compiler and additional options specified by a `DockerOptions` object `opts`. If you use a `DockerOptions` object, you cannot specify any other options using name-value pairs.

Examples

Create Docker Image Using Results

Create a Docker image from a standalone application on a Linux system.

Install and configure Docker on your system.

Create a standalone application using `magicsquare.m` and save the build results to a `compiler.build.Results` object.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
buildResults = compiler.build.standaloneApplication(appFile);
```

Pass the `Results` object as an input to the `compiler.package.docker` function to build the Docker image.

```
compiler.package.docker(buildResults);
```

Customize Docker Image Using Results and Name Value Arguments

Customize a standalone Docker image using name-value pairs on a Linux system to specify the image name and build directory.

Create a standalone application using `magicsquare.m` and save the build results to a `compiler.build.Results` object.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
buildResults = compiler.build.standaloneApplication(appFile);
```

Build a Docker image using the `Results` object and specify additional options as name-value arguments.

```
compiler.package.docker(buildResults,...
'ImageName','mymagicapp',...
'DockerContext','/home/mluser/Documents/MATLAB/docker');
```

Customize Docker Image Using Results and Options Object

Customize a Docker image using a `DockerOptions` object on a Linux system.

Create a standalone application using `magicsquare.m` and save the build results to a `compiler.build.Results` object.

```
buildResults = compiler.build.standaloneApplication('magicsquare.m');
```

Create a `DockerOptions` object to specify additional build options, such as setting a custom image name and disabling the Docker build command.

```
opts = compiler.package.DockerOptions(buildResults,...
'ImageName','magicdocker');
```

You can modify property values of an existing `DockerOptions` object using dot notation. For example, populate the `DockerContext` folder without calling 'docker build'.

```
opts.ExecuteDockerBuild = 'Off';
```

Pass the `DockerOptions` and `Results` objects as inputs to the `compiler.package.docker` function to build the Docker image.

```
compiler.package.docker(buildResults,'Options',opts);
```

Create Docker Image Using Files and Name Value Arguments

Create a Docker image using files generated by MATLAB Compiler and specify the image name on a Linux system.

Build a standalone application using the `mcc` command.

```
mcc -o runhoudini -m houdini.m
```

Build the Docker image by passing the generated files to the `compiler.package.docker` function.

```
compiler.package.docker('runhoudini','buildresult.json',...
'ImageName','launchapp','EntryPoint','runhoudini');
```

Customize Docker Image Using Files and Options Object

Customize a Docker image using files generated by MATLAB Compiler and a `DockerOptions` object on a Linux system.

Create a standalone application using `magicsquare.m` and save the build results to a `compiler.build.Results` object..

```
buildResults = compiler.build.standaloneApplication(magicsquare.m');
```

Create a `DockerOptions` object to specify additional build options, such as the build folder.

```
opts = compiler.package.DockerOptions(buildResults,...
'DockerContext','DockerImages')
```

```
opts = 

  DockerOptions with properties:

              EntryPoint: 'magicsquare'
               ImageName: 'magicsquare'
            RuntimeImage: ''
  AdditionalInstructions: {}
      AdditionalPackages: {}
      ExecuteDockerBuild: on
           ContainerUser: 'appuser'
           DockerContext: './DockerImages'
          VerbosityLevel: 'verbose'
```

Build the Docker image by passing the generated files to the `compiler.package.docker` function.

```
cd magicsquarestandaloneApplication
```

```
compiler.package.docker('magicsquare','buildresult.json',...
'Options',opts);
```

Input Arguments

`results` — Build results
`compiler.build.Results` object

Build results created by a `compiler.build` function, specified as a `compiler.build.Results` object.

`files` — Files and folders for installation
character vector | string scalar | string array | cell array of strings

Files and folders for installation, specified as a character vector, string scalar, string array, or cell array of strings. These files are typically generated by MATLAB Compiler and can also include any additional files and folders required by the installed application to run. Files generated by MATLAB Compiler in a particular release can be packaged using the `compiler.package.docker` function of the same release.

Example: `'myDockerFiles/'`

Data Types: `char` | `string` | `cell`

`filepath` — Path to `buildresult.json` file
character vector | string scalar

Path to the `buildresult.json` file, specified as a character vector or string scalar. This file is generated by MATLAB Compiler. The path can be relative to the current working directory or absolute. In releases prior to R2025a, use `requiredMCRProducts.txt` instead of `buildresult.json`.

Path to the `buildresult.json` file, specified as a character vector or string scalar. This file is generated by MATLAB Compiler. The path can be relative to the current working directory or absolute.

Example: `'/home/mluser/Documents/MATLAB/magicsquare/buildresult.json'`

Data Types: `char` | `string`

`imageName` — Name of Docker image
character vector | string scalar

Name of the Docker image. It must comply with Docker naming rules.

Example: `'helloworld'`

Data Types: `char` | `string`

`opts` — Docker options
`DockerOptions` object

Docker options, specified as a `DockerOptions` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'ExecuteDockerBuild','on'`

`AdditionalInstructions` — Additional commands to pass to Docker image
`''` (default) | character vector | string scalar | cell array of character vectors

Additional commands to pass to the Docker image, specified as a character vector, a string scalar, or a cell array of character vectors. Commands are added to the `Dockerfile` and execute during image generation.

For information on valid `Dockerfile` commands, see `https://docs.docker.com/engine/reference/builder/`.

Example: `'AdditionalInstructions',{'RUN mkdir /myvol','RUN echo "hello world" > /myvol/greeting','VOLUME /myvol'}`

Data Types: `char` | `string`

`AdditionalPackages` — Additional packages to install into Docker image
`''` (default) | character vector | string scalar | cell array of character vectors

Additional Ubuntu 22.04 packages to install into the Docker image, specified as a character vector, a string scalar, or a cell array of character vectors.

Example: `'AdditionalPackages','syslog-ng'`

Data Types: `char` | `string`

`ContainerUser` — Name of Linux user
`'appuser'` (default) | character vector | string scalar

Name of the Linux user the Docker container will run as, specified as a character vector or a string scalar. The argument must comply with system user naming standards. If the specified user does not exist at the time of creation, a new user will be created with no permissions. If this property is not set, the container will run as the user `appuser` by default, or the user specified in the `FROM` command in the `Dockerfile`.

Example: `'ContainerUser','root'`

Data Types: `char` | `string`

`DockerContext` — Path to build folder
`'`*`ImageName`*`docker'` (default) | character vector | string scalar

Path to the build folder where the Docker image is built, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

If no path is specified, the function creates a build folder named *`ImageName`*`docker` in the current working directory.

Example: `'DockerContext','/home/mluser/Documents/MATLAB/docker/magicsquaredocker'`

Data Types: `char` | `string`

`EntryPoint` — Command executed at image start-up
`''` (default) | character vector | string scalar

The command to be executed at image start-up, specified as a character vector or a string scalar.

Example: `'EntryPoint','exec top -b'`

Data Types: `char` | `string`

`ExecuteDockerBuild` — Flag to build Docker image
`'on'` (default) | on/off logical value

Flag to build the Docker image, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then the function will build the Docker image.
- If you set this property to `'off'`, then the function will populate the `DockerContext` folder without calling 'docker build'.

Example: `'ExecuteDockerBuild','Off'`

Data Types: `logical`

`ImageName` — Name of Docker image
`''` (default) | character vector | string scalar

Name of the Docker image, specified as a character vector or a string scalar. The name must comply with Docker naming rules. Docker repository names must be lowercase. If the main executable or archive file is named using uppercase letters, then the uppercase letters are replaced with lowercase letters in the Docker image name.

Example: `'ImageName','magicsquare'`

Data Types: `char` | `string`

`RuntimeImage` — Name of MATLAB Runtime image
`''` (default) | character vector | string scalar

Name of the MATLAB Runtime image, specified as a character vector or a string scalar. You can use the `compiler.runtime.createDockerImage` function to create a custom MATLAB Runtime image that can run multiple applications. If not specified, MATLAB Compiler generates a selective MATLAB Runtime image that can only run the packaged application.

Example: `'RuntimeImage','mcrimage'`

Data Types: `char` | `string`

`VerbosityLevel` — Output verbosity level
`'verbose'` (default) | `'concise'` | `'none'`

Output verbosity level, specified as one of the following options:

- `'verbose'` (default) — Display all screen output, including Docker output that occurs from the commands '`docker pull`' and '`docker build`'.
- `'concise'` — Display progress information without Docker output
- `'none'` — Do not display output.

Example: `'VerbosityLevel','concise'`

Data Types: `char` | `string`

Version History

Introduced in R2020b

See Also

`compiler.package.DockerOptions` | `compiler.build.standaloneApplication` | `compiler.build.Results` | `compiler.runtime.createDockerImage`

Topics

"Package MATLAB Standalone Applications into Docker Images" on page 14-2
"Create Microservice Docker Image" (MATLAB Compiler SDK)

compiler.package.DockerOptions

Create a Docker options object

Syntax

```
opts = compiler.package.DockerOptions(results)
opts = compiler.package.DockerOptions(results,Name,Value)
opts = compiler.package.DockerOptions('ImageName',imageName)
opts = compiler.package.DockerOptions('ImageName',imageName,Name,Value)
```

Description

Caution This function is only supported on Linux operating systems.

`opts = compiler.package.DockerOptions(results)` creates a `DockerOptions` object `opts` using the `compiler.build.Results` object `results`. The `Results` object is created by a `compiler.build` function. The `DockerOptions` object is passed as an input to the `compiler.package.docker` function to specify build options.

`opts = compiler.package.DockerOptions(results,Name,Value)` creates a `DockerOptions` object `opts` using the `compiler.build.Results` object `results` and additional options specified as one or more pairs of name-value arguments. Options include the build folder, entry point command, and image name.

`opts = compiler.package.DockerOptions('ImageName',imageName)` creates a default `DockerOptions` object with the image name specified by `imageName`.

`opts = compiler.package.DockerOptions('ImageName',imageName,Name,Value)` creates a default `DockerOptions` object with the image name specified by `imageName` and additional options specified as one or more pairs of name-value arguments.

Examples

Create a Docker Options Object Using Build Results

Create a `DockerOptions` object using the build results from a standalone application on a Linux system.

Create a standalone application using `magicsquare.m` and save the build results to a `compiler.build.Results` object.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
buildResults = compiler.build.standaloneApplication(appFile);
```

Create a `DockerOptions` object using the build results from the `compiler.build.standaloneApplication` function.

```
opts = compiler.package.DockerOptions(buildResults);
```

You can modify property values of an existing `DockerOptions` object using dot notation. For example, set the build folder.

```
opts.DockerContext = 'myDockerFiles';
```

The `DockerOptions` and `Results` objects are passed as inputs to the `compiler.package.docker` function to build the Docker image.

```
compiler.package.docker(buildResults,'Options',opts);
```

Customize Docker Options Object Using Build Results

Create a `DockerOptions` object using build results from a standalone application and customize it using name-value arguments.

Create a standalone application using `magicsquare.m` and save the build results to a `compiler.build.Results` object.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
buildResults = compiler.build.standaloneApplication(appFile);
```

Create a `DockerOptions` object using the build results from the `compiler.build.standaloneApplication` function. Use name-value arguments to specify the image name and build folder.

```
opts = compiler.package.DockerOptions(buildResults,...
'DockerContext','Docker/MagicSquare',...
'ImageName','magic-square-')
```

```
opts = 

  DockerOptions with properties:

              EntryPoint: 'magicsquare'
               ImageName: 'magic-square-'
            RuntimeImage: ''
  AdditionalInstructions: {}
      AdditionalPackages: {}
      ExecuteDockerBuild: on
           ContainerUser: 'appuser'
           DockerContext: './Docker/MagicSquare'
          VerbosityLevel: 'verbose'
```

Create Docker Options Object Using Image Name

Create a default `DockerOptions` object to specify the image name.

Create a `DockerOptions` object.

```
opts = compiler.package.DockerOptions('ImageName','helloworld')
```

```
opts = 

  DockerOptions with properties:
```

```
                EntryPoint: ''
                 ImageName: 'helloworld'
              RuntimeImage: ''
    AdditionalInstructions: {}
        AdditionalPackages: {}
        ExecuteDockerBuild: on
             ContainerUser: 'appuser'
             DockerContext: './helloworlddocker'
            VerbosityLevel: 'verbose'
```

You can modify property values of an existing `DockerOptions` object using dot notation. For example, populate the `DockerContext` folder without calling 'docker build'.

```
opts.ExecuteDockerBuild = 'Off'
```

```
opts = 

  DockerOptions with properties:

                EntryPoint: ''
                 ImageName: 'helloworld'
              RuntimeImage: ''
    AdditionalInstructions: {}
        AdditionalPackages: {}
        ExecuteDockerBuild: off
             ContainerUser: 'appuser'
             DockerContext: './helloworlddocker'
            VerbosityLevel: 'verbose'
```

Customize Docker Options Object Using Image Name

Create a `DockerOptions` object using the image name and customize it using name-value arguments.

Create a `DockerOptions` object. Use name-value arguments to specify the build folder and entry point command.

```
opts = compiler.package.DockerOptions('ImageName','myapp',...
'DockerContext','Docker/MyDockerApp',...
'EntryPoint',"exec top -b")
```

```
opts = 

  DockerOptions with properties:

                EntryPoint: 'exec top -b'
                 ImageName: 'myapp'
              RuntimeImage: ''
    AdditionalInstructions: {}
        AdditionalPackages: {}
        ExecuteDockerBuild: on
             ContainerUser: 'appuser'
```

```
      DockerContext: './Docker/MyDockerApp'
     VerbosityLevel: 'verbose'
```

Input Arguments

`results` — Build results
`compiler.build.Results` object

Build results created by a `compiler.build` function, specified as a `compiler.build.Results` object.

`imageName` — Name of Docker image
character vector | string scalar

Name of the Docker image. It must comply with Docker naming rules.

Example: `'hello-world'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'ExecuteDockerBuild','on'`

`AdditionalInstructions` — Additional commands to pass to Docker image
`''` (default) | character vector | string scalar | cell array of character vectors

Additional commands to pass to the Docker image, specified as a character vector, a string scalar, or a cell array of character vectors. Commands are added to the `Dockerfile` and execute during image generation.

For information on valid `Dockerfile` commands, see `https://docs.docker.com/engine/reference/builder/`.

Example: `'AdditionalInstructions',{'RUN mkdir /myvol','RUN echo "hello world" > /myvol/greeting','VOLUME /myvol'}`

Data Types: `char` | `string`

`AdditionalPackages` — Additional packages to install into Docker image
`''` (default) | character vector | string scalar | cell array of character vectors

Additional Ubuntu 22.04 packages to install into the Docker image, specified as a character vector, a string scalar, or a cell array of character vectors.

Example: `'AdditionalPackages','syslog-ng'`

Data Types: `char` | `string`

`ContainerUser` — Name of Linux user
`'appuser'` (default) | character vector | string scalar

Name of the Linux user the Docker container will run as, specified as a character vector or a string scalar. The argument must comply with system user naming standards. If the specified user does not exist at the time of creation, a new user will be created with no permissions. If this property is not set, the container will run as the user `appuser` by default, or the user specified in the `FROM` command in the `Dockerfile`.

Example: `'ContainerUser','root'`

Data Types: `char` | `string`

`DockerContext` — Path to build folder
`'`*`ImageName`*`docker'` (default) | character vector | string scalar

Path to the build folder where the Docker image is built, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

If no path is specified, the function creates a build folder named *`ImageName`*`docker` in the current working directory.

Example: `'DockerContext','/home/mluser/Documents/MATLAB/docker/magicsquaredocker'`

Data Types: `char` | `string`

`EntryPoint` — Command executed at image start-up
`''` (default) | character vector | string scalar

The command to be executed at image start-up, specified as a character vector or a string scalar.

Example: `'EntryPoint','exec top -b'`

Data Types: `char` | `string`

`ExecuteDockerBuild` — Flag to build Docker image
`'on'` (default) | on/off logical value

Flag to build the Docker image, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then the function will build the Docker image.
- If you set this property to `'off'`, then the function will populate the `DockerContext` folder without calling 'docker build'.

Example: `'ExecuteDockerBuild','Off'`

Data Types: `logical`

`ImageName` — Name of Docker image
`''` (default) | character vector | string scalar

Name of the Docker image, specified as a character vector or a string scalar. The name must comply with Docker naming rules. Docker repository names must be lowercase. If the main executable or archive file is named using uppercase letters, then the uppercase letters are replaced with lowercase letters in the Docker image name.

Example: `'ImageName','magicsquare'`

Data Types: `char` | `string`

`RuntimeImage` — Name of MATLAB Runtime image
`''` (default) | character vector | string scalar

Name of the MATLAB Runtime image, specified as a character vector or a string scalar. You can use the `compiler.runtime.createDockerImage` function to create a custom MATLAB Runtime image that can run multiple applications. If not specified, MATLAB Compiler generates a selective MATLAB Runtime image that can only run the packaged application.

Example: `'RuntimeImage','mcrimage'`

Data Types: `char` | `string`

`VerbosityLevel` — Output verbosity level
`'verbose'` (default) | `'concise'` | `'none'`

Output verbosity level, specified as one of the following options:

- `'verbose'` (default) — Display all screen output, including Docker output that occurs from the commands '`docker pull`' and '`docker build`'.
- `'concise'` — Display progress information without Docker output
- `'none'` — Do not display output.

Example: `'VerbosityLevel','concise'`

Data Types: `char` | `string`

Output Arguments

`opts` — Docker options object
`DockerOptions` object

Docker image build options, returned as a `DockerOptions` object.

Limitations

- Only standalone applications can be packaged into Docker images as of R2020b.

Version History

Introduced in R2020b

See Also

`compiler.package.docker` | `compiler.build.standaloneApplication` | `compiler.build.Results`

compiler.package.installer

Create an installer for files generated by MATLAB Compiler

Syntax

```
compiler.package.installer(results)
compiler.package.installer(results,Name,Value)
compiler.package.installer(results,"Options",opts)
compiler.package.installer(files,filePath,"ApplicationName",appName)
compiler.package.installer(files,filePath,"ApplicationName",appName,
Name,Value)
compiler.package.installer(files,filePath,"Options",opts)
```

Description

`compiler.package.installer(results)` creates an installer using the `compiler.build.Results` object `results` generated from a `compiler.build` function.

`compiler.package.installer(results,Name,Value)` creates an installer using the `compiler.build.Results` object `results` with additional options specified using one or more name-value arguments.

`compiler.package.installer(results,"Options",opts)` creates an installer using the `compiler.build.Results` object `results` with installer options specified by an `InstallerOptions` object `opts`. If you use an `InstallerOptions` object, you cannot specify any other options using name-value arguments.

`compiler.package.installer(files,filePath,"ApplicationName",appName)` creates an installer for files generated by the `mcc` command. The installed application name is specified by `appName`. The installer file extension is determined by the operating system in which you run the function.

`compiler.package.installer(files,filePath,"ApplicationName",appName, Name,Value)` creates an installer for files generated by the `mcc` command. The installed application name is specified by `appName`. The installer can be customized using optional name-value arguments.

`compiler.package.installer(files,filePath,"Options",opts)` creates an installer for files generated by the `mcc` command with installer options specified by an `InstallerOptions` object `opts`. If you use an `InstallerOptions` object, you cannot specify any other options using name-value arguments.

Examples

Create Installer Using Results Object

Create an installer for a standalone application using the results from the `compiler.build.standaloneApplication` function.

In MATLAB, locate the MATLAB code that you want to deploy as a standalone application. For this example, compile using the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,"extern","examples","compiler","magicsquare.m");
```

Build a standalone application using the `compiler.build.standaloneApplication` command.

```
results = compiler.build.standaloneApplication(appFile);
```

Create an installer for the standalone application using the `compiler.package.installer` function.

```
compiler.package.installer(results);
```

The function generates an installer named `MyAppInstaller` within a folder named `magicsquareinstaller`.

The default installation folder for the application is the value of the `%ProgramFiles%` environment variable on the target machine.

Create Installer on Offline Machine

Create an installer for a standalone application using the results from the `compiler.build.standaloneApplication` function on an offline machine.

On the offline machine, use the `compiler.runtime.download` function to obtain the URL to the MATLAB Runtime installer at the matching version and update level.

```
compiler.runtime.download
```

```
Downloading MATLAB Runtime installer. It may take several minutes...

Error using compiler.runtime.download
A connection could not be established to download the Runtime Installer.
Download the runtime from:
https://ssd.mathworks.com/supportfiles/downloads/R2025b/Release/0/deployment_files/installer/comp
and update the runtime location in Compiler Settings.
```

Using a machine that is connected to the Internet, download the installer using the URL provided and transfer the installer to the offline machine.

On the offline machine, in MATLAB, open the **Settings** menu and select **MATLAB Compiler**. Specify the location of the MATLAB Runtime installer.

On the offline machine, locate the MATLAB code that you want to deploy as a standalone application. For this example, compile using the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

```
copyfile(fullfile(matlabroot,"extern","examples","compiler","magicsquare.m"));
```

Build a standalone application using the `compiler.build.standaloneApplication` command.

```
results = compiler.build.standaloneApplication("magicsquare.m");
```

Create an installer for the standalone application using the `compiler.package.installer` function.

```
compiler.package.installer(results);
```

The function generates an installer named `MyAppInstaller` within a folder named `magicsquareinstaller`.

The default installation folder for the application is the value of the `%ProgramFiles%` environment variable on the target machine.

Customize Installer Using Results Object

Create an installer for a standalone application using the results from the `compiler.build.standaloneApplication` function and customize it using name-value arguments.

Copy the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

```
copyfile(fullfile(matlabroot,"extern","examples","compiler","magicsquare.m"));
```

Build a standalone application using the `compiler.build.standaloneApplication` command.

```
results = compiler.build.standaloneApplication("magicsquare.m");
```

Create an installer for the standalone application using the `compiler.package.installer` function using the `Results` object. Use name-value arguments to specify the installer name and include MATLAB Runtime within the installer.

```
compiler.package.installer(results,...
    "InstallerName","MyMagicInstaller",...
    "RuntimeDelivery","installer");
```

The function generates an installer named `MyMagicInstaller` within a folder named `magicsquareinstaller`.

Customize Installer Using Results Object and Options Object

Create an installer for a standalone application on a Windows system using the results from the `compiler.build.standaloneApplication` function. Customize the installer using an `InstallerOptions` object.

Copy the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

```
copyfile(fullfile(matlabroot,"extern","examples","compiler","magicsquare.m"));
```

Build a standalone application using the `compiler.build.standaloneApplication` command.

```
results = compiler.build.standaloneApplication("magicsquare.m");
```

Create an `InstallerOptions` object. Use name-value arguments to specify the application name, author company, author name, installer name, and summary.

```
opts = compiler.package.InstallerOptions(results,...
    "ApplicationName","MagicSquare_Generator",...
    "AuthorCompany","Boston Common",...
    "AuthorName","Frog",...
    "InstallerName","MagicSquare_Installer",...
    "Summary","Generates a magic square.")
```

```
opts = 

  InstallerOptions with properties:
```

```
            RuntimeDelivery: 'web'
            InstallerSplash: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\
              InstallerIcon: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\
              InstallerLogo: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\
                PackageType: 'auto'
            AdditionalFiles: {}
      AddRemoveProgramsIcon: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\
                 AuthorName: 'Frog'
                AuthorEmail: ''
              AuthorCompany: 'Boston Common'
                    Summary: 'Generates a magic square.'
                Description: ''
          InstallationNotes: ''
       OptionalDependencies: 'all'
                   Shortcut: 'D:\Work\magicsquarestandaloneApplication\magicsquare.exe'
                    Verbose: 'off'
                    Version: '1.0'
              InstallerName: 'MagicSquare_Installer'
            ApplicationName: 'MagicSquare_Generator'
                  OutputDir: '.\MagicSquare_Generatorinstaller'
    DefaultInstallationDir: '%ProgramFiles%\MagicSquare_Generator'
```

Create an installer for the standalone application using the `Results` and `InstallerOptions` objects as inputs to the `compiler.package.installer` function.

```
compiler.package.installer(results,"Options",opts);
```

The function generates an installer named `MagicSquare_Installer` within a folder named `MagicSquare_Generatorinstaller`.

Create Installer Using Files

Create an installer for a standalone application on a Windows system.

Write a MATLAB function that generates a magic square. Save the function in a file named `mymagic.m`.

```
function out = mymagic(in)
out = magic(in)
```

Build a standalone application using the `mcc` command.

```
mcc -m mymagic.m
```

```
buildresult.json
includedSupportPackages.txt
mccExcludedFiles.log
mymagic.exe
readme.txt
requiredMCRProducts.txt
unresolvedSymbols.txt
```

Create an installer for the standalone application using the `compiler.package.installer` function.

```
compiler.package.installer("mymagic.exe",...
    "buildresult.json",...
    "ApplicationName","MagicSquare_Generator")
```

The function generates an installer named `MyAppInstaller` within a folder named `MagicSquare_Generatorinstaller`.

Customize Installer Using Files

Customize an installer for a standalone application using name-value arguments.

Build a standalone application using the `compiler.build.standaloneApplication` command.

```
copyfile(fullfile(matlabroot,"extern","examples","compiler","magicsquare.m"));
buildResults = compiler.build.standaloneApplication("magicsquare.m");
```

Save the path to the generated `buildresult.json` file.

```
runtimeProducts = fullfile(buildResults.Options.OutputDir,"buildresult.json")
```

Save the list of files from the standalone application build results.

```
fileList = buildResults.Files

fileList = [fileList; {'UsageNotes.txt'}];
```

Create an installer for the standalone application using the `compiler.package.installer` function. Optionally, you can add additional files to the installer using the `AdditionalFiles`. These files are installed in the installation directory along with the application executable.

```
compiler.package.installer(fileList, runtimeProducts,...
    "ApplicationName","CustomMagicSquare",...
    "AdditionalFiles","UsageNotes.txt",...
    "InstallerName","Installer_With_Addl_Files",...
    "Summary","See UsageNotes.txt for info.")
```

Customize Installer Using Files and Installer Options Object

Customize an installer for a standalone application on a Windows system using an `InstallerOptions` object.

Create an `InstallerOptions` object.

```
opts = compiler.package.InstallerOptions("ApplicationName","MagicSquare_Generator",...
    "AuthorCompany","Boston Common",...
    "AuthorName","Frog",...
    "InstallerName","MagicSquare_Installer",...
    "Summary","Generates a magic square.")
```

Pass the `InstallerOptions` object as an input to the function.

```
compiler.package.installer("mymagic.exe","buildresults.json","Options",opts)
```

Input Arguments

`results` — Build results object

`Results` object

Build results, specified as a `compiler.build.Results` object. Create the `Results` object by saving the output from a `compiler.build` function.

`files` — List of files and folders for installation

character vector | string scalar | cell array of character vectors | string array

List of files and folders for installation, specified as a character vector, a string scalar, a cell array of character vectors, or a string array. These files are typically generated by the `mcc` command or a `compiler.build` function and can also include any additional files and folders required by the installed application to run. Additional files are installed in the installation directory along with the application executable.

- Files generated in a particular release can be packaged using the `compiler.package.installer` function of the same release.
- Files of type `.ctf` on one operating system can be packaged using the `compiler.package.installer` function on a different operating system, as long as the build command and the `compiler.package.installer` function are from the same release.

Example: `{'mymagic.exe','UsageNotes.txt'}`

Data Types: `char` | `string`

`filePath` — Path to `buildresult.json` file
character vector | string scalar

Path to the `buildresult.json` file, specified as a character vector or string scalar. This file is generated by MATLAB Compiler. The path can be relative to the current working directory or absolute. In releases prior to R2025a, use `requiredMCRProducts.txt` instead of `buildresult.json`.

Example: `"D:\Documents\MATLAB\work\MagicSquare\buildresult.json"`

Data Types: `char` | `string`

`appName` — Name of the installed application
character vector | string scalar

Name of the installed application, specified as a character vector or a string scalar.

Example: `"MagicSquare_Generator"`

Data Types: `char` | `string`

`opts` — Installer options object
`InstallerOptions` object

Installer options, specified as an `InstallerOptions` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `"Version","9.5"` specifies the version of the installed application.

`AdditionalFiles` — Additional files
character vector | string scalar | cell array of character vectors | string array

Additional files and folders to be installed with the application, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute.

Example: `"AdditionalFiles",["myimage.png","data.mat"]`

Data Types: `char` | `string` | `cell`

`AddRemoveProgramsIcon` — Add or remove programs icon
character vector | string scalar | string array

Add or remove programs icon, specified as a character vector or string scalar. The image is used as the icon in the list of Windows applications within settings. Paths can be relative to the current working directory or absolute.

The default path is:

`"`*`matlabroot`*`\toolbox\compiler\packagingResources\default_icon_48.png"`

Example: `"AddRemoveProgramsIcon","win_icon.png"`

Data Types: `char` | `string`

`ApplicationName` — Application name
`""` (default) | character vector | string scalar

Name of installed application, specified as a character vector or a string scalar.

Example: `"ApplicationName","MagicSquare_Generator"`

Data Types: `char` | `string`

`AuthorCompany` — Company name
`""` (default) | character vector | string scalar

Name of company that created the application, specified as a character vector or a string scalar.

Example: `"AuthorCompany","Boston Common"`

Data Types: `char` | `string`

`AuthorEmail` — Email address
`""` (default) | character vector | string scalar

Email address of the application author, specified as a character vector or a string scalar.

Example: `"AuthorEmail","frog@example.com"`

Data Types: `char` | `string`

`AuthorName` — Name
`""` (default) | character vector | string scalar

Name of application author, specified as a character vector or a string scalar.

Example: `"AuthorName","Frog"`

Data Types: `char` | `string`

`DefaultInstallationDir` — Default installation path
character vector | string scalar

Default directory where you want the installer to install the application, specified as a character vector or a string scalar.

If no path is specified, the default path for each operating system is:

Operating System	Default Installation Directory
Windows	`C:\Program Files\`*`appName`*
Linux	`/usr/`*`appName`*
macOS	`/Applications/`*`appName`*

Example: `"DefaultInstallationDir","C:\Users\MW_Programs\MagicSquare_Generator"`

Data Types: `char` | `string`

`Description` — Detailed application description
`""` (default) | character vector | string scalar

Detailed description of the application, specified as a character vector or a string scalar.

Example: `"Description","The MagicSquare_Generator application generates an n-by-n matrix constructed from the integers 1 through n2 with equal row and column sums."`

Data Types: `char` | `string`

`InstallationNotes` — Notes
`""` (default) | character vector | string scalar

Notes about additional requirements for using application, specified as a character vector or a string scalar.

Example: `"InstallationNotes","This is a Linux installer."`

Data Types: `char` | `string`

`InstallerIcon` — Path to icon image
character vector | string scalar

Path to an image file used as the icon for the installed application, specified as a character vector or a string scalar.

The default path is:

```
"matlabroot\toolbox\compiler\packagingResources\default_icon_48.png"
```

Example: `"InstallerIcon","D:\Documents\MATLAB\work\images\myIcon.png"`

`InstallerLogo` — Path to installer image
character vector | string scalar

Path to an image file used as the installer logo, specified as a character vector or a string scalar. The logo will be resized to 112 pixels by 290 pixels.

The default path is:

```
"matlabroot\toolbox\compiler\packagingResources\default_logo.png"
```

Example: `"InstallerLogo","D:\Documents\MATLAB\work\images\myLogo.png"`

`InstallerName` — Name of installer file
`MyAppInstaller` (default) | character vector | string scalar

Name of the installer file, specified as a character vector or a string scalar. The extension is determined by the operating system in which the function is executed.

Example: `"InstallerName","MagicSquare_Installer"`

`InstallerSplash` — Path to splash screen image
character vector | string scalar

Path to an image file used as the installer splash screen, specified as a character vector or a string scalar. The splash screen icon will be resized to 400 pixels by 400 pixels.

The default path is:

"*matlabroot*\toolbox\compiler\packagingResources\default_splash.png"

Example: `"InstallerSplash","D:\Documents\MATLAB\work\images\mySplash.png"`

`OptionalDependencies` — Optional dependencies to include
`"all"` (default) | `"none"`

Optional MATLAB Runtime dependencies to include in the installer, specified as one of the following:

- `"all"` — Option to include all of the optional dependencies in the installer, including graphical support. This option is the default behavior.
- `"none"` — Option to not include any of the optional dependencies in the installer. Use this option to minimize the size of the generated installer.

Example: `"OptionalDependencies","none"`

Data Types: `char` | `string`

`OutputDir` — Path to folder where the installer will be saved
character vector | string scalar

Path to folder where the installer is saved, specified as a character vector or a string scalar.

If no path is specified, the default path for each operating system is:

Operating System	Default Installation Directory
Windows	.\appNameinstaller
Linux	./*appName*installer
macOS	./*appName*installer

The `.` in the directories listed above represents the present working directory.

Example: `"OutputDir","D:\Documents\MATLAB\work\MagicSquare"`

PackageType — Installer file type
`"auto"` (default) | `"zip"`

Choice on the file type of the generated installer.

- `"auto"` — Option for installer to automatically select the suitable package type when creating the installer. If the installer size is 2GB or greater, the installer is packaged as a ZIP file. This is the default option.
- `"zip"` — Option to generate a ZIP file as an output when creating a new installer. This option is only supported on Windows.

Example: `"PackageType","zip"`

Data Types: `char` | `string`

`RuntimeDelivery` — MATLAB Runtime delivery option
`"web"` (default) | `"installer"` | `"none"`

MATLAB Runtime delivery option, specified as one of the following:

- `"web"` — Option for the installer to download MATLAB Runtime from the MathWorks website during application installation. This option is the default behavior.
- `"installer"` — Option to include MATLAB Runtime within the installer so that it can be installed during application installation without connecting to the MathWorks website. Use this option if you think your end user might not have access to the internet.
- `"none"` — Option to not install MATLAB Runtime during application installation. Use this option if you think your end user will install MATLAB Runtime using another method.

Note If you use either the `"web"` or `"installer"` option, the installer reduces disk space usage by installing only the MATLAB Runtime components needed to run the specified application. To create a minimal MATLAB Runtime installer that can run one or more specific MATLAB Compiler applications, see `compiler.runtime.customInstaller`.

Example: `"RuntimeDelivery","installer"`

Data Types: `char` | `string`

`Shortcut` — Path to shortcut
`""` (default) | character vector | string scalar

Path to a file or folder that the installer will create a shortcut to at install time, specified as a character vector or a string scalar.

Example: `"Shortcut",".\mymagic.exe"`

Data Types: `char` | `string`

`Summary` — Summary description of application
`""` (default) | character vector | string scalar

Summary description of the application, specified as a character vector or a string scalar.

Example: `"Summary","Generates a magic square."`

Data Types: `char` | `string`

`Version` — Version of installed application
`"1.0"` (default) | character vector | string scalar

Version number of the installed application, specified as a character vector or a string scalar.

Example: "Version","2.0"

Data Types: char | string

Verbose — Flag to control output verbosity
"off" (default) | on/off logical value

Flag to control output verbosity, specified as "on" or "off", or as numeric or logical 1 (true) or 0 (false). A value of "on" is equivalent to true, and "off" is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type matlab.lang.OnOffSwitchState.

- If you set this property to "on", then the MATLAB command window displays progress information indicating compiler output during the packaging process.
- If you set this property to "off", then the command window does not display progress information. This is the default behavior.

Example: "Verbose","on"

Data Types: logical

Tips

- The installer created by this function installs your MATLAB Compiler generated artifacts and optionally the full MATLAB Runtime. To create a minimal MATLAB Runtime installer that can run one or more specific MATLAB Compiler applications, see compiler.runtime.customInstaller.
- Installers that contain MATLAB Runtime require access to the full MATLAB Runtime installer at the same version *and* update level as the version of MATLAB used to create them. Use the compiler.runtime.download function to obtain the MATLAB Runtime installer at the same release and update level.

Version History

Introduced in R2020a

R2021b: Specify support packages

You can use the SupportPackages option to specify support packages to include in the deployable code archive.

R2023a: Specify additional files

You can use the AdditionalFiles option to specify additional files to include in the installer.

R2023a: Specify icon for Windows application list

You can use the AddRemoveProgramsIcon option to specify the image used as the icon in the list of Windows applications within settings.

R2023b: Control verbosity

You can use the `Verbose` option to control output verbosity.

R2024a: Package installer as ZIP file

You can use the `PackageType` option to choose whether the installer is packaged as a ZIP file if the installer size is less than 2GB on Windows. By default, a ZIP installer is created only when the installer size exceeds 2 GB.

R2024a: Exclude MATLAB Runtime from installer

You can use the `RuntimeDelivery` option with the value set to `"none"` to not install MATLAB Runtime during application installation. Use this option if you think your end user will install MATLAB Runtime using another method.

R2025a: Optional dependencies

You can use the `OptionalDependencies` option to control whether to include all of the optional MATLAB Runtime dependencies in the installer.

R2025a: Build using `buildresult.json`

You can use the JSON file named `buildresult.json` in place of `requiredMCRProducts.txt` in deployment workflows. The `requiredMCRProducts.txt` file will be removed in a future release.

See Also
`compiler.package.InstallerOptions` | `compiler.runtime.customInstaller` | `compiler.build.Results` | `mcc`

compiler.package.InstallerOptions

Options for creating MATLAB Compiler package installers

Syntax

```
opts = compiler.package.InstallerOptions(results)
opts = compiler.package.InstallerOptions(results,Name,Value)
opts = compiler.package.InstallerOptions('ApplicationName',appName)
opts = compiler.package.InstallerOptions('ApplicationName',appName,
Name,Value)
```

Description

`opts = compiler.package.InstallerOptions(results)` creates a default `InstallerOptions` object `opts` using the `compiler.build.Results` object `results` generated from a `compiler.build` function. The `InstallerOptions` object is passed as an input to the `compiler.package.installer` function.

`opts = compiler.package.InstallerOptions(results,Name,Value)` creates an `InstallerOptions` object `opts` using the `compiler.build.Results` object `results` with additional options specified using one or more name-value arguments. The `InstallerOptions` object is passed as an input to the `compiler.package.installer` function.

`opts = compiler.package.InstallerOptions('ApplicationName',appName)` creates a default `InstallerOptions` object `opts` with application name specified by `appName`. The `InstallerOptions` object is passed as an input to the `compiler.package.installer` function.

`opts = compiler.package.InstallerOptions('ApplicationName',appName, Name,Value)` creates an `InstallerOptions` object `opts` with application name specified by `appName` and additional customizations specified by name-value arguments. The `InstallerOptions` object is passed as an input to the `compiler.package.installer` function.

Examples

Create an Installer Options Object Using Results

Create an `InstallerOptions` object using the results from the `compiler.build.standaloneApplication` function and additional options specified as name-value arguments.

For this example, build a standalone application using the file `magicsquare.m` located in `*matlabroot*\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
results = compiler.build.standaloneApplication(appFile);
```

Create an `InstallerOptions` object. Use name-value arguments to specify the application name, author company, author name, installer name, and summary.

```
opts = compiler.package.InstallerOptions(results,...
    'ApplicationName','MagicSquare_Generator',...
    'AuthorCompany','Boston Common',...
    'AuthorName','Frog',...
    'InstallerName','MagicSquare_Installer',...
    'Summary','Generates a magic square.')
```

```
opts = 

  InstallerOptions with properties:

           RuntimeDelivery: 'web'
           InstallerSplash: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\default_splash.png'
             InstallerIcon: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\default_icon_48.png'
             InstallerLogo: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\default_logo.png'
           AdditionalFiles: {}
     AddRemoveProgramsIcon: 'C:\Program Files\MATLAB\R2025b\toolbox\compiler\packagingResources\default_icon_48.png'
                AuthorName: 'Frog'
               AuthorEmail: ''
             AuthorCompany: 'Boston Common'
                   Summary: 'Generates a magic square.'
               Description: ''
         InstallationNotes: ''
      OptionalDependencies: 'all'
                  Shortcut: 'D:\Work\magicsquarestandaloneApplication\magicsquare.exe'
                   Version: '1.0'
             InstallerName: 'MagicSquare_Installer'
           ApplicationName: 'MagicSquare_Generator'
                 OutputDir: '.\MagicSquare_Generatorinstaller'
       DefaultInstallationDir: 'C:\Program Files\MagicSquare_Generator'
                   Verbose: 'off'
```

You can modify the property values of an existing `InstallerOptions` object using dot notation. For example, set the installer name to `MyMagicInstaller`.

```
opts.InstallerName = 'MyMagicInstaller'
```

To create an installer for the standalone application, use the `Results` and `InstallerOptions` objects as inputs to the `compiler.package.installer` function.

```
compiler.package.installer(results,'Options',opts);
```

The function generates an installer named `MagicSquare_Installer` within a folder named `MagicSquare_Generatorinstaller`.

You can modify the property values of an existing `InstallerOptions` object using dot notation. For example, specify installation notes.

```
opts.InstallationNotes = 'Windows installer for MagicSquare'
```

Create an Installer Options Object Using Application Name

Create an `InstallerOptions` object with an application name and additional options specified as name-value arguments.

```
opts = compiler.package.InstallerOptions('ApplicationName','MagicSquare_Generator',...
    'AuthorCompany','Boston Common',...
    'AuthorName','Frog',...
    'InstallerName','MagicSquare_Installer',...
    'Summary','Generates a magic square.');
```

Input Arguments

`results` — Build results object

Results object

Build results, specified as a `compiler.build.Results` object. Create the `Results` object by saving the output from a `compiler.build` function.

`appName` — Name of the installed application
character vector | string scalar

Name of the installed application, specified as a character vector or a string scalar.

Example: `'MagicSquare_Generator'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `"Version","9.5"` specifies the version of the installed application.

`AdditionalFiles` — Additional files
character vector | string scalar | cell array of character vectors | string array

Additional files and folders to be installed with the application, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute.

Example: `"AdditionalFiles",["myimage.png","data.mat"]`

Data Types: `char` | `string` | `cell`

`AddRemoveProgramsIcon` — Add or remove programs icon
character vector | string scalar | string array

Add or remove programs icon, specified as a character vector or string scalar. The image is used as the icon in the list of Windows applications within settings. Paths can be relative to the current working directory or absolute.

The default path is:

`"`*`matlabroot`*`\toolbox\compiler\packagingResources\default_icon_48.png"`

Example: `"AddRemoveProgramsIcon","win_icon.png"`

Data Types: `char` | `string`

`ApplicationName` — Application name
`""` (default) | character vector | string scalar

Name of installed application, specified as a character vector or a string scalar.

Example: `"ApplicationName","MagicSquare_Generator"`

Data Types: `char` | `string`

`AuthorCompany` — Company name
`""` (default) | character vector | string scalar

Name of company that created the application, specified as a character vector or a string scalar.

Example: `"AuthorCompany","Boston Common"`

Data Types: `char` | `string`

`AuthorEmail` — Email address
`""` (default) | character vector | string scalar

Email address of the application author, specified as a character vector or a string scalar.

Example: `"AuthorEmail","frog@example.com"`

Data Types: `char` | `string`

`AuthorName` — Name
`""` (default) | character vector | string scalar

Name of application author, specified as a character vector or a string scalar.

Example: `"AuthorName","Frog"`

Data Types: `char` | `string`

`DefaultInstallationDir` — Default installation path
character vector | string scalar

Default directory where you want the installer to install the application, specified as a character vector or a string scalar.

If no path is specified, the default path for each operating system is:

Operating System	Default Installation Directory
Windows	`C:\Program Files\`*`appName`*
Linux	`/usr/`*`appName`*
macOS	`/Applications/`*`appName`*

Example: `"DefaultInstallationDir","C:\Users\MW_Programs\MagicSquare_Generator"`

Data Types: `char` | `string`

`Description` — Detailed application description
`""` (default) | character vector | string scalar

Detailed description of the application, specified as a character vector or a string scalar.

Example: `"Description","The MagicSquare_Generator application generates an n-by-n matrix constructed from the integers 1 through n2 with equal row and column sums."`

Data Types: `char` | `string`

`InstallationNotes` — Notes
`""` (default) | character vector | string scalar

Notes about additional requirements for using application, specified as a character vector or a string scalar.

Example: `"InstallationNotes","This is a Linux installer."`

Data Types: `char` | `string`

`InstallerIcon` — Path to icon image
character vector | string scalar

Path to an image file used as the icon for the installed application, specified as a character vector or a string scalar.

The default path is:

"*matlabroot*\toolbox\compiler\packagingResources\default_icon_48.png"

Example: `"InstallerIcon","D:\Documents\MATLAB\work\images\myIcon.png"`

`InstallerLogo` — Path to installer image
character vector | string scalar

Path to an image file used as the installer logo, specified as a character vector or a string scalar. The logo will be resized to 112 pixels by 290 pixels.

The default path is:

"*matlabroot*\toolbox\compiler\packagingResources\default_logo.png"

Example: `"InstallerLogo","D:\Documents\MATLAB\work\images\myLogo.png"`

`InstallerName` — Name of installer file
`MyAppInstaller` (default) | character vector | string scalar

Name of the installer file, specified as a character vector or a string scalar. The extension is determined by the operating system in which the function is executed.

Example: `"InstallerName","MagicSquare_Installer"`

`InstallerSplash` — Path to splash screen image
character vector | string scalar

Path to an image file used as the installer splash screen, specified as a character vector or a string scalar. The splash screen icon will be resized to 400 pixels by 400 pixels.

The default path is:

"*matlabroot*\toolbox\compiler\packagingResources\default_splash.png"

Example: `"InstallerSplash","D:\Documents\MATLAB\work\images\mySplash.png"`

`OptionalDependencies` — Optional dependencies to include
`"all"` (default) | `"none"`

Optional MATLAB Runtime dependencies to include in the installer, specified as one of the following:

- `"all"` — Option to include all of the optional dependencies in the installer, including graphical support. This option is the default behavior.
- `"none"` — Option to not include any of the optional dependencies in the installer. Use this option to minimize the size of the generated installer.

Example: `"OptionalDependencies","none"`

Data Types: `char` | `string`

`OutputDir` — Path to folder where the installer will be saved
character vector | string scalar

Path to folder where the installer is saved, specified as a character vector or a string scalar.

If no path is specified, the default path for each operating system is:

Operating System	Default Installation Directory
Windows	`.\`*`appName`*`installer`
Linux	`./`*`appName`*`installer`
macOS	`./`*`appName`*`installer`

The `.` in the directories listed above represents the present working directory.

Example: `"OutputDir","D:\Documents\MATLAB\work\MagicSquare"`

`PackageType` — Installer file type
`"auto"` (default) | `"zip"`

Choice on the file type of the generated installer.

- `"auto"` — Option for installer to automatically select the suitable package type when creating the installer. If the installer size is 2GB or greater, the installer is packaged as a ZIP file. This is the default option.
- `"zip"` — Option to generate a ZIP file as an output when creating a new installer. This option is only supported on Windows.

Example: `"PackageType","zip"`

Data Types: `char` | `string`

`RuntimeDelivery` — MATLAB Runtime delivery option
`"web"` (default) | `"installer"` | `"none"`

MATLAB Runtime delivery option, specified as one of the following:

- `"web"` — Option for the installer to download MATLAB Runtime from the MathWorks website during application installation. This option is the default behavior.
- `"installer"` — Option to include MATLAB Runtime within the installer so that it can be installed during application installation without connecting to the MathWorks website. Use this option if you think your end user might not have access to the internet.
- `"none"` — Option to not install MATLAB Runtime during application installation. Use this option if you think your end user will install MATLAB Runtime using another method.

Note If you use either the `"web"` or `"installer"` option, the installer reduces disk space usage by installing only the MATLAB Runtime components needed to run the specified application. To create a minimal MATLAB Runtime installer that can run one or more specific MATLAB Compiler applications, see `compiler.runtime.customInstaller`.

Example: `"RuntimeDelivery","installer"`

Data Types: `char` | `string`

`Shortcut` — Path to shortcut
`""` (default) | character vector | string scalar

Path to a file or folder that the installer will create a shortcut to at install time, specified as a character vector or a string scalar.

Example: `"Shortcut",".\mymagic.exe"`

Data Types: `char` | `string`

`Summary` — Summary description of application
`""` (default) | character vector | string scalar

Summary description of the application, specified as a character vector or a string scalar.

Example: `"Summary","Generates a magic square."`

Data Types: `char` | `string`

`Version` — Version of installed application
`"1.0"` (default) | character vector | string scalar

Version number of the installed application, specified as a character vector or a string scalar.

Example: `"Version","2.0"`

Data Types: `char` | `string`

`Verbose` — Flag to control output verbosity
`"off"` (default) | on/off logical value

Flag to control output verbosity, specified as `"on"` or `"off"`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `"on"` is equivalent to `true`, and `"off"` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `"on"`, then the MATLAB command window displays progress information indicating compiler output during the packaging process.
- If you set this property to `"off"`, then the command window does not display progress information. This is the default behavior.

Example: `"Verbose","on"`

Data Types: `logical`

Output Arguments

`opts` — Installer options object
`InstallerOptions` object

Installer options, returned as an `InstallerOptions` object.

Version History

Introduced in R2020a

See Also

`compiler.package.installer` | `mcc`

compiler.runtime.createDockerImage

Create a Docker image that contains MATLAB Runtime

Syntax

```
compiler.runtime.createDockerImage(results)
compiler.runtime.createDockerImage(filepath)
compiler.runtime.createDockerImage( ___ ,Name,Value)
name = compiler.runtime.createDockerImage( ___ )
```

Description

`compiler.runtime.createDockerImage(results)` creates a MATLAB Runtime Docker image using a vector of `compiler.build.Results` objects `results`. The `results` objects can be created by any `compiler.build` function.

`compiler.runtime.createDockerImage(filepath)` creates a MATLAB Runtime Docker image using a list of `buildresult.json` files generated by MATLAB Compiler.

`compiler.runtime.createDockerImage( ___ ,Name,Value)` creates a MATLAB Runtime Docker image with additional options specified as one or more name-value arguments. Options include the build folder, verbosity level, and image name.

`name = compiler.runtime.createDockerImage( ___ )` creates a MATLAB Runtime Docker image and returns the name of the image as a character vector.

Examples

Create MATLAB Runtime Docker Image for Standalone Applications

Create a MATLAB Runtime Docker image that can run two standalone application Docker containers.

Install and configure Docker on your system.

Create two standalone applications using `compiler.build.standaloneApplication`. Save the output from each function as a `compiler.build.Results` object.

```
results1 = compiler.build.standaloneApplication('magicsquare.m');
```

```
results2 = compiler.build.standaloneApplication('houdini.mlapp');
```

Create a MATLAB Runtime Docker image that can be used for both deployed standalone applications.

```
name = compiler.runtime.createDockerImage([results1,results2]);
```

Create a Docker image for each standalone application. Specify the name of the MATLAB Runtime image you created as an input to the `compiler.package.docker` function.

```
compiler.package.docker(results1,'RuntimeImage',name)
```

```
compiler.package.docker(results2,'RuntimeImage',name)
```

Deploy the Docker images. For details, see “Package MATLAB Standalone Applications into Docker Images” on page 14-2.

Create MATLAB Runtime Docker Image for Microservices

Requires MATLAB Compiler SDK

Create a MATLAB Runtime Docker image that can run two microservice Docker containers.

Create two production server archives using `compiler.build.productionServerArchive`.

```
results1 = compiler.build.productionServerArchive('addmatrix.m',...
'OutputDir','add');
```

```
results2 = compiler.build.productionServerArchive('eigmatrix.m',...
'OutputDir','eig');
```

Create a MATLAB Runtime Docker image that can be used for both microservices. Use name-value arguments to specify the image name and build folder.

```
compiler.runtime.createDockerImage(["add/requiredMCRProducts.txt",...
"eig/requiredMCRProducts.txt"],'ImageName','micro-combo-image',...
'DockerContext','runtime')
```

Create a microservice Docker image for each production server archive. Specify the name of the MATLAB Runtime image you created as an input to the `compiler.package.microserviceDockerImage` function.

```
compiler.package.microserviceDockerImage(results1,...
'ImageName','add-micro','RuntimeImage','micro-combo-image');
```

```
compiler.package.microserviceDockerImage(results2,...
'ImageName','eig-micro','RuntimeImage','micro-combo-image');
```

Deploy the microservices. For details, see “Create Microservice Docker Image” (MATLAB Compiler SDK).

Create MATLAB Runtime Docker Image with Custom Base Layer

Create a MATLAB Runtime Docker image with a custom base layer image. For more details on this example, see .

Install and configure Docker on your system.

Create a standalone application using `compiler.build.standaloneApplication`. Save the output as a `compiler.build.Results` object.

```
results1 = compiler.build.standaloneApplication('magicsquare.m');
```

Create a custom base layer using a Dockerfile. For this example, create a new file named `Dockerfile.customdeps` that specifies the Linux distribution and installs all of the dependencies required for MATLAB Runtime. You can add custom commands or dependencies to fulfill your deployment requirements.

At the MATLAB command window, build the custom base layer image using Docker and the Dockerfile you created.

```
depsImageName = "mycompanybase:r2024b";
system("docker build -f Dockerfile.customdeps -t " + depsImageName + " .");
```

Create a MATLAB Runtime Docker image layer that uses the custom base layer you created.

```
runtimeImageName = "custom-matlabruntime:r2024b";
compiler.runtime.createDockerImage(buildResults, ...
    "BaseImage",depsImageName, ...
    "ImageName",runtimeImageName)
```

Create a Docker image for the standalone application layer. Specify the name of the MATLAB Runtime image you created as an input to the `compiler.package.docker` function.

```
compiler.package.docker(buildResults, ...
    "ImageName","mymagicimg","RuntimeImage",runtimeImageName);
```

Deploy the Docker image. For details, see “Package MATLAB Standalone Applications into Docker Images” on page 14-2.

Input Arguments

`results` — Build results
Vector of `compiler.build.Results` objects

One or more build results created by a `compiler.build` function, such as `compiler.build.standaloneApplication` or `compiler.build.productionServerArchive`, specified as a vector of `compiler.build.Results` objects.

Example: `[results1,results2]`

`filepath` — Paths to `requiredMCRProducts.txt`
character vector | string scalar | string array | cell array of strings

One or more paths to `requiredMCRProducts.txt`, which is generated by MATLAB Compiler.

Example: `["fun1/requiredMCRProducts.txt","fun2/requiredMCRProducts.txt"]`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:

BaseImage — Name of base Docker image
character vector | string scalar

Name of the base Docker image that is used in the `FROM` line to create the MATLAB Runtime image, specified as a character vector or a string scalar. If this property is not specified, a base image with all required dependencies is created.

Example: `'BaseImage','matlab-deps'`

Data Types: `char` | `string`

`DockerContext` — Path to build folder
`'ImageNamedocker'` (default) | character vector | string scalar

Path to the build folder where the Docker image is built, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

If no path is specified, the function creates a build folder named *`ImageName`*`runtime` in the current working directory.

Example: `'DockerContext','./docker/magicruntime'`

Data Types: `char` | `string`

`ExecuteDockerBuild` — Flag to build Docker image
`'on'` (default) | on/off logical value

Flag to build the Docker image, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then the function will build the Docker image.
- If you set this property to `'off'`, then the function will populate the `DockerContext` folder without calling '`docker build`'.

Example: `'ExecuteDockerBuild','Off'`

Data Types: `logical`

`ImageName` — Name of Docker image
character vector | string scalar

Name of the MATLAB Runtime Docker image, specified as a character vector or a string scalar. The name must comply with Docker naming rules. Docker repository names must be lowercase. If specified, this property is used as the `name` output argument.

If not specified, the default image name resembles the following:

```
matlabruntime/r2025b/update0/1000000000000000
```

Example: `'ImageName','magicruntime'`

Data Types: `char` | `string`

`OptionalDependencies` — Optional dependencies to include
`'none'` (default) | `'all'`

Optional MATLAB Runtime dependencies to include in the installer, specified as one of the following:

- `'none'` — Option to not include any of the optional dependencies in the installer. Use this option to minimize the size of the generated installer. This option is the default behavior.
- `'all'` — Option to include all of the optional dependencies in the installer, including graphical support.

Example: `'OptionalDependencies','all'`

Data Types: `char` | `string`

`VerbosityLevel` — Output verbosity level
`'verbose'` (default) | `'concise'` | `'none'`

Output verbosity level, specified as one of the following options:

- `'verbose'` (default) — Display all screen output, including Docker output that occurs from the commands '`docker pull`' and '`docker build`'.
- `'concise'` — Display progress information without Docker output
- `'none'` — Do not display output.

Example: `'VerbosityLevel','concise'`

Data Types: `char` | `string`

Output Arguments

`name` — Name of image

Name of the generated MATLAB Runtime Docker image, specified as a character vector. You can set the name using the `ImageName` property.

Data Types: `char`

Limitations

- This function requires an installation of Docker. On macOS, Docker Desktop is supported. On Windows, Docker Desktop and Docker in WSL2 are supported.

Version History

Introduced in R2023b

See Also

`compiler.package.docker` | `compiler.package.microserviceDockerImage`

Topics

"Package MATLAB Standalone Applications into Docker Images" on page 14-2
"Create Microservice Docker Image" (MATLAB Compiler SDK)

External Websites
`https://www.docker.com`

createDeploymentScript

Create a deployment script from a MATLAB Compiler PRJ file

Syntax

```
createDeploymentScript(prjfile,outputfile)
```

Description

`createDeploymentScript(prjfile,outputfile)` creates a deployment script file by obtaining the information from a MATLAB Compiler project file `prjfile` and generating a MATLAB script file at the location defined by `outputfile`.

Examples

Create Deployment Script

Create a deployment script named `deployMyMagic` using the project file `mymagic.prj`.

Create a MATLAB Compiler project using a compiler app, such as the Standalone Application Compiler. Add a main file and save the project as `mymagic.prj`.

Save the compiler settings from `mymagic.prj` in a deployment script named `deployMyMagic` using `createDeploymentScript`.

```
createDeploymentScript("mymagic.prj", "deployMyMagic.m");
```

Input Arguments

`prjfile` — Path to MATLAB Compiler or MATLAB Compiler SDK project file

Path to the MATLAB Compiler or MATLAB Compiler SDK project file, specified as a character vector or string scalar.

Example: `"mymagic.prj"`

Data Types: `char` | `string`

`outputfile` — Path to file to write as deployment script

Path to the file to write as the deployment script, specified as a character vector or string scalar. The file must not exist and must include either the extension `.m` or no extension. If you do not specify an extension, `.m` is appended. If you do not specify a full path, `outputfile` is saved in the current folder.

Example: `"deployMyMagic.m"`

Data Types: `char` | `string`

Version History

Introduced in R2022b

See Also

`mcc` | `compiler.build.Results`

ctfroot

Location of files related to deployed application

Syntax

`root = ctfroot`

Description

`root = ctfroot` returns the name of the folder where the deployable archive for the application is expanded.

Use this function to access any file that the user would have included in their project (excluding the ones in the packaging folder).

Examples

Determine location of deployable archive

```
appRoot = ctfroot;
```

Output Arguments

`root` — Path to expanded deployable archive
character vector

Path to expanded deployable archive returned as a character vector in the form: *`application_name`*`_mcr`. .

Version History

Introduced in R2006a

deploytool

(Not recommended) Open the Standalone Application Compiler app

Note deploytool is not recommended. Use `standaloneApplicationCompiler` instead. For more information, see Version History.

Syntax

`deploytool`

Description

`deploytool` opens the Standalone Application Compiler app. The behavior is equivalent to the `standaloneApplicationCompiler` function.

Version History

Introduced in R2006b

R2025a: deploytool behavior change

Not recommended starting in R2025a

In R2024b and earlier releases, deploytool opens a list of application deployment apps. Starting in R2025a, deploytool opens the Standalone Application Compiler app.

R2025a: `-build` and `-package` options have been removed

The `-build` and `-package` options have been removed. To build applications, use one of the `compiler.build` family of functions or the `mcc` command; and to package and create an installer, use the `compiler.package.installer` function.

See Also

Standalone Application Compiler

Topics

"Create Standalone Application Using Standalone Application Compiler App" on page 18-5

getmcruserdata

Retrieve MATLAB array value associated with a given key

Syntax

```
value = getmcruserdata(key)
```

Description

`value = getmcruserdata(key)` returns MATLAB data associated with the string `key` in the current MATLAB Runtime instance. If there is no data associated with the key, it returns an empty matrix.

This function is part of the MATLAB Runtime User Data interface API. It is available both in MATLAB and in deployed applications created with MATLAB Compiler and MATLAB Compiler SDK.

Examples

Get the magic square data associated with the string `'magic'` in the current instance of the MATLAB Runtime.

```
value = magic(3);
setmcruserdata('magic', value);
getmcruserdata('magic')
```

```
ans =
     8     1     6
     3     5     7
     4     9     2
```

Input Arguments

key — Key associated with MATLAB data
string

`key` is the MATLAB string with which MATLAB data `value` is associated within the current instance of the MATLAB Runtime.

Output Arguments

value — Value of MATLAB data
any MATLAB data type including matrices, cell arrays, and Java objects

`value` is the MATLAB data associated with input string `key` for the current instance of the MATLAB Runtime.

Version History

Introduced in R2008b

See Also

`setmcruserdata`

isdeployed

Determine whether code is running in deployed or MATLAB mode

Syntax

```
x = isdeployed
```

Description

`x = isdeployed` returns logical 1 (`true`) when the function is running in deployed mode using MATLAB Runtime and 0 (`false`) if it is running in a MATLAB session.

An application running in deployed mode consists of a collection of MATLAB functions and data packaged using MATLAB Compiler into software components that run outside a MATLAB session using MATLAB Runtime libraries.

Examples

Access Files from Deployed Functions

You can access files included in a packaged application by using the `which` function, or by specifying the file location relative to `ctfroot`.

Add a file such as `extern_app.exe` to your MATLAB Compiler project.

Check if the code is running in deployed mode by using `isdeployed`. Then, obtain the path to the file by using the `which` function.

```
if isdeployed
    locate_externapp = which(fullfile('extern_app.exe'));
end
```

The `which` function returns the path to the file `extern_app.exe`, as long as it is located within the deployable archive.

For more information, see “Include and Access Files in Packaged Applications” on page 5-15.

Protect Use of `ADDPATH`

The path of a deployed application is fixed at compile time and cannot change. Use `isdeployed` to ensure that the application does not attempt to use path modifying functions, such as `addpath`, after deployment.

```
if ~(ismcc || isdeployed)
    addpath(mypath);
end
```

Display Documentation

You cannot use the `doc` function to open the Help browser from a deployed application. Instead, redirect a help query to the MathWorks website.

```
if ~isdeployed
    doc(mfile);
else
   web('https://www.mathworks.com/support.html');
end
```

Suppress Warnings for Non-Deployable Function

Use `isdeployed` with the `%#exclude` pragma to suppress compile time warnings for the non-deployable function `edit`.

```
if ~isdeployed
    %#exclude edit
    edit('readme.txt');
end
```

The pragma excludes the function from dependency analysis during compilation.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Returns true and false as appropriate for MEX targets.
- Returns false for SIM targets, which you should query using `coder.target`.
- Returns false for other targets.

See Also

`ismcc` | `mcc` | `%#exclude`

Topics

"Write Deployable MATLAB Code" on page 5-10
"Include and Access Files in Packaged Applications" on page 5-15

ismcc

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` that runs outside of MATLAB in a system command prompt, 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 must be used in `matlabrc` or `hgrc` (or any function called within them, for example `startup.m`) to guard code from being executed by MATLAB Compiler (`mcc`) or MATLAB Compiler SDK.

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 in `startup.m`, as shown in the example on this page.

Examples

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

Version History

Introduced in R2008b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Returns true and false as appropriate for MEX and SIM targets.
- Returns false for other targets.

See Also

`isdeployed` | `mcc`

mcc

Compile MATLAB functions for deployment

Syntax

```
mcc [options] mfilename1 mfilename2 ... mfilenameN
mcc(options,mfilename)

mcc -m [options] mfilename
mcc -e [options] mfilename

mcc -W 'excel:addin_name,class_name,version=version_number' [options]
mfilename1 mfilename2 ... mfilenameN

mcc -W 'hadoop:archive_name,CONFIG:config_file' mfilename

mcc -m [options] mfilename

mcc -W python:package_name [options] mfilename1 mfilename2 ... mfilenameN

mcc -W
'dotnet:assembly_name,api=api_type,class_name,framework_version,security,remo
te_type' [options] mfilename1 mfilename2 ... mfilenameN
mcc -W
'dotnet:assembly_name,api=api_type,class_name,framework_version,security,remo
te_type' [options] 'class{class_name:mfilename1,mfilename2,...,mfilenameN}'

mcc -W 'java:package_name,class_name' [options] mfilename1 mfilename2 ...
mfilenameN
mcc -W 'java:package_name,class_name' [options] '
class{class_name:mfilename1,mfilename2,...,mfilenameN}'

mcc -l [options] mfilename1 mfilename2 ... mfilenameN

mcc -W 'cpplib:library_name[,{all|legacy|generic}]' [options] mfilename1
mfilename2 ... mfilenameN

mcc -W 'com:component_name,class_name' [options] mfilename1 mfilename2 ...
mfilenameN
mcc -W 'com:component_name,class_name' [options] '
class{class_name:mfilename1,mfilename2,...,mfilenameN}'

mcc -U -W 'CTF:archive_name[,DISCOVERY:FunctionSignatures.json]
[,ROUTES:ArchiveRoutes.json]' [options] mfilename1 mfilename2 ... mfilenameN

mcc -W 'mpsxl:addin_name,class_name,version' input_marshaling_flags
output_marshaling_flags [options] mfilename1 mfilename2 ... mfilenameN
```

Description

You can use `mcc` to package and deploy MATLAB programs as standalone applications, Excel add-ins, Spark applications, or Hadoop jobs.

If you have a MATLAB Compiler SDK license, you can use `mcc` to create C/C++ shared libraries, .NET assemblies, Java packages, Python packages, MATLAB Production Server deployable archives, or Excel add-ins for MATLAB Production Server.

General Usage

`mcc [options] mfilename1 mfilename2 ... mfilenameN` compiles the functions as specified by the options. The options used depend on the intended results of the compilation. The first file acts as the entry point to the compiled artifact.

You can also call this syntax from a system command prompt.

Note Arguments that contain special characters (such as period or space) must be surrounded by single quotes. Use double quotes when executing from a Windows command prompt.

`mcc(options,mfilename)` compiles the function as specified by the options. Specify file names and options as character vectors or strings. This syntax allows you to use MATLAB variables as input arguments.

Standalone Application

`mcc -m [options] mfilename` compiles the function into a standalone application. The executable type is determined by your operating system.

As an alternative, the `compiler.build.standaloneApplication` function supports most common workflows.

`mcc -e [options] mfilename` compiles the function into a standalone application that does not open a Windows command prompt on execution. The `-e` option works only on Windows operating systems.

As an alternative, the `compiler.build.standaloneWindowsApplication` function supports most common workflows.

Excel Add-In

`mcc -W 'excel:`*`addin_name`*`,`*`class_name`*`,version=`*`version_number`*`' [options] mfilename1 mfilename2 ... mfilenameN` creates a Microsoft Excel add-in using the specified files. Before creating Excel add-ins, install a supported compiler.

You can only create Excel add-ins on Windows.

- *`addin_name`* — Specifies the name of the add-in. If you do not specify the name, `mcc` uses *`mfilename1`* as the default.
- *`class_name`* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses *`addin_name`* as the class name. If specified, *`class_name`* must be different from *`mfilename1`*.

- *`version_number`* — Specifies the version number of the add-in file as *`major.minor.bug.build`* in the file system. You are not required to specify a version number. If you do not specify a version number, `mcc` sets the version number to `1.0.0.0` by default.
  - *`major`* — Specifies the major version number. If you do not specify a number, `mcc` sets *`major`* to `1`.
  - *`minor`* — Specifies the minor version number. If you do not specify a number, `mcc` sets *`minor`* to `0`.
  - *`bug`*— Specifies the bug fix maintenance release number. If you do not specify a number, `mcc` sets *`bug`* to `0`.
  - *`build`*— Specifies the build number. If you do not specify a number, `mcc` sets *`build`* to `0`.

As an alternative, the `compiler.build.excelAddIn` function supports most common workflows.

MapReduce Applications on Hadoop

Linux only

`mcc -W 'hadoop:`*`archive_name`*`,CONFIG:`*`config_file`*`' mfilename` generates a deployable archive from `mfilename` that can be run as a job by Hadoop.

- *`archive_name`* — Specifies the name of the generated archive.
- *`config_file`* — Specifies the path to the configuration file for creating a deployable archive. For more information, see “Configuration File for Creating Deployable Archive Using the mcc Command”.

Simulink Simulations

Requires Simulink Compiler

`mcc -m [options] mfilename` compiles a MATLAB application that contains a Simulink simulation into a standalone application. For more information, see “Create and Deploy a Script with Simulink Compiler” (Simulink Compiler).

Python Package

Requires MATLAB Compiler SDK

`mcc -W python:`*`package_name`*` [options] mfilename1 mfilename2 ... mfilenameN` creates a Python package using the specified files.

- *`package_name`* — Specifies the name of the Python package preceded by an optional namespace, which is a period-separated list such as `companyname.groupname.component`.

As an alternative, the `compiler.build.pythonPackage` function supports most common workflows.

.NET Assembly

Requires MATLAB Compiler SDK

`mcc -W 'dotnet:`*`assembly_name`*`,api=`*`api_type`*`,`*`class_name`*`,`*`framework_version`*`,`*`security`*`,`*`remote_type`*`' [options] mfilename1 mfilename2 ... mfilenameN` creates a .NET assembly with a single class using the specified files. Before creating .NET assemblies, see “MATLAB Compiler SDK .NET Target Requirements” (MATLAB Compiler SDK).

- *`assembly_name`* — Specifies the name of the assembly preceded by an optional namespace, which is a period-separated list such as `companyname.groupname.component`.
- *`api_type`* — Specifies the API type of the assembly. Values are `matlab-data` and `mwarray`. The default value is `mwarray`.
- *`class_name`* — 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 assembly. 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 assembly 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 assembly. Values are `remote` and `local`.

`mcc -W 'dotnet:`*`assembly_name`*`,api=`*`api_type`*`,`*`class_name`*`,`*`framework_version`*`,`*`security`*`,`*`remote_type`*`' [options] 'class{class_name:mfilename1,mfilename2,...,mfilenameN}'` creates a .NET assembly with multiple classes using the specified files. You can include additional class specifiers by adding `class{___}` arguments.

As an alternative, the `compiler.build.dotNETAssembly` function supports most common workflows.

Java Package

Requires MATLAB Compiler SDK

`mcc -W 'java:`*`package_name`*`,`*`class_name`*`' [options] mfilename1 mfilename2 ... mfilenameN` creates a Java package from the specified files. Before creating Java packages, see Configure Your Java Environment (MATLAB Compiler SDK).

- *`package_name`* — Specifies the name of the Java package preceded by an optional namespace, which is a period-separated list such as `companyname.groupname.component`.
- *`class_name`* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the last item in *`package_name`*.

`mcc -W 'java:`*`package_name`*`,`*`class_name`*`' [options] ' class{class_name:mfilename1,mfilename2,...,mfilenameN}'` creates a Java package with multiple classes from the specified files. You can include additional class specifiers by adding `class{___}` arguments.

As an alternative, the `compiler.build.javaPackage` function supports most common workflows.

C Shared Library

Requires MATLAB Compiler SDK

`mcc -l [options] mfilename1 mfilename2 ... mfilenameN` compiles the listed functions into a C shared library and generates C wrapper code for integration with other applications.

As an alternative, the `compiler.build.cSharedLibrary` function supports most common workflows.

C++ Shared Library

Requires MATLAB Compiler SDK

`mcc -W 'cpplib:`*`library_name`*`[,{all|legacy|generic}]' [options] mfilename1 mfilename2 ... mfilenameN` compiles the listed functions into a C++ shared library and generates C++ wrapper code for integration with other applications.

- *`library_name`* — Specifies the name of the shared library.
- `all`— Generates shared libraries using both the `mwArray` API and the generic interface that uses the MATLAB Data API. This is the default behavior.
- `legacy`— Generates shared libraries using the `mwArray` API.
- `generic`— Generates shared libraries using the MATLAB Data API.

As an alternative, the `compiler.build.cppSharedLibrary` function supports most common workflows.

COM Component

Requires MATLAB Compiler SDK

`mcc -W 'com:`*`component_name`*`,`*`class_name`*`' [options] mfilename1 mfilename2 ... mfilenameN` compiles the listed functions into a generic Microsoft COM component.

- *`component_name`* — Specifies the name of the COM component.
- *`class_name`* — Specifies the name of the class.

As an alternative, the `compiler.build.comComponent` function supports most common workflows.

`mcc -W 'com:component_name,`*`class_name`*`' [options] ' class{class_name:mfilename1,mfilename2,...,mfilenameN}'` creates a Microsoft COM component with multiple classes from the specified files. You can include additional class specifiers by adding `class{____}` arguments.

Deployable Archive for MATLAB Production Server

Requires MATLAB Compiler SDK

`mcc -U -W 'CTF:`*`archive_name`*`[,DISCOVERY:`*`FunctionSignatures.json`*`] [,ROUTES:`*`ArchiveRoutes.json`*`]' [options] mfilename1 mfilename2 ... mfilenameN` creates a deployable archive (`.ctf` file) for use with a MATLAB Production Server instance.

- *`archive_name`* — Specifies the name of the deployable archive.
- *`FunctionSignatures.json`* — Specifies the JSON file that contains information about your MATLAB functions, specified as an absolute or relative path. This options is relevant only for RESTful clients using the discovery API. For more information, see “MATLAB Function Signatures in JSON” (MATLAB Production Server).

- *`ArchiveRoutes.json`* — Specifies the JSON file that contains URL routes for mapping client requests to MATLAB web handler functions within the archive. Use this option to organize routes by deployable archive name instead of defining them all in the server-level routes file specified by the `routes-file` server configuration property. For more information on web request handlers, see “Handle Custom Routes and Payloads in HTTP Requests” (MATLAB Production Server).

The syntax also creates a server-side deployable archive (`.ctf` file) for Microsoft Excel add-ins.

As an alternative, the `compiler.build.productionServerArchive` function supports most common workflows, but it does not support creating archive-specific routes.

Excel Add-In for MATLAB Production Server

Requires MATLAB Compiler SDK

`mcc -W 'mpsxl:`*`addin_name`*`,`*`class_name`*`,`*`version`*`'` *`input_marshaling_flags`* *`output_marshaling_flags`* `[options] mfilename1 mfilename2 ... mfilenameN` creates a client-side Microsoft Excel add-in from the specified files that can be used to send requests to MATLAB Production Server from Excel. Creating the client-side add-in *must* be preceded by creating a server-side deployable archive (`.ctf` file) from the specified files. A purely client side add-in is not viable.

- *`addin_name`* — Specifies the name of the add-in.
- *`class_name`* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the *`addin_name`* as the default.
- *`version`* — Specifies the version of the add-in 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.
- *`input_marshaling_flags`* — Specifies options for how data is marshaled between Microsoft Excel and MATLAB.
  - `-replaceBlankWithNaN` — Specifies that a blank in Microsoft Excel is marshaled into NaN in MATLAB. If you do not specify this flag, blanks are marshaled into 0.
  - `-convertDateToString` — Specifies that dates in Microsoft Excel are marshaled into MATLAB character vectors. If you do not specify this flag, dates are marshaled into MATLAB doubles.
- *`output_marshaling_flags`* — Specifies options for how data is marshaled between MATLAB and Microsoft Excel.
  - `-replaceNaNWithZero` — Specifies that NaN in MATLAB is marshaled into a 0 in Microsoft Excel. If you do not specify this flag, NaN is marshalled into `#QNAN` in Visual Basic.
  - `-convertNumericToDate` — Specifies that MATLAB numeric values are marshaled into Microsoft Excel dates. If you do not specify this flag, Microsoft Excel does not receive dates as output.

As an alternative, the `compiler.build.excelClientForProductionServer` function supports most common workflows.

Examples

Create Standalone Application

Create a standalone application and include the data file `data.mat`.

In MATLAB, locate the MATLAB code that you want to deploy as a standalone application. For this example, compile using the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
```

Build a standalone application with a reference to `appFile` using the MATLAB command syntax. Use this command at the MATLAB command prompt.

```
mcc('-m',appFile,'-a','data.mat','-v')
```

The function generates a standalone application named `magicsquare`. The executable file type depends on the operating system on which the application is created.

As an alternative, the `compiler.build.standaloneApplication` function supports most common workflows.

Create Standalone Application with No Command Prompt (Windows only)

Create a standalone application on Windows that does not open a command prompt window on execution.

You can use the `mcc` command at the MATLAB command prompt or the Windows command window.

```
mcc -e myapp.mlapp -o VisualApp
```

The function generates a standalone Windows application named `VisualApp`.

As an alternative, the `compiler.build.standaloneWindowsApplication` function supports most common workflows.

Create Excel Add-In (Windows only)

Create an Excel add-in on Windows with the system level version number `5.2.1.7`.

To generate the Excel add-in file (`.xla`), you must enable "Trust access to the VBA project object model" in your Excel settings. If you do not do this, you can manually create the add-in by importing the generated `.bas` file into Excel.

```
mcc -W 'excel:magicExcel,myClass,version=5.2.1.7' -b mymagic.m
```

The function generates an Excel add-in named `magicExcel`.

As an alternative, the `compiler.build.excelAddIn` function supports most common workflows.

Create Hadoop MapReduce Application (Linux only)

Create a MapReduce application on a Linux system that can be run as a job by Hadoop.

Create a configuration file named `config.txt` in your work folder that specifies configuration info.

```
mw.ds.in.type = tabulartext
mw.ds.in.format = infoAboutDataset.mat
mw.ds.out.type = keyvalue
mw.mapper = maxArrivalDelayMapper
mw.reducer = maxArrivalDelayReducer
```

For more information, see “Configuration File for Creating Deployable Archive Using the mcc Command”.

Copy the example files `maxArrivalDelayMapper.m`, `maxArrivalDelayReducer.m`, and `airlinesmall.csv` located in the folder *`matlabroot`*`/toolbox/matlab/demos` to your work folder.

Compile the files using the `mcc` command.

```
mcc -W 'hadoop:maxArrivalDelay,CONFIG:config.txt' maxArrivalDelayMapper.m maxArrivalDelayReducer
```

The function generates a shell script named `run_maxarrivaldelay.sh`, a deployable archive named `maxArrivalDelayMapper.ctf`, and a readme file with usage details.

For more details, see “Include MATLAB Map and Reduce Functions into Hadoop Job”.

Create Simulink Simulation

Requires Simulink Compiler

Create a standalone application that runs a Simulink Simulation.

Create a Simulink model using Simulink. This example uses the model `sldemo_suspn_3dof`.

Create a MATLAB application that uses APIs from Simulink Compiler to simulate the model. For more information, see “Deploy Simulations with Tunable Parameters” (Simulink Compiler).

```
function deployParameterTuning(outputFile, mbVariable)

    if ischar(mbVariable) || isstring(mbVariable)
        mbVariable = str2double(mbVariable);
    end

    if isnan(mbVariable) || ~isa(mbVariable, 'double') || ~isscalar(mbVariable)
        disp('mb must be a double scalar or a string or char that can be converted to a double scalar');
    end

    in = Simulink.SimulationInput('sldemo_suspn_3dof');
    in = in.setVariable('Mb', mbVariable);
    in = simulink.compiler.configureForDeployment(in);
    out = sim(in);

    save(outputFile, 'out');

end
```

Use `mcc` to create a standalone application from the MATLAB application.

```
mcc -m deployParameterTuning.m
```

The function generates an executable named `deployParameterTuning`.

For more information, see “Create and Deploy a Script with Simulink Compiler” (Simulink Compiler).

Create Python Package

Requires MATLAB Compiler SDK

Create a Python package using the file `mymagic.m` located in the subfolder `pymagic`.

```
mcc -W python:magicPython pymagic/mymagic.m
```

The function generates a Python package named `magicPython`.

As an alternative, the `compiler.build.pythonPackage` function supports most common workflows.

Create .NET Assembly Using MATLAB Data API

Requires MATLAB Compiler SDK

Create a MATLAB Data API .NET assembly using the file `mymagic.m`. Specify a namespace, class name, framework version, security, and remoting type in the `-W` argument.

```
mcc -W 'dotnet:company.group.magic,api=matlab-data,dotnetClass,0.0,Private,local' mymagic.m
```

The function generates a .NET assembly archive named `magic.ctf`.

As an alternative, the `compiler.build.dotNETAssembly` function supports most common workflows.

Create .NET Assembly Using `mwArray`

Requires MATLAB Compiler SDK

Create a `mwArray` .NET assembly using the file `mymagic.m`. Specify a class name and framework version using the `-W` argument.

```
 mcc -W 'dotnet:magic,magicClass,5.0' mymagic.m
```

The function generates a .NET assembly named `magic.dll`.

As an alternative, the `compiler.build.dotNETAssembly` function supports most common workflows.

Create Java Package

Requires MATLAB Compiler SDK

Create a Java package using the file `mymagic.m`. Specify a class name and namespace using the `-W` argument.

```
mcc -W 'java:company.group.javamagic,magicClass' mymagic.m
```

The function generates a Java package named `magic.jar`.

As an alternative, the `compiler.build.javaPackage` function supports most common workflows.

Create C Shared Library

Requires MATLAB Compiler SDK

Create a C shared library using the file `mymagic.m`.

```
mcc -W lib:magiclibrary mymagic.m
```

The function generates a C shared library named `magic.dll`.

As an alternative, the `compiler.build.cSharedLibrary` function supports most common workflows.

Create C++ Shared Library Using MATLAB Data API

Requires MATLAB Compiler SDK

Create a C++ shared library that uses the MATLAB Data API using the file `mda_magic.m`.

```
mcc -W 'cpplib:matlabmagiccpp,generic' mda_magic.m
```

The function generates a C++ shared library archive named `matlabmagiccpp.ctf`.

As an alternative, the `compiler.build.cppSharedLibrary` function supports most common workflows.

Create C++ Shared Library Using `mwArray`

Requires MATLAB Compiler SDK

Create a C++ shared library that uses the `mwArray` API using the file `mwa_magic.m`.

```
mcc -W 'cpplib:mwarraymagiccpp,legacy' mwa_magic.m
```

The function generates a C++ shared library named `mwarraymagiccpp.dll`.

As an alternative, the `compiler.build.cppSharedLibrary` function supports most common workflows.

Create COM Component

Requires MATLAB Compiler SDK

Create a COM component using the files mymagic.m, data2.m, and data2.m. Use the class argument to map the magic function to a class named MagicClass and the data functions to a class named DataClass.

```
mcc -W com:magicCOM 'class{MagicClass:mymagic.m}' 'class{DataClass:data1.m,data2.m}'
```

The function generates a COM component named magicCOM_1_0.dll.

As an alternative, the compiler.build.comComponent function supports most common workflows.

Create Deployable Archive for MATLAB Production Server

Requires MATLAB Compiler SDK

Create a MATLAB Production Server archive using the file mymagic.m. Use the -W argument to specify JSON files that define function signatures and web handler route mappings. For more details on function signatures, see “MATLAB Function Signatures in JSON” (MATLAB Production Server). For more details on web handlers, see “Handle Custom Routes and Payloads in HTTP Requests” (MATLAB Production Server).

```
mcc -U -W 'CTF:mps_magic,DISCOVERY:magicFunctionSignatures.json,ROUTES:magicRoutes.json' mymagic
```

The function generates a deployable archive named mps_magic.ctf.

As an alternative, the compiler.build.productionServerArchive function supports most common workflows, but it does not support the ROUTES options for specifying archive-specific route mappings to web request handlers.

Create Excel add-in for MATLAB Production Server

Requires MATLAB Compiler SDK

Create a MATLAB Production Server archive using the file mymagic.m.

```
mcc -U -W CTF:mps_magic mymagic.m
```

Next, create an Excel add-in for MATLAB Production Server.

To generate the Visual Basic files, enable **Trust access to the VBA project object model** in Excel. If you do not do this, you can manually create the add-in by importing the .bas file into Excel.

```
mcc -W 'mpsxl:magicAddin,myExcelClass,version=1.0' mymagic.m
```

The function generates an Excel add-in named magicAddin.

As an alternative, the `compiler.build.excelClientForProductionServer` function supports most common workflows.

Input Arguments

Tip To view a table of `mcc` input arguments in alphabetical order, see “mcc Command Arguments Listed Alphabetically” on page A-2.

`mfilename` — File to be compiled
file name

File to be compiled, specified as a character vector or string scalar.

`mfilename1 mfilename2 ... mfilenameN` — Files to be compiled
list of file names

One or more files to be compiled, specified as a space-separated list of file names. The first file is used as the entry point for the compiled artifact.

`class{`*`class_name`*`:mfilename1,mfilename2,...,mfilenameN}` — Files to be included in a class
list of file names

One or more files to be included in the class *`class_name`*, specified as a comma-separated list of file names. You can include multiple class specifiers by adding additional `class{____}` arguments. The argument applies only to the COM component, Java package, and .NET assembly targets.

Target and Platform

`-W '`*`target:artifact_name[,options]`*`'` — Build target
`main` | `WinMain` | `excel` | `hadoop` | `spark` | `lib` | `cpplib` | `com` | `dotnet` | `java` | `python` | `CTF` | `mpsxl`

Build target and associated options, specified as one of the syntaxes listed below.

The compiler generates wrapper functions that allow another programming language to run the corresponding MATLAB function and any necessary global variable definitions.

Target	Syntax	Equivalent Option
Standalone Application	`-W 'main:`*`app_name`*`,version=`*`version`*`'`	`-m`
Standalone Application (no Windows console)	`-W 'WinMain:`*`app_name`*`,version=`*`version`*`'`	`-e`
Excel Add-In	`-W 'excel:`*`addin_name`*`,`*`class_name`*`,version=`*`version`*`'`	None

Target	Syntax	Equivalent Option
Hadoop MapReduce Application	`-W 'hadoop:`*`archive_name`*`,CONFIG:`*`configFile`*`'`	None
Spark Application	`-W 'spark:`*`app_name`*`,`*`version`*`'`	None

The following targets require MATLAB Compiler SDK.

Target	Syntax	Equivalent Option
C Shared Library	`-W 'lib:`*`library_name`*`'`	`-l`
C++ Shared Library	`-W 'cpplib:`*`library_name`*`[, {all|legacy|generic}]'`	None
COM Component	`-W 'com:`*`component_name`*`,`*`class_name`*`'`	None
.NET Assembly	`-W 'dotnet:`*`assembly_name`*`,api={matlab-data|mwarray},`*`class_name`*`,`*`framework_version`*`,`*`security`*`,{remote|local}'`	None
Java Package	`-W 'java:`*`package_name`*`,`*`class_name`*`'`	None
Python Package	`-W 'python:`*`package_name`*`,`*`class_name`*`'`	None
MATLAB Production Server Deployable Archive	`-W 'CTF:`*`archive_name`*`'`	None
MATLAB Production Server Excel Add-In	`-W 'mpsxl:`*`addin_name`*`,`*`class_name`*`,`*`version`*`'`	None

Note `-W` values that contain special characters, such as commas or periods, must be surrounded by single quotes. Use double quotes when executing from a Windows command prompt.

-T *phase:type* — Output target phase and type

`compile:exe` | `compile:lib` | `link:exe` | `link:lib`

Output target phase and type, specified as one of the following options. If not specified, `mcc` uses the default type for the target specified by the `-W` option.

Target	Description
`compile:exe`	Generate a C/C++ wrapper file and compile C/C++ files to an object form suitable for linking into a standalone application.

Target	Description
`compile:lib`	Generate a C/C++ wrapper file and compile C/C++ files to an object form suitable for linking into a shared library or DLL.
`link:exe`	Same as `compile:exe` and also link object files into a standalone application.
`link:lib`	Same as `compile:lib` and also link object files into a shared library or DLL.

Example: `-T link:lib`

`-A` *`arch`* — Add platform

`win64` | `maci64` | `glnxa64` | `maca64` | `all`

Add the platform designated by *`arch`* to the list of allowed platforms. At compile time, MATLAB Compiler automatically detects platform dependencies and restricts to compatible platforms. For instance, if you include a platform-specific MEX file, ordinarily you are restricted to the platform you compile on. If you implement code for the component to run on other platforms, you can allow the component to run on those platforms using `-A`.

Steps should be taken to ensure the component will successfully run on a given platform when using the `-A` option. For more information, see “Ensure Multiplatform Portability for Compiled Applications” (MATLAB Compiler SDK). Running the component on an incompatible platform will result in an unsupported platform error message that lists compatible platforms.

Valid platforms are `win64`, `maci64`, `glnxa64`, `maca64`, and `all`. To add multiple platforms, use multiple `-A` options.

The `-A` option only applies to targets that use a deployable archive (`.ctf` file). Applicable targets are web apps, .NET MATLAB Data API, C++ MATLAB Data API, Python, MATLAB Production Server, and MapReduce for Spark or Hadoop.

`-B` *`bundle[:parameters]`* — Options bundle file

`ccom` | `cexcel` | `cjava` | `cmpsxl` | `cpplib` | `csharedlib` | `dotnet` | file name

Specify an options bundle file, where *`bundle`* is the name of a file that contains a set of `mcc` command line options, arguments, filenames, and/or other `-B` options.

A bundle can include replacement parameters for compiler options that accept names and version numbers. If more than one parameter is passed, you must enclose the expression that follows the `-B` in single quotes. For example, `mcc -B 'cexcel:component,class,1.0' ....`

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

The following bundle files are located in *`matlabroot`*`\toolbox\compiler\bundles`.

Available Bundle Files

Bundle File	Target	Contents
`ccom`	COM component	`-W com:%1%,%2%,%3% -T link:lib`
`cexcel`	Excel Add-in	`-W excel:%1%,%2%,%3% -T link:lib -b -S`
`cjava`	Java package	`-W java:%1%,%2%`
`cmpsxl`	Excel Add-In for MATLAB Production Server	`-W mpsxl:%1%,%2%,%3% -T link:lib`
`cpplib`	C++ library	`-W cpplib:%1% -T link:lib`
`csharedlib`	C library	`-W lib:%1% -T link:lib`
`dotnet`	.NET assembly	`-W dotnet:%1%,%2%,%3%,%4%,%5% -T link:lib`

Standalone Applications

`-m` — Generate standalone application

Generate a standalone application. `-m` is equivalent to `-W main -T link:exe`. On Windows, the command prompt opens on execution of the application.

`-e` — Generate standalone Windows application

Generate a standalone Windows application that does not open a Windows command prompt on execution. `-e` is equivalent to `-W WinMain -T link:exe`.

This option works only on Windows operating systems.

`-o` ***executablename*** — Executable name

file name

Specify the name of the final executable of a standalone application. A suitable platform-dependent extension is added to the specified name (for example, `.exe` for Windows standalone applications).

Example: `-o myexecutable`

`-r` ***icon*** — Add icon resource

file path

Add icon resource to the executable of a standalone application. Paths can be relative to the current working directory or absolute.

Example: `-r path\to\icon.ico`

`-n` — Interpret command line inputs as MATLAB doubles

Interpret command line inputs as MATLAB doubles. If you do not specify this option, command line inputs are treated as MATLAB character vectors.

Excel Add-Ins and COM Components

`-b` — Generate Visual Basic file

Generate a Visual Basic file (`.bas`) and an Excel add-in file (`.xla`). The `.bas` file contains the Microsoft Excel Formula Function interface to the COM object generated by MATLAB Compiler. When imported into a workbook, this Visual Basic code allows the MATLAB function to be used as a cell formula function.

Note To generate the Excel add-in file (`.xla`), you must enable "Trust access to the VBA project object model" in your Excel settings.

`-u` — Register COM component for current user

Register COM component for the current user only on the development machine. The argument applies only to the generic COM component and Microsoft Excel add-in targets.

C Shared Libraries

`-l` — Generate C shared library

Generate a C shared library. `-l` is equivalent to `-W lib -T link:lib`.

`-c` — Suppress code linking

Suppress compiling and linking the generated C wrapper code. The `-c` option cannot be used independently of the `-l` option.

MATLAB Production Server

`-U` — Generate MATLAB Production Server archive

Generate a MATLAB Production Server archive (`.ctf` file). This argument must be before `mfilename`, and you must also specify the option `-W 'CTF:`*`archive_name`*`'`.

Additional Files

`-a` *`filepath`* — Add file or folder
file path

Add file or folder to the deployable archive. File paths can be relative or absolute. For additional details, see “Include and Access Files in Packaged Applications” on page 5-15.

To add multiple files, use multiple `-a` options, specify a folder, or use wildcards.

If a folder name is specified with the `-a` option, the entire contents of that folder are added recursively to the deployable archive. For example,

```
mcc -m hello.m -a ./testdir
```

specifies that all files in `testdir`, as well as all files in its subfolders, are added to the deployable archive. The folder subtree in `testdir` is preserved in the deployable archive.

If the file name includes the wildcard pattern (*), only the files in the folder that match the pattern are added to the deployable archive, and subfolders of the given path are not processed recursively.

For example, the following command adds all files in `./testdir` to the deployable archive, and subfolders under `./testdir` are not processed recursively.

```
mcc -m hello.m -a ./testdir/*
```

The following command adds all files with the extension `.m` in `./testdir`, and subfolders of `./testdir` are not processed recursively.

```
mcc -m hello.m -a ./testdir/*.m
```

Including Java Classes

If you use the `-a` flag 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.

`-h` ***`helpfile`*** — Add help text file

file path

Add a custom help text file. Paths can be relative to the current working directory or absolute. This option applies to standalone applications, C/C++ shared libraries, COM, and Excel targets.

Display help file contents by calling the application at the command line with the `-?` or `/?` argument.

Example: `-h path\to\helpfile`

`-X` — Exclude data files

Exclude data files read by common MATLAB file I/O functions during dependency analysis. For examples on how to use the `-X` option, see `%#exclude`. For more information, see “Dependency Analysis Using MATLAB Compiler” on page 5-2.

`-Z` ***`supportpackage`*** — Specify support packages

`autodetect` | `none` | support package

Specify the method of adding support packages to the deployable archive as one of the following options.

Syntax	Description
`-Z autodetect`	The dependency analysis process detects and includes the required support packages automatically. This is the default behavior of mcc.
`-Z none`	No support packages are included. Using this option can cause runtime errors.
`-Z '`*`packagename`*`'`	Only the specified support package is included. To specify multiple support packages, use multiple `-Z` inputs.

Example: `-Z 'Deep Learning Toolbox Converter for TensorFlow Models'`

mcc Build Options

`-d` *`outputfolder`* — Output folder

folder name

Place build output in the specified folder *`outputfolder`*. Paths can be relative to the current directory or absolute.

`-C` — Do not embed deployable archive

Do not embed the deployable archive in binaries. This option is ignored for Java libraries.

`-Y` *`licensefile`* — License file

file name

Override the default license file with the specified file *`licensefile`*. This option can only be used on the system command line.

`-?` — Display help text

Display `mcc` help text in the console.

Protect Source Code

`-j` — Obfuscate `.m` files

Obfuscate `.m` files. This option generates a P-code file with a `.p` extension for each `.m` file included in the `mcc` command before packaging.

P-code files are an obfuscated, execute-only form of MATLAB code. For more details, see `pcode`.

`-J` *`filename`* — Specify secret manifest JSON file

Specify a secret manifest JSON file to embed the specified secret keys in the deployable archive.

If your MATLAB code calls the `getSecret`, `getSecretMetadata`, or `isSecret` function, you must specify the secret keys to embed in the deployable archive in a JSON secret manifest file. If your code calls `getSecret` and you do not specify the `-J` option, `mcc` issues a warning and generates a template JSON file in the output folder named `<`*`component_name`*`>_secrets_manifest.json`. Modify this file by specifying the secret key names in the **Embedded** field.

The `setSecret` function is not deployable. To deploy secret keys, you must call `setSecret` in MATLAB before calling `mcc`.

For more information on deployment using secrets, see "Handle Sensitive Information in Deployed Applications" on page 18-17. For an example on deploying a standalone application with secret keys, see "Access Sensitive Information in Standalone Application" on page 18-14.

Example: `-J myapp_secrets_manifest.json`

`-k '`*`file=<keyfile>;loader=<mexfile>`*`'` — Specify encryption key and loader interface

key file and MEX-file

Specify an AES encryption key and a MEX file loader interface to retrieve the decryption key at runtime.

The key file must be in one of the following supported formats:

- Binary 256-bit AES key, with a 32 byte file size
- Hex encoded AES key, with a 64 byte file size

The loader MEX file must be an interface with the following arguments:

- `prhs[0]` — Input, char array specified as the static value `'get'`
- `prhs[1]` — Input, char array specified as the CTF component UUID
- `plhs[0]` — Output, 32 byte UINT8 numeric array or 64 byte hex encoded char array, depending on the key format

Avoid sharing the same key across multiple CTFs.

If you do not specify any arguments after `-k`, `mcc` generates a 256-bit AES key and a loader MEX file that can be used for demonstration purposes.

For C++ shared libraries, as an alternative to specifying both key and MEX loader at compile time, you can specify only the encryption key. You then provide the hex encoded 64 byte decryption key at runtime in your C++ application as an argument for the `initMATLABLibrary` function using the MATLAB Data API or the `<library>InitializeWithKey` function using the MWArray API. For this workflow, the syntax is `-k '<keyfile>'`.

Example: `-k 'file=path\to\encryption.key;loader=path\to\loader_interface.mexw64'`

`-s` — Obfuscate folder structures and file names

Obfuscate folder structures and file names in the deployable archive (`.ctf` file) from the end user. Optionally encrypt additional file types.

The `-s` option directs `mcc` to place user code and data contained in `.m`, `.mlapp`, `.p`, `.mat`, MLX, SFX, and MEX files into a user package within the CTF. During runtime, MATLAB code and data is decrypted and loaded directly from the user package rather than extracted to the file system. MEX files are temporarily extracted from the user package before being loaded.

To manually include additional file types in the user package, add each file type in a separate extension tag to the file `*matlabroot*/toolbox/compiler/advanced_package_supported_files.xml`.

The following are **not** supported:

- `ver` function
- Calling external libraries such as DLLs
- Out-of-process MATLAB Runtime (C++ shared library for MATLAB Data Array)
- Out-of-process MEX file execution (`mexhost`, `feval`, `matlab.mex.MexHost`)
- `.mat` files other than v7.3

MATLAB Runtime

`-R` *`option`* — Provide MATLAB Runtime options

`'-logfile,`*`filename`*`'` | `-nodisplay` | `-nojvm` | `'-startmsg,`*`message`*`'` | `'-completemsg,`*`message`*`'` | `-singleCompThread` | `-softwareopengl`

Provide MATLAB Runtime options that are passed to the application at initialization time. This option is only used when building standalone applications or Excel add-ins. For more details, see “MATLAB Runtime Startup Options” on page 8-2.

You can specify multiple `-R` options. When you specify multiple `-R` options, they are processed from left to right. For example, specify initialization start and end messages and a log file.

```
mcc -e -R '-logfile,bar.txt' -R '-startmsg,MATLAB Runtime initialized' -R '-completemsg,Initiali
```

Option	Description	Target
`'-logfile,`*`filename`*`'`	Specify a log file name. The file is created in the application folder at runtime and contains information about MATLAB Runtime initialization and all text piped to the command window. Option must be in single quotes. Use double quotes when executing the command from a Windows Command Prompt. If a file path is specified, the path must exist and must not include environment variables.	Standalone applications and Excel add-ins
`-nodisplay`	Suppress the MATLAB `nodisplay` run-time warning. On Linux, open MATLAB Runtime without display functionality.	Standalone applications and Excel add-ins
`-nojvm`	Do not use the Java Virtual Machine (JVM).	Standalone applications and Excel add-ins
`'-startmsg,`*`message`*`'`	Customizable user message displayed at initialization time. For more details, see “Display MATLAB Runtime Initialization Messages” on page 8-5.	Standalone applications
`'-completemsg,`*`message`*`'`	Customizable user message displayed when initialization is complete. For more details, see “Display MATLAB Runtime Initialization Messages” on page 8-5.	Standalone applications
`-singleCompThread`	Limit MATLAB to a single computational thread.	Standalone applications and Excel add-ins
`-softwareopengl`	Use Mesa Software OpenGL for rendering.	Standalone applications and Excel add-ins

Caution When running on macOS, if you use `-nodisplay` as one of the options included in `mclInitializeApplication`, then the call to `mclInitializeApplication` must occur before calling `mclRunMain`.

-S — Create single MATLAB Runtime instance

Create a single MATLAB Runtime instance that is shared across all class instances.

The standard behavior for the MATLAB Runtime is that every instance of a class gets its own MATLAB Runtime context. The 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 affect other instances of the same class.

In a singleton MATLAB Runtime, all instances of a class share the context. If multiple instances of a class are created, they use the context created by the first instance which saves startup time and some resources. However, any changes made to the global workspace or the base workspace by one instance impacts all class instances. For example, if `instance1` creates a global variable `A` in a singleton MATLAB Runtime, then `instance2` can use variable `A`.

Singleton MATLAB Runtime is only supported for the following specific targets.

Target supported by Singleton MATLAB Runtime	Usage Instructions
Excel add-in	Singleton MATLAB Runtime is the default behavior. You do not need to perform other steps.
.NET assembly	Singleton MATLAB Runtime is the default behavior. You do not need to perform other steps.
COM component	Using `mcc`, pass the `-S` flag.
Java package	

MATLAB Compiler Search Path

`-I` *`folder`* — Add folder to search path

folder

Add a new folder to the search path used by MATLAB Compiler during dependency analysis. Each `-I` option appends the folder to the beginning of the list of paths. For example, the following syntax sets up the search path so that `directory1` is searched first for MATLAB files, followed by `directory2`.

```
-I <directory1> -I <directory2>
```

This option is important for compilation environments where the MATLAB path is not available.

If used in conjunction with the `-N` option, the `-I` option adds the folder to the compilation path in the same position where it appeared in the MATLAB path, rather than at the head of the path. For more information, see “Dependency Analysis Function and User Interaction with the Compilation Path” on page 6-2.

`-N` — Clear path

Clear the search 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`
- *`matlabroot`*`\toolbox\shared\bigdata`

`-N` also retains all subfolders in this list that appear on the MATLAB path at compile time. This option 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`.

When using the `–N` option, use the `–I` option to force inclusion of a folder, which is placed at the head of the compilation path. Use the `–p` option to conditionally include folders and their subfolders; if they are present in the MATLAB path, they appear in the compilation path in the same order.

`-p` *`folder`* — Conditionally add folders to path
folder

Conditionally add specific folders and subfolders under *`matlabroot`*`\toolbox` to the compilation MATLAB path. The files are added to the beginning of the path in the same order in which they appear in the MATLAB path. You must use this option in conjunction with the option `-N`.

Use the syntax `-N -p` *`directory`*, where `directory` is a relative or absolute path to the folder to be included.

You can add more than one folder by specifying multiple `-p` entries. For example, add the folders `foo` and `bar`.

```
-N -p foo -p bar
```

- If a folder that is on the original MATLAB path is included with `-p`, the folder and all its subfolders that appear on the original path are added to the compilation path in the same order.
- If a folder that is not on the original MATLAB path is included with `-p`, that folder is ignored. (You can use `-I` to force its inclusion.)

mbuild Options

`-f` *`filename`* — mbuild options file
file name

Specify *`filename`* as the options file when calling `mbuild`. This option is a direct pass-through to `mbuild` that lets you use different ANSI compilers for different invocations of the compiler. If you specify an absolute path to the file on Windows, it must start with two forward slashes (`/`) and use forward slashes throughout.

This option specifically applies to the C/C++ shared libraries, COM, and Excel targets.

`-M` *`options`* — `mbuild` compile-time options

Define `mbuild` compile-time options. The *`options`* argument is passed directly to `mbuild`. This option provides a mechanism for defining compile-time options, for example, `-M "-Dmacro=value"`.

Note Multiple `-M` options do not accumulate; only the rightmost `-M` option is used.

To pass options such as `/bigobj`, delineate the string according to your platform.

Platform	Syntax
MATLAB	`-M 'COMPFLAGS=$COMPFLAGS /bigobj'`
Windows command prompt	`-M COMPFLAGS="$COMPFLAGS /bigobj"`

Platform	Syntax
Linux and macOS command prompt	`-M CFLAGS='$CFLAGS /bigobj'`

Complex Number Representation

Since R2018a, MATLAB uses an interleaved storage representation, where the real and imaginary parts of each number are stored together. However, when you generate C shared libraries based on `mxArray`, or C++ shared libraries based on `mwArray` using the `mcc` command, these libraries default to a separate storage representation for complex numbers, a representation that can degrade performance.

To use interleaved representation in these shared libraries, utilize the `-M R2018a` option.

For an optimal approach when generating C++ shared libraries, choose the MATLAB Data Array for C++, which inherently uses the interleaved storage representation.

When integrating shared libraries generated by `mcc` using the `mbuild` command, the flags should match. If `-M -R2018a` is passed to the `mcc` command, `-R2018a` should be passed to `mbuild`.

Debugging

`-G` — Include debug symbols

Include debugging symbol information for the code generated by MATLAB Compiler SDK. This option also causes `mbuild` to pass appropriate debugging flags to the system compiler. The debug option lets you backtrace up to the point where you can identify if the failure occurred in the initialization of MATLAB Runtime, the function call, or the termination routine. This option does not let you debug your MATLAB files with an external debugger.

This option is applicable to the C, C++, Java, Excel, and .NET targets.

`-K` — Keep partial output

Keep partial 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.

`-v` — Display verbose output

Display verbose output. Output displays the compilation steps, including:

- MATLAB Compiler version number
- Source file names as they are processed
- Names of the generated output files as they are created
- Invocation of `mbuild`

The `-v` option also passes the `-v` option to `mbuild` and displays information about `mbuild`.

`-w` ***`option[:warning]`*** — Control warning messages

`list` | `enable` | `disable` | `error` | `on` | `off`

Control the display of warning messages.

Syntax	Description
`-w list`	List the compile-time warnings that have abbreviated identifiers along with their status.
`-w enable[:<warning>]`	Enable specific compile-time warnings associated with `<warning>`. Omit the optional `<warning>` to apply the `enable` action to all compile-time warnings.
`-w disable[:<warning>]`	Disable specific compile-time warnings associated with `<warning>`. Omit the optional `<warning>` to apply the `disable` action to all compile-time warnings.
`-w error[:<warning>]`	Treat specific compile-time and runtime warnings associated with `<warning>` as an error. Omit the optional `<warning>` to apply the `error` action to all compile-time and runtime warnings.
`-w on[:<warning>]`	Turn on runtime warnings associated with `<warning>`. Omit the optional `<warning>` to apply the `on` action to all runtime warnings. This option is enabled by default.
`-w off[:<warning>]`	Turn off runtime warnings for specific error messages defined by `<warning>`. Omit the optional `<warning>` to apply the `off` action to all runtime warnings.

The `<warning>` argument can be either a full identifier, such as `Compiler:compiler:COM_WARN_OPTION_NOJVM`, or one of the abbreviated identifiers listed by `-w list`.

You can display the full identifier that corresponds to a warning by issuing the following statement in your MATLAB code after the warning takes place.

```
[msg, warnID] = lastwarn
```

If you specify multiple `-w` options, they are processed from left to right.

For example, disable all warnings except `repeated_file`.

```
-w disable -w enable:repeated_file
```

You can also turn warnings on or off globally. For example, to turn off warnings for all deployed applications, specify the following in `startup.m` using `isdeployed`.

```
if isdeployed
    warning off
end
```

Limitations

- `mcc` cannot create web apps. To create web apps, use the Web App Compiler app or the `compiler.build.webAppArchive` function.
- You can use `mcc` to build components on MATLAB Online Server™, which acts as a Linux environment. To build targets that require an external compiler, including Java, C++, and COM components, the MATLAB Online Server worker must be set up to run the required toolchain.

Tips

- On Windows, you can generate a system-level file version number for your target file by appending `version=`*`version_number`* to the target generating `mcc` syntax. For an example, see “Create Excel Add-In (Windows only)” on page 16-110.

  `version_number` — Specifies the version of the target file as *`major.minor.bug.build`* in the file system. You are not required to specify a version number. If you do not specify a version number, `mcc` sets the version number, by default, to `1.0.0.0`.

  - *`major`* — Specifies the major version number. If you do not specify a version number, `mcc` sets *`major`* to `1`.
  - *`minor`* — Specifies the minor version number. If you do not specify a version number, `mcc` sets *`minor`* to `0`.
  - *`bug`* — Specifies the bug fix maintenance release number. If you do not specify a version number, `mcc` sets *`bug`* to `0`.
  - *`build`* — Specifies build number. If you do not specify a version number, `mcc` sets *`build`* to `0`.

  This functionality is supported for standalone applications and Excel add-ins in MATLAB Compiler.

  This functionality is supported for C shared libraries, C++ shared libraries, COM components, .NET assemblies, and Excel add-ins for MATLAB Production Server in MATLAB Compiler SDK.

Version History

Introduced before R2006a

R2021b: Obfuscate folder structures and file names

Use the `-s` option to obfuscate folder structures and file names in the deployable archive (`.ctf` file) from the end user.

R2022b: Obfuscate MATLAB code

Use the `-j` option to convert all .m files to P-files before packaging before packaging.

R2022b: Specify AES encryption key

Use the `-k` option to specify a 256-bit AES encryption key and a MEX-file loader interface to retrieve the decryption key at runtime.

R2023b: Provide decryption key at runtime for C++ applications

For C++ shared libraries, as an alternative to specifying both key and MEX loader at compile time, you can specify only the encryption key with the `-k` option. You then provide the hex encoded 64 byte decryption key at runtime in your C++ application as an argument for the `initMATLABLibrary` function using the MATLAB Data API or the `<library>InitializeWithKey` function using the MWArray API. For this workflow, the syntax is `-k '<`*`keyfile`*`>'`.

R2024a: Package code with secrets

Use the `-J` option to include a JSON file that specifies secrets to embed within your deployable code archive.

R2025a: JSON file lists required and optional products
Behavior changed in R2025a

At compile time, mcc creates a JSON file named `buildresult.json` that contains information on required and optional runtime product dependencies. The information corresponds to the `RuntimeDependencies` property of the `compiler.build.Results` object. You can use the JSON file in place of `requiredMCRProducts.txt` in deployment workflows. The JSON file will replace `requiredMCRProducts.txt` in a future release.

See Also

`ismcc` | `isdeployed` | `compiler.build.Results`

Topics
"mcc Command Arguments Listed Alphabetically" on page A-2
"Write Deployable MATLAB Code" on page 5-10
"Include and Access Files in Packaged Applications" on page 5-15

mcrinstaller

Display version and location information for MATLAB Runtime installer corresponding to current platform

Syntax

```
[installer_path, major, minor, platform] = mcrinstaller
```

Description

`[installer_path, major, minor, platform] = mcrinstaller` displays information about available MATLAB Runtime installers.

If no MATLAB Runtime installer is found, you are prompted to download an installer using the command `compiler.runtime.download`.

You must distribute the MATLAB Runtime library to your end users to enable them to run applications developed with MATLAB Compiler or MATLAB Compiler SDK.

For more information about the MATLAB Runtime installer, see “Download and Install MATLAB Runtime” on page 7-4.

Examples

Find MATLAB Runtime Installer Location

Display the location of MATLAB Runtime installers for a particular platform. This example shows output for a `win64` system. The release number is called `R20xxx` indicating the release for which the MATLAB Runtime installer has been downloaded.

```
mcrinstaller
```

```
C:\Program Files\MATLAB\R20xxx\toolbox\compiler\deploy\win64\MCR_R20xxx_win64_installer.exe
```

For example, for R2018b, the path would be:

```
C:\Program Files\MATLAB\R2018b\toolbox\compiler\deploy\win64\MCR_R2018b_win64_installer.exe
```

Output Arguments

`installer_path` — Full path to the installer
character vector

The `installer_path` is the full path to the installer for the current platform.

`major` — Major version number
positive integer scalar

The `major` is the major version number of the installer.

`minor` — Minor version number
positive integer scalar

The `minor` is the minor version number of the installer.

`platform` — Name of the current platform
character vector

The `platform` is the name of the current platform (returned by `COMPUTER(arch)`).

Version History

Introduced in R2009a

See Also

`mcrversion` | `compiler.runtime.download`

Topics
"Download and Install MATLAB Runtime" on page 7-4

mcrversion

Return MATLAB Runtime version number that matches MATLAB version

Syntax

```
[major,minor] = mcrversion
```

Description

`[major,minor] = mcrversion` returns the MATLAB Runtime version number matching the version of MATLAB from where the command is executed. The MATLAB Runtime version number consists of two digits, separated by a decimal point. This function returns each digit as a separate output variable: `major`, `minor`.

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.

Examples

Return the MATLAB Runtime Version

Return the MATLAB Runtime Version Number Matching the Version of MATLAB.

```
[major, minor] = mcrversion
```

```
major =
     9
minor =
     9
```

Output Arguments

`major` — Major version number
positive integer scalar

Major version number returned as a positive integer scalar.

Data Types: `double`

`minor` — Minor version number
positive integer scalar

Minor version number returned as a positive integer scalar.

Data Types: `double`

Version History

Introduced in R2008a

See Also

`compiler.runtime.download` | `mcrinstaller`

Topics

"Download and Install MATLAB Runtime" on page 7-4

setmcruserdata

Associate MATLAB data value with a key

Syntax

```
void setmcruserdata(key, value)
```

Description

`void setmcruserdata(key, value)` associates the MATLAB data `value` with the string `key` in the current MATLAB Runtime instance. If there is already a `value` associated with the `key`, it is overwritten.

This function is part of the MATLAB Runtime User Data interface API. It is available both in MATLAB and in deployed applications created with MATLAB Compiler and MATLAB Compiler SDK.

Examples

Store a cell array and associate it with the string `'PI_Data'` in the current instance of the MATLAB Runtime.

```
value = {3.14159, 'March 14th is PI day'};
setmcruserdata('PI_Data', value);
```

Input Arguments

value — Value of MATLAB data
any MATLAB data type including matrices, cell arrays, and Java objects

`Value` is the MATLAB data associated with input string `key` for the current instance of the MATLAB Runtime.

key — Key associated with MATLAB data
string

`key` is a MATLAB string with which MATLAB data `value` is associated within the current instance of the MATLAB Runtime.

Version History

Introduced in R2008a

See Also

`getmcruserdata`

compiler.runtime.download

Download MATLAB Runtime installer

Syntax

```
compiler.runtime.download
```

Description

`compiler.runtime.download` downloads the MATLAB Runtime installer matching the version and update level of MATLAB from where the command is executed. If the installer has already been downloaded to the machine, it returns a message stating that the MATLAB Runtime installer exists and specifies its location. If the machine is offline, it returns a message that contains the download URL to the installer.

Examples

Download MATLAB Runtime Installer

Download the MATLAB Runtime installer on Windows using MATLAB R2025a.

```
compiler.runtime.download
```

```
Downloading MATLAB Runtime installer. It may take several minutes...

MATLAB Runtime installer has been downloaded to:
 "C:\Users\<username>\AppData\Local\MathWorks\MatlabRuntimeCache\MCRInstaller25.1\MATLAB_Runtime_
```

Display Location of MATLAB Runtime Installer

If you already have downloaded the latest version of the MATLAB Runtime installer, this command gives the following result on Windows using MATLAB R2025a:

```
compiler.runtime.download
```

```
An existing MATLAB Runtime installer was found at:
 "C:\Users\<username>\AppData\Local\MathWorks\MatlabRuntimeCache\MCRInstaller25.1\MATLAB_Runtime_
```

Display MATLAB Runtime Installer Download URL

If you are not connected to the internet, this command displays a URL that you can open in a web browser to download the MATLAB Runtime installer. The URL corresponds to the version and update level of MATLAB, as indicated by `<release>/Release/<update_level>`.

Using MATLAB R2025b with no updates, display the download URL for MATLAB Runtime.

```
compiler.runtime.download
```

```
Downloading MATLAB Runtime installer. It may take several minutes...

Error using compiler.runtime.download
A connection could not be established to download the Runtime Installer.
Download the runtime from:
https://ssd.mathworks.com/supportfiles/downloads/R2025b/Release/0/deployment_files/installer/comp
and update the runtime location in Compiler Settings.
```

Version History

Introduced in R2018a

See Also

`mcrinstaller` | `mcrversion`

MATLAB Compiler Quick Reference

mcc Command Arguments Listed Alphabetically

Option	Description	Comment
`-?`	Display `mcc` help message.	
`-a filepath`	Add `filepath` to the deployable archive.	If you specify a folder name, all files in the folder are added. If you use a wildcard (*), all files matching the wildcard are added.
`-A arch`	Add platforms to the list of supported platforms detected automatically by the compiler.	*`arch`* = `win64`, `maci64`, `maca64`, `glnxa64`, or `all`
`-b`	Generate Excel compatible formula function.	Requires MATLAB Compiler for Excel add-ins.
`-B bundle[:parameters]`	Replace `-B bundle` on the `mcc` command line with the contents of *`bundle`*.	The file should contain only `mcc` command-line options. MathWorks included bundle files are located in *`matlabroot`*`\toolbox\compiler\bundles`.
`-c`	Suppress compiling and linking of the generated C wrapper code.	Must be used in conjunction with the `-l` option.
`-C`	Direct `mcc` to not embed the deployable archive in generated binaries.	
`-d outputfolder`	Place output in folder specified by *`outputfolder`*.	
`-e`	Suppress appearance of the MS-DOS Command Window when launching the generated standalone application.	Use *`-e`* in place of the *`-m`* option. Available for Windows only. Equivalent to `-W WinMain -T link:exe`. The Standalone Application Compiler app suppresses the MS-DOS command window by default. To enable it, select **Standalone Windows Application** in the **Application Type** area.
`-f filename`	Use the specified options file, *`filename`*, when calling `mbuild`.	`mbuild -setup` is recommended. Valid for C/C++ shared libraries, COM, and Excel targets.
`-G`	Include debugging symbol information for generated C/C++ code.	
`-h helpfile`	Specify a custom help text file.	Display help file contents at runtime using `-?` or `/?`. Valid for standalone applications, C/C++ shared libraries, COM, and Excel targets.
`-I folder`	Add folder to search path for MATLAB files.	
`-j`	Automatically convert all `.m` files to P-files before packaging.	

Option	Description	Comment
`-J filename`	Specify a secret manifest JSON file to embed the specified secret keys in the deployable archive.	
`-k 'file=<keyfile>;loader=<mexfile>'`	Specify AES encryption key *`keyfile`* and MEX file loader interface *`mexfile`* to retrieve decryption key at runtime.	If you do not specify any arguments after `-k`, `mcc` generates a 256-bit AES key and a loader MEX file. For C++ shared libraries, you can specify only the encryption key and provide the decryption key in your C++ application at runtime.
`-K`	Direct `mcc` to not delete output files if the compilation ends prematurely, due to error.	Default behavior is to dispose of any partial output if the command fails to execute successfully.
`-l`	Create a C shared library.	Equivalent to `-W lib -T link:lib`.
`-m`	Generate a standalone application.	Equivalent to `-W main -T link:exe`. On Windows, the command prompt opens on execution of the application. The standalone app compiler suppresses the MS-DOS command window by default. To enable it, deselect **Do not display the Windows Command Shell (console) for execution** in the **Additional Runtime Settings** area.
`-M options`	Pass compile-time options to `mbuild`.	
`-n`	Automatically treat numeric inputs as MATLAB doubles.	Cannot be used in a compiler app.
`-N`	Clear the compile-time search path of all but a minimal, required set of folders.	Uses the following folders: • *`matlabroot`*`\toolbox\matlab` • *`matlabroot`*`\toolbox\local` • *`matlabroot`*`\toolbox\compiler` • *`matlabroot`*`\toolbox\shared\bigdata`
`-o executablename`	Specify name of standalone application executable file.	Adds appropriate extension.
`-p folder`	Add *`folder`* to compile-time search path in an order-sensitive context.	Requires `-N` option.
`-r icon`	Embed resource *`icon`* in binary.	Use to specify an application icon.

Option	Description	Comment
`-R option`	Specify run-time options for MATLAB Runtime.	Valid only for standalone applications and Excel add-ins. *`option`* = `-nojvm`, `-nodisplay`, `'-logfile,`*`filename`*`'`, `-startmsg`, and `-completemsg` *`filename`*
`-s`	Obfuscate folder structures and file names in the deployable archive (`.ctf` file) from the end user..	
`-S`	Create singleton MATLAB Runtime.	Default for generic COM components. Valid for Microsoft Excel and Java packages.
`-T phase:type`	Specify the output target phase and type.	Cannot be used in a compiler app.
`-u`	Register COM component for current user only on development machine.	Valid only for generic COM components and Excel add-ins.
`-U`	Generate a deployable archive (`.ctf` file) for MATLAB Production Server.	Equivalent to `-W 'CTF'`.
`-v`	Verbose; display compilation steps and warning messages.	
`-w option[:warning]`	Control warning messages.	Valid arguments are `list`, `enable[:`*`warning`*`]`, `disable[:`*`warning`*`]`, `error[:`*`warning`*`]`, `on[:`*`warning`*`]`, and `off[:`*`warning`*`]`.
`-W 'target[:options]'`	Specify build target and associated options.	*`target`* = `main`, `WinMain`, `excel`, `hadoop`, `spark`, `lib`, `cpplib`, `com`, or `dotnet`, `java`, `python`, `CTF`, or `mpsxl`.
`-X`	Ignore data files detected by dependency analysis.	For more information, see “Dependency Analysis Using MATLAB Compiler” on page 5-2.
`-Y licensefile`	Override the default license file with the specified file *`licensefile`*.	Can only be used on the system command line.
`-Z supportpackage`	Specify method of including support packages.	*`supportpackage`* = `'autodetect'` (default), `'none'`, or *`packagename`*.

Packaging Log and Output Folders

By default, the deployment app places the packaging log and all output files in the target folder location. If you specify a custom location, the app creates any folders that do not exist at compile time.

mcc Command Line Arguments Grouped by Task

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 Compiler)

Deployable Archive

Option	Description	Comment
`-a` *`path`*	Add `path` to the deployable archive.	If you specify a folder name, all files in the folder are added. If you use a wildcard (*), all files matching the wildcard are added.
`-C`	Directs `mcc` to not embed the deployable archive in C/C++ and main/Winmain shared libraries and standalone binaries by default.	None
`-h` *`filename`*	Specify a custom help text file.	Display help file contents at runtime using `-?` or `/?`. Valid for standalone applications, C/C ++ shared libraries, COM, and Excel targets.

Protect Source Code

Option	Description	Comment
`-j`	Automatically convert all `.m` files to P-files before packaging.	
`-k "file=<`*`key_file_path`*`>;loader=<`*`mex_file_path`*`>"`	Specify AES encryption key and MEX-file loader interface to retrieve decryption key at runtime.	If you do not specify any arguments after `-k`, `mcc` generates a 256-bit AES key and a loader MEX-file.
`-s`	Obfuscate folder structures and file names in the deployable archive (`.ctf` file) from the end user.	

Debugging

Option	Description	Comment
`-?`	Display help message.	None
`-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`

MATLAB Compiler for Excel Add-Ins

Option	Description	Comment
`-b`	Generate Excel compatible formula function.	Requires MATLAB Compiler. Cannot be used in a `deploytool` app.
`-u`	Registers COM component for current user only on development machine	Valid only for generic COM components and Microsoft Excel add-ins (requiring MATLAB Compiler)

MATLAB Path

Option	Description	Comment
`-I` *`directory`*	Add folder to search path for MATLAB files.	
`-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 Runtime

Option	Description	Comment
`-R` *`option`*	Specify run-time options for MATLAB Runtime.	*`option`* `= -nojvm -nodisplay '-logfile,`*`filename`*`' -startmsg -completemsg` *`filename`*
`-S`	Create Singleton MATLAB Runtime.	Default for generic COM components. Valid for Microsoft Excel and Java packages.

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) Cannot be used in a `deploytool` app.

Override Default Outputs

Option	Description	Comment
`-d` *`directory`*	Place output in specified folder.	None
`-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 display the Windows Command Shell (console) for execution** in the **Additional Runtime Settings** area.
`-o outputfile`	Specify name of final output file.	Adds appropriate extension

Wrappers and Libraries

Option	Description	Comment
`-c`	Suppress compiling and linking of the generated C wrapper code.	Must be used in conjunction with the `-l` option.
`-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`

Licenses

Option	Description	Comment
`-Y licensefile`	Use `licensefile` when checking out a MATLAB Compiler license.	The `-Y` flag works only with the command-line mode. `>>!mcc -m foo.m -Y license.lic`

17

Apps

Standalone Application Compiler

Package MATLAB programs for deployment as standalone applications

Description

The Standalone Application Compiler packages MATLAB programs into applications that can run outside of MATLAB. The interactive menus and dialog boxes used in the compiler apps build `compiler.build` commands that are customized to your specification.

Compiler app advantages include:

- You can perform deployment tasks with a single intuitive interface.
- You can organize your files in a MATLAB project.
- Your project state persists between sessions.
- You can load previously stored compiler tasks from a prepopulated menu.
- You can package applications with an installer for distribution.

Open the Standalone Application Compiler App

- MATLAB toolstrip: On the **Apps** tab, under **Application Deployment**, click the app icon.
- MATLAB command prompt: Enter `standaloneApplicationCompiler`.

Examples

- “Create Standalone Application Using Standalone Application Compiler App” on page 18-5

Version History

Introduced in R2025a

R2025a: Change in app entry point

Behavior changed in R2025a

In R2024b and earlier releases, you can generate standalone applications using the Application Compiler. Starting in R2025a, use the Standalone Application Compiler to create standalone applications.

R2025a: Change in output folder name

Behavior changed in R2025a

In R2024b and earlier releases, the compiler apps generate output files in the folders `for redistribution`, `for redistribution_files_only`, and `for testing`. Starting in R2025a, the compiler apps generate output files in a folder named after the compiler task.

See Also

Apps

Functions
`compiler.build.standaloneApplication` | `compiler.build.standaloneWindowsApplication`

Topics
"Create Standalone Application Using Standalone Application Compiler App" on page 18-5

Examples

Create Standalone Application from MATLAB

Supported Platforms: Windows, Linux, macOS

This example shows how to use MATLAB Compiler to package a function that prints a magic square to the command prompt. The target system does not require a licensed copy of MATLAB to run the application.

Note The application is not cross platform, and the executable type depends on the platform on which it was generated.

Create Function in MATLAB

In MATLAB, locate the MATLAB code that you want to deploy as a standalone application.

For this example, compile using the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

```
copyfile(fullfile(matlabroot,'extern','examples','compiler','magicsquare.m'));
```

```
function m = magicsquare(n)

if ischar(n)
    n=str2double(n);
end
m = magic(n);
disp(m)
```

In the MATLAB command window, enter `magicsquare(5);`.

The output is:

```
    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 Standalone Application Using compiler.build.standaloneApplication

Build a standalone application using a programmatic approach. Alternatively, if you want to create a standalone application package using a graphical interface, see “Create Standalone Application Using Standalone Application Compiler App” on page 18-5.

1 Build the standalone application using the `compiler.build.standaloneApplication` function.

   ```
   buildResults = compiler.build.standaloneApplication("magicsquare.m");
   ```

   You can specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.build.standaloneApplication`.

   The `compiler.build.Results` object `buildResults` contains information on the build type, generated files, included support packages, and build options.

The function generates the following files within a folder named `magicsquarestandaloneApplication` in your current working directory:

- `includedSupportPackages.txt` — Text file that lists all support files included in the application.
- `magicsquare.exe` or `magicsquare` — Executable file that has the `.exe` extension if compiled on a Windows system, or no extension if compiled on Linux or macOS systems.
- `run_magicsquare.sh` — Shell script file that sets the library path and executes the application. This file is only generated on Linux and macOS systems.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations on page 13-2.
- `readme.txt` — Text file that contains information on deployment prerequisites and the list of files to package for deployment.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

Note The generated files do not include an installer for the application or MATLAB Runtime. To create an installer using the `buildResults` object, see `compiler.package.installer`.

2. To test `magicsquare` from MATLAB with the input argument `4`, navigate to the `magicsquarestandaloneApplication` folder and execute one of the following commands based on your operating system:

Operating System	Test in MATLAB Command Window
Windows	`!magicsquare 4`
macOS	`system(['./run_magicsquare.sh ',matlabroot,' 4']);`
Linux	`!./magicsquare 4`

Run Standalone Application

1. In your system command prompt, navigate to the folder containing your standalone executable.
2. Run `magicsquare` with the input argument `5` by using one of the following commands based on your operating system:

Operating System	Command
Windows	`magicsquare 5`

Linux	Using the shell script:
	
	`./run_magicsquare.sh <MATLAB_RUNTIME_INSTALL_DIR> 5`
	
	Using the executable:
	
	`./magicsquare 5`

Operating System	Command

macOS	Using the shell script:
	
	`./run_magicsquare.sh <MATLAB_RUNTIME_INSTALL_DIR> 5`
	
	Using the executable:
	
	`./magicsquare.app/Contents/macOS/magicsquare 5`

Note To run the application without using the shell script on Linux and macOS, you must first add MATLAB Runtime to the library path. For more information, see “Set MATLAB Runtime Path for Deployment” on page 15-2.

3 The application outputs a 5-by-5 magic square 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
```

Tips

- To produce a standalone application that does not launch a Windows command shell, use `compiler.build.standaloneWindowsApplication`.
- To specify additional compilation options, you can use the `mcc` command to create a standalone application that does not include MATLAB Runtime or an installer.

See Also

`compiler.build.standaloneApplication` | `compiler.build.standaloneWindowsApplication` | `compiler.build.Results` | `compiler.package.installer` | Standalone Application Compiler | `mcc`

More About

- “Create Standalone Application Using Standalone Application Compiler App” on page 18-5
- “Download and Install MATLAB Runtime” on page 7-4
- “Set MATLAB Runtime Path for Deployment” on page 15-2
- “Install Deployed Application” on page 7-14

Create Standalone Application Using Standalone Application Compiler App

This example shows how to use the Standalone Application Compiler app to package a MATLAB function into a deployable standalone application that does not require MATLAB to run.

Create a standalone application using the previous version of the Standalone Application Compiler app as shown in Create Standalone Application from MATLAB Function Using Application Compiler App.

Create MATLAB Function

First, write MATLAB code to compile into a standalone application. You can compile a MATLAB function, class, or app of one of these file types: `.m`, `.p`, `.mlx`, `.mlapp`, or `.mex`. Your code must be in a finished state and ready for the end user to run. For more information, see “Write Deployable MATLAB Code” on page 5-10.

For this example, create a function file named `modfun.m` that contains this code.

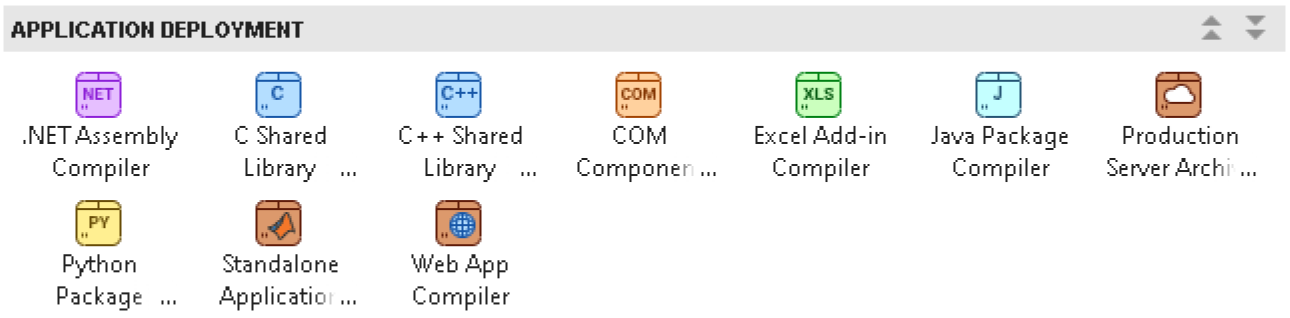
```
function modfun(m,n)
axis([-1 1 -1 1])
axis square
axis off
hold on
z = exp(2i*pi*(0:n)/n);
for j = 0:n
    zj = [z(j+1),z(mod(j*m,n)+1)];
    plot(real(zj),imag(zj))
end
end
```

The `modfun` function connects `n` equally spaced points around the complex unit circle with straight lines using multiples of `m` as the modulus. The `j`th line connects `z(j+1)` to `z(mod(j*m,n)+1)`.

Create Project and Compiler Task

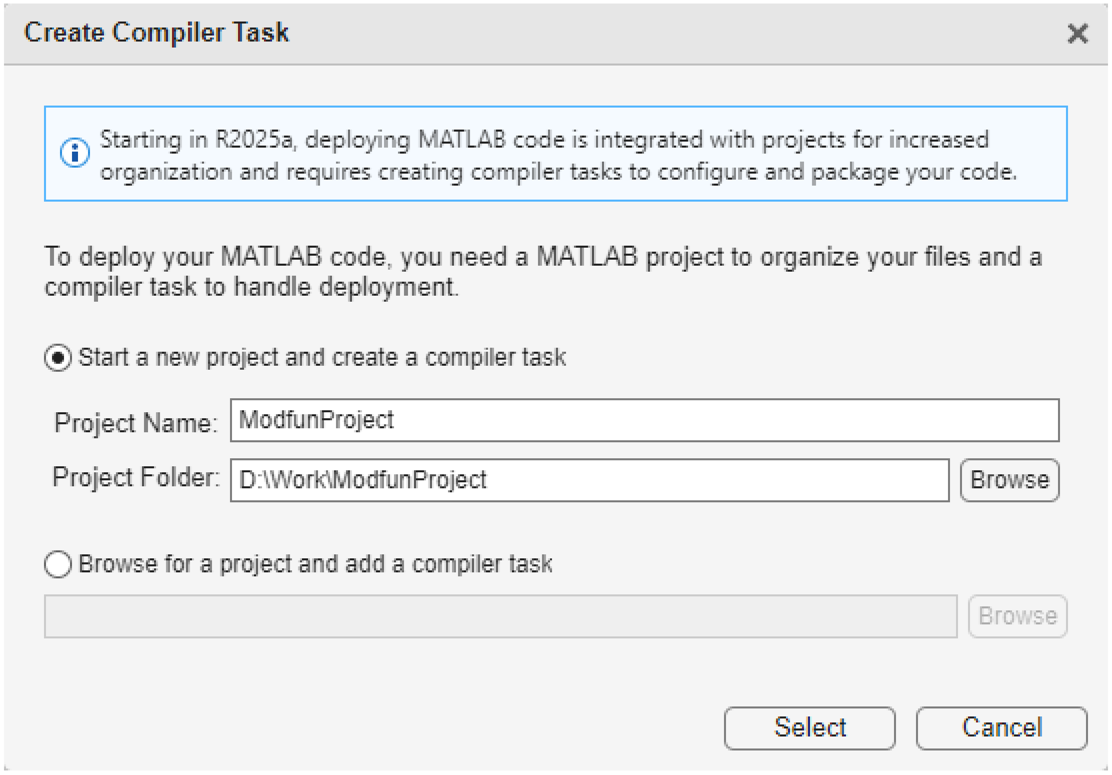
Create a compiler task for your function by using the Standalone Application Compiler app. Compiler tasks allow you to compile files in a project for a specific deployment target.

To open the app, on the **Apps** tab, expand the **Apps** gallery. In the **Application Deployment** section, click **Standalone Application Compiler**.



You can also open the app using the `standaloneApplicationCompiler` function at the MATLAB Command Window.

After you open the app, the Create Compiler Task dialog box prompts you to add a compiler task to a new or an existing MATLAB project. For this example, select **Start a new project and create a compiler task** and create a new project named `ModfunProject` in your working folder. For more information on creating and using MATLAB projects, see “Create Projects”.


Create Compiler Task

Starting in R2025a, deploying MATLAB code is integrated with projects for increased organization and requires creating compiler tasks to configure and package your code.

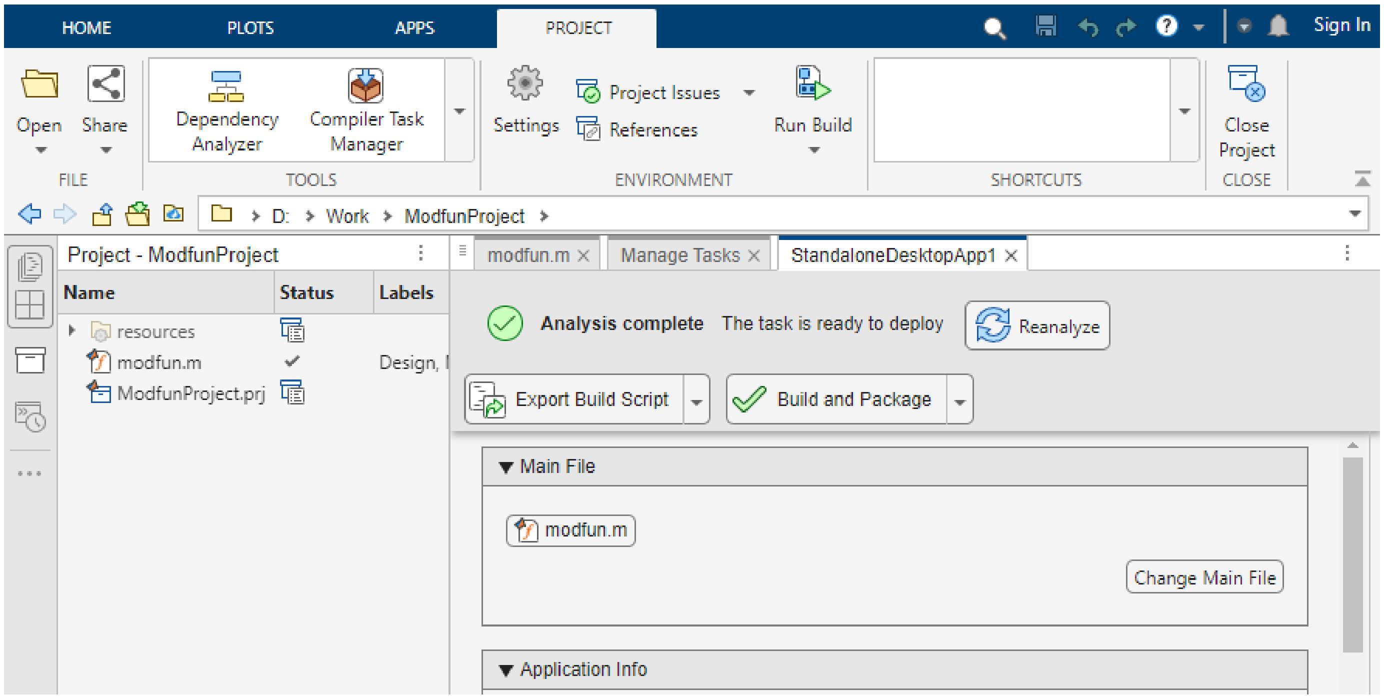
To deploy your MATLAB code, you need a MATLAB project to organize your files and a compiler task to handle deployment.

(●) Start a new project and create a compiler task

Project Name: ModfunProject

Project Folder: D:\Work\ModfunProject

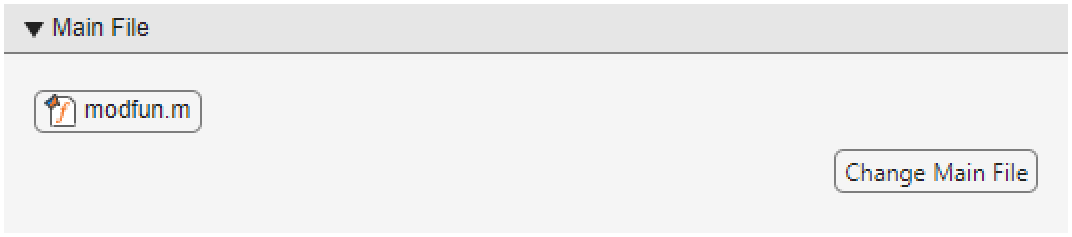
A new compiler task named `StandaloneDesktopApp1` opens in the Editor. You can compile code for other deployment targets by opening the Compiler Task Manager app or going to the **Manage Tasks** tab and creating a new compiler task.



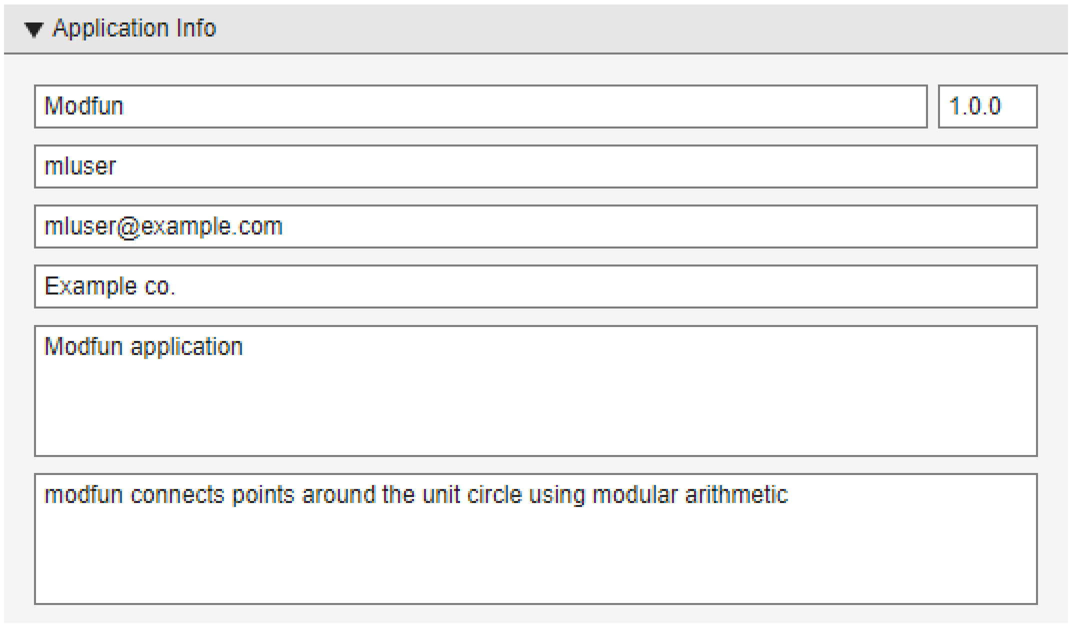
Specify Build Options

You can specify options for the standalone application and its installer before packaging to customize the building and packaging process. For instance, you can add a splash screen or icon, obfuscate the MATLAB code, and specify the method of including MATLAB Runtime in the generated installer.

For this example, in the **Main File** section of the compiler task, click **Add Main File** and select `modfun.m`. In the Project panel, the file now has the labels `Design` and `Main Function`.

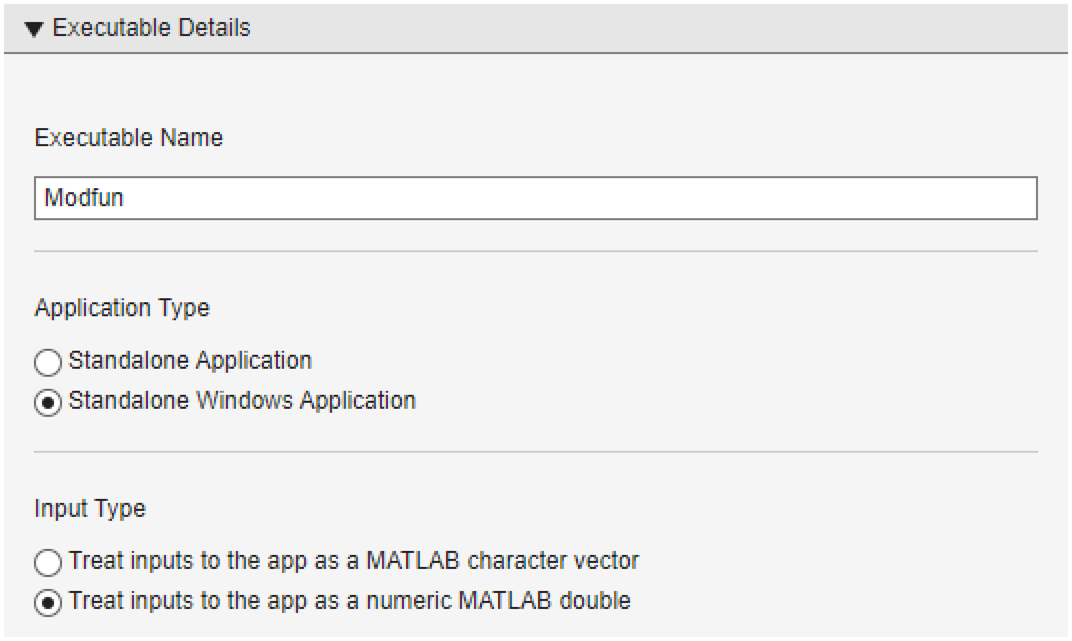


In the **Application Info** section, replace the string `My Desktop Application` with the name for your standalone application, `Modfun Application`. You can specify other details in this section, such as the author, company, and description.



Modfun
1.0.0
mluser
mluser@example.com
Example co.
Modfun application
modfun connects points around the unit circle using modular arithmetic

In the **Executable Details** section, name the executable `Modfun`. Under **Application Type**, select **Standalone Windows Application** if you are using Windows; otherwise, select **Standalone Application**. Under **Input Type**, select **Treat inputs to the app as a numeric MATLAB double**.

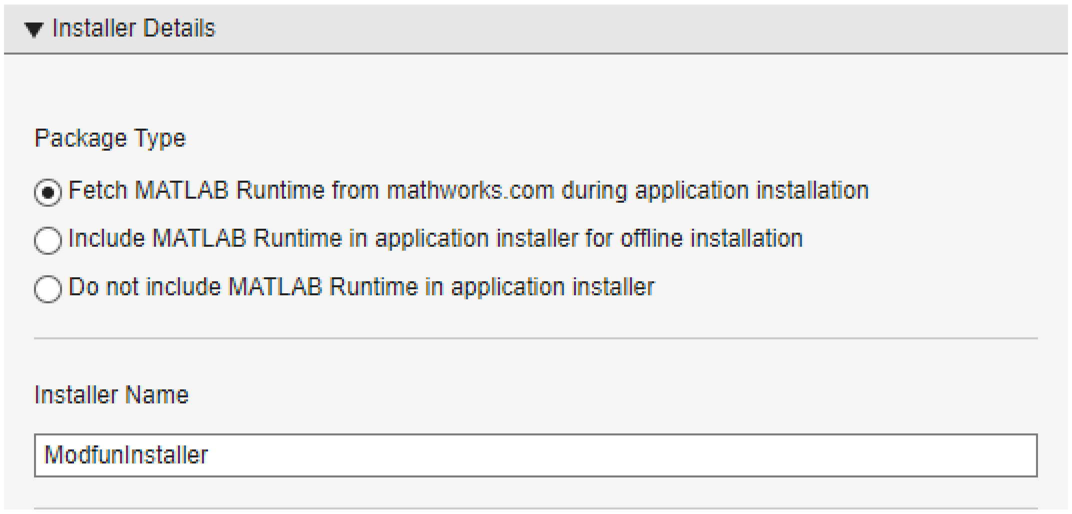


Executable Name: Modfun

Application Type
- Standalone Application
- [✓ Standalone Windows Application]

Input Type
- Treat inputs to the app as a MATLAB character vector
- [✓ Treat inputs to the app as a numeric MATLAB double]

In the **Installer Details** section, name the generated installer `ModfunInstaller`. You can also specify other installer options such as the runtime delivery method. These options correspond to the options available with the `compiler.package.installer` function.


▼ Installer Details

Package Type

● Fetch MATLAB Runtime from mathworks.com during application installation

○ Include MATLAB Runtime in application installer for offline installation

○ Do not include MATLAB Runtime in application installer

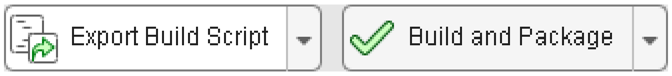
Installer Name

ModfunInstaller

View Code and Package Standalone Application

To view code that contains instructions on building and packaging your component, click the arrow next to the **Export Build Script** button and select **Show Code**. On the right, a window opens that displays a deployment script with the `compiler.build.standaloneApplication` or `compiler.build.standaloneWindowsApplication` and `compiler.package.installer` functions that correspond to your build options.

You can convert this code to a MATLAB script file by clicking the **Export Build Script** button. Running the generated build script is equivalent to clicking the **Build and Package** button.



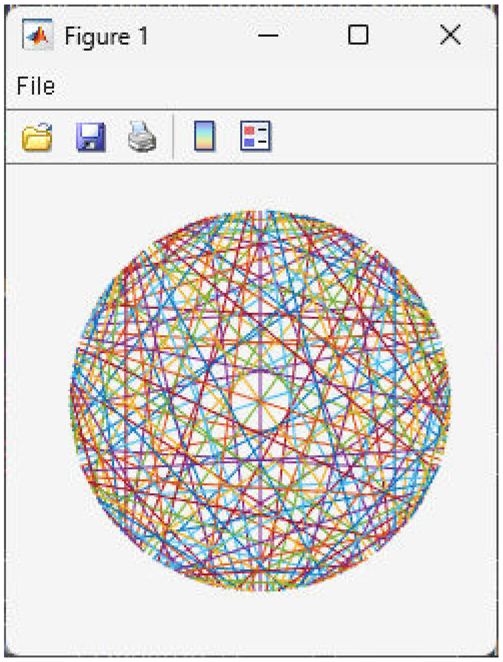
To generate both the standalone application and an installer, click **Build and Package**. To create the application executable without an installer, click **Build and Package** > **Build**.

The compiler generates files in the `<`*`compiler_task_name`*`>/output` folder in your project folder. The `build` subfolder contains the standalone application executable, and the `package` subfolder contains an installer for your standalone application along with MATLAB Runtime. To choose a different output location for the generated files, update the paths in the **Output Locations** section.

Test and Deploy Application

You can test the application in MATLAB before deployment. Run the executable in the MATLAB Command Window using the bang operator and the inputs 111 and 200 to view the figure.

```
!D:\Work\ModfunProject\StandaloneDesktopApp1\output\build\Modfun.exe 111 200
```



To deploy your application outside of MATLAB, you must have MATLAB Runtime installed at the same version as the MATLAB version used to build the application. Ensure that your end users can run the application by including MATLAB Runtime in the installer or by supplying users with information on how to download it. For information on installing and using MATLAB Runtime, see “About MATLAB Runtime” on page 7-2.

For details on installing your standalone application, see “Install Deployed Application” on page 7-14. For more information on deployment, see “Steps for Deployment with MATLAB Compiler” on page 1-3.

See Also

`compiler.build.standaloneApplication` | `compiler.build.standaloneWindowsApplication` | `compiler.package.installer`

Related Examples

- “Steps for Deployment with MATLAB Compiler” on page 1-3
- “Install Deployed Application” on page 7-14
- “Create Standalone Application from MATLAB” on page 18-2
- “About MATLAB Runtime” on page 7-2

Deploy Parallel-Enabled MATLAB Function as Standalone Application

This example shows how to deploy a Monte-Carlo simulation that runs in parallel as a standalone application using MATLAB® Compiler™.

When you deploy a MATLAB® function that runs in parallel as a standalone application, the cluster profile of the cluster where the code runs must be available to the MATLAB function. You can set the cluster profile by:

- Including the path to the cluster profile in the MATLAB function code.
- Including the cluster profile as an additional file when packaging the MATLAB function into a standalone application.
- Passing the cluster profile to the standalone application at run time.

You can use the `setmcruserdata` function to set the cluster profile within the MATLAB function and the `mcruserdata` option to pass the cluster profile to the standalone application at run time. For details, see “Use Parallel Computing Toolbox in Deployed Applications” on page 3-6.

In this example, the cluster profile is included as an additional file when packaging the MATLAB function into a standalone application. For details on how to pass the cluster profile at run time, see Pass Cluster Profile at Run Time on page 18-13.

Create MATLAB Function

Write a MATLAB function named `mcsim` that runs a simple stochastic simulation based on the dollar auction. The function runs multiple simulations to find the market value for a one dollar bill using a Monte-Carlo method. Save the function in a file named `mcsim.m`. For details, see “Use parfor to Speed Up Monte-Carlo Code” (Parallel Computing Toolbox).

```
function B = mcsim(nTrials, nPlayers, incr, dropoutRate)

%% specify cluster profile
mpSettingsPath = which('deployLocal.mlsettings');
setmcruserdata('ParallelProfile', mpSettingsPath);

%% parfor-loop to produce nTrials samples
% store the last bid from each trial in B
t = zeros(1,5);
for j = 1:5
    tic
    parfor i = 1:nTrials
        bids = dollarAuction(nPlayers,incr,dropoutRate);
        B(i) = bids.Bid(end);
    end
    t(j) = toc;
end
parforTime = min(t);

%% display results
disp(['parforTime = ' num2str(parforTime)])
disp(['Average market value = ' num2str(mean(B))])
```

In this example, the cluster profile is included as an additional file when packaging the MATLAB function into a standalone application. Since any files added to the application while packaging are added to the application's search path, using the `which` command in the above code returns the absolute path to the cluster profile. You can then use the `setmcruserdata` function to set the cluster profile using the path to the cluster profile.

Create Standalone Application Using `compiler.build.standaloneApplication`

Use the `compiler.build.standaloneApplication` function to create a standalone application from the MATLAB function.

```
appFile = "mcsim.m";
parallelProfileFile = fullfile(pwd,'deployLocal.mlsettings');
buildResults = compiler.build.standaloneApplication(appFile,...
    "AdditionalFiles",parallelProfileFile,...
    "TreatInputsAsNumeric",'on')
```

```
buildResults = 
  Results with properties:

                  BuildType: 'standaloneApplication'
                      Files: {2×1 cell}
    IncludedSupportPackages: {}
                    Options: [1×1 compiler.build.StandaloneApplicationOptions]
```

The function generates the executable within a folder named `mcsimstandaloneApplication` in your current working directory.

NOTE:

1 The generated standalone application is not cross platform, and the executable type depends on the platform on which it was generated.
2 The generated standalone application does not include MATLAB Runtime or an installer. To create an installer using the `buildResults` object, use the `compiler.package.installer` function.

Test Standalone Application

To run `mcsim` from within MATLAB, navigate to the `mcsimstandaloneApplication` folder from within the MATLAB desktop environment and execute one of the following commands based on your operating system:

Windows

```
!mcsim.exe 10000 20 0.05 0.01
```

```
Starting parallel pool (parpool) using the 'deployLocal' profile ...
Connected to parallel pool with 4 workers.
parforTime = 5.3772
Average market value = 5.8624
```

Linux

```
!./mcsim 10000 20 0.05 0.01
```

macOS

```
system(['./run_mcsim.sh', matlabroot, '10000','20', '0.05', '0.01']);
```

Run Standalone Application

In your system command prompt, navigate to the folder containing your standalone executable.

Run `mcsim` using one of the following commands based on your operating system:

Windows

```
mcsim.exe 10000 20 0.05 0.01
```

Linux

Using the shell script:

```
./run_mcsim.sh <MATLAB_RUNTIME_INSTALL_DIR> 10000 20 0.05 0.01
```

Using the executable:

```
./mcsim 10000 20 0.05 0.01
```

macOS

Using the shell script:

```
./run_mcsim.sh <MATLAB_RUNTIME_INSTALL_DIR> 10000 20 0.05 0.01
```

Using the executable:

```
./mcsim.app/Contents/macOS/mcsim 10000 20 0.05 0.01
```

Pass Cluster Profile at Run Time

To pass the cluster profile at run time, use the `mcruserdata` option when executing the application.

```
mcsim 10000 20 0.05 0.01 -mcruserdata ParallelProfile:'<path>\clusterProfile.mlsettings'
```

See Also

`setmcruserdata`

Related Examples

- “Use Parallel Computing Toolbox in Deployed Applications” on page 3-6

Access Sensitive Information in Standalone Application

In this section...

"Store SFTP Credentials in Local MATLAB Vault" on page 18-14
"Specify Secrets in Secret Manifest JSON File" on page 18-15
"Write MATLAB Code to Deploy" on page 18-15
"Create Standalone Application Using mcc" on page 18-15
"Run Application" on page 18-15

This example shows you how to create an application that retrieves stored secret values to access an SFTP server.

You can avoid exposing sensitive information, such as passwords, in your application code by storing secrets in your MATLAB vault. For more information on using secrets in deployment, see "Handle Sensitive Information in Deployed Applications" on page 18-17.

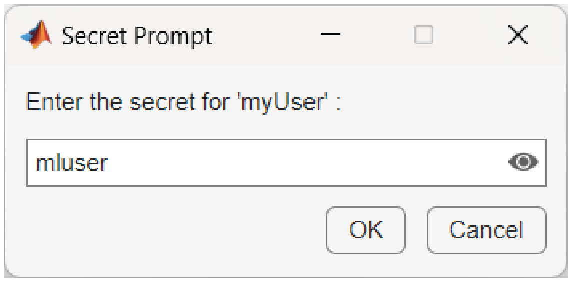
Store SFTP Credentials in Local MATLAB Vault

At the MATLAB command prompt, store your SFTP server credentials in your local MATLAB vault by calling the `setSecret` function with the name of your secret. The secret name is a unique case-sensitive text identifier for the secret, which is stored **unencrypted** in your vault as a string scalar.

1 Store the username using `setSecret`.

```
setSecret("myUser")
```

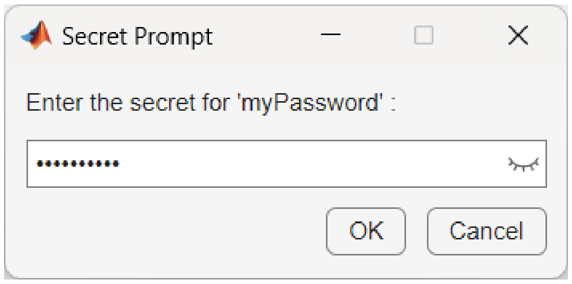
Use the Secret Prompt dialog box to set your username secret value and add the secret to your MATLAB vault. Toggle the eye icon in the text box to hide or show the characters in the field.



2 Store the password using `setSecret`.

```
setSecret("myPassword")
```

Use the dialog box to set your password secret value.



Specify Secrets in Secret Manifest JSON File

In your working folder, create a secrets manifest file named `secrets_manifest.json` that specifies which secrets in the MATLAB vault to embed in the deployable archive. For this example, embed the secrets named `myUser` and `myPassword`.

```
{
     "Embedded": {
       "description": "All secret names specified in this section will be put into the deployed C
       "secret": ["myUser", "myPassword"]
     }
}
```

Write MATLAB Code to Deploy

Write MATLAB code to package into a standalone application.

Save the following MATLAB code as `secretapp.m`. Replace the SFTP server `sftp.example.net` with your SFTP server hostname.

```
s = sftp("sftp.example.net",getSecret("myUser"),Password=getSecret("myPassword"))
close(s)
```

The code uses the `getSecret` function to retrieve your username and password from the vault. It connects to the SFTP server by calling the `sftp` function, which creates an SFTP connection object. After displaying information about the server connection, the connection is closed.

Create Standalone Application Using mcc

Package the code into a standalone application using `mcc`. Use the `mcc -J` option to specify the JSON secret manifest file.

```
mcc -m secretapp.m -J secrets_manifest.json
```

`mcc` generates a standalone application named `secretapp` in your working directory. The file extension depends on the platform used to generate the application.

Note The generated standalone executable does not include MATLAB Runtime or an installer. To create an installer that installs the application and MATLAB Runtime, use the `compiler.package.installer` function.

Run Application

You can test the application in MATLAB using the system command syntax.

```
!secretapp
```

```
  SFTP with properties:

                      Host: "sftp.example.net"
                  Username: "mluser"
                      Port: 22
              ServerSystem: "Windows"
```

```
                 DatetimeType: "datetime"
                 ServerLocale: "en_US"
                 DirParserFcn: @matlab.io.ftp.parseDirListingForWindows
      RemoteWorkingDirectory: "/home/mluser"
```

If you want to run the application on another machine, you must install MATLAB Runtime at the same update level or newer. For more information, see “Download and Install MATLAB Runtime” on page 7-4.

Note To run the application without using the generated shell script on Linux and macOS, you must first add MATLAB Runtime to the library path. For details, see “Set MATLAB Runtime Path for Deployment” on page 15-2.

See Also

`mcc` | `getSecret` | `setSecret` | `compiler.package.installer`

Related Examples

- “Handle Sensitive Information in Deployed Applications” on page 18-17

Handle Sensitive Information in Deployed Applications

You can increase the security of your application code by storing sensitive information, such as passwords, as secrets in your MATLAB vault.

When you set a secret value in MATLAB, it is stored in your local MATLAB vault. The MATLAB vault provides encrypted and persistent storage for secrets. Your vault and secrets persist across MATLAB sessions. You can store secrets in your MATLAB vault using the `setSecret` function and list currently stored secrets using `listSecrets`.

A secret can be any sensitive information that you would like to store securely in an encrypted form. Each secret consists of a name, value, and optional metadata.

- Secret name — A unique case-sensitive text identifier for the secret. The secret name is stored **unencrypted** in your vault as a string scalar.
- Secret value — A text value associated with the secret. The Secret Prompt dialog box, where you enter the secret value, supports copy-paste functionality. The secret value is stored **encrypted** in your vault using industry standard AES-256 encryption. The secret value is returned as a string scalar.
- Secret metadata — A dictionary containing additional information associated with the secret. Metadata can aid in the identification, usage, and lifecycle management of the secret. The optional secret metadata is stored **unencrypted** in your vault.

For example, this secret contains the following database credentials:

- Secret name — `"databasePassword"`
- Secret value — `"CpyA/&qRFzB2$X*jf"`
- Secret metadata — dictionary (string ⟼ cell) with 3 entries:

  `"databaseName"` ⟼ `{["productionDB"]}`

  `"host"` ⟼ `{["db.example.com"]}`

  `"port"` ⟼ `{["5432"]}`

For more information on secrets and the MATLAB vault, see “Keep Sensitive Information Out of Code”.

Package Code with Secrets

If the MATLAB code you want to deploy handles sensitive information, you can use the `getSecret` function in your application code to retrieve a secret value, which is decrypted at run time.

These functions that manage secrets are deployable:

- `getSecret` – Retrieve a secret from your vault.
- `getSecretMetadata` – Retrieve metadata of a secret in your vault.
- `isSecret` – Determine if a secret exists in your vault.

All other secret management functions, including `setSecret`, are not deployable.

Package Secrets in Deployable Archive

You can use the functionality provided by the MATLAB vault in standalone applications by including secrets in the deployable archive.

To package secrets with a standalone application, you specify the secret names in a secrets manifest JSON file using the `mcc -J` option. You can also use the `-J` flag in the **Additional Runtime Settings** area of the compiler apps.

You can specify a secrets manifest file with a `compiler.build` function using the `SecretsManifest` option.

For MATLAB Compiler to retrieve secrets from your local MATLAB vault and embed them in the deployable code archive at compile time, you must call `setSecret` in MATLAB to store each secret in your vault before you package your code.

For an example on creating a standalone application that uses secrets, see “Access Sensitive Information in Standalone Application” on page 18-14.

Store Secret Values as Environment Variables

As an alternative to packaging secrets within the archive, you can store secret values in environment variables on the target platform. For instance, if your deployed code runs in a container, you can set the environment variables when you create the container instance. Access secrets stored in environment variables using the `getSecret` function, specifying the environment variable name as the secret name.

In the instance where a secret stored in your vault shares a name with an environment variable, `getSecret` retrieves the value of the environment variable.

Access Secrets on MATLAB Web App Server

On MATLAB Web App Server, secrets are stored in the server vault. To retrieve and use secrets in a web application, call the `getSecret` function in the application code.

The Web App Server administrator can add, remove, or modify secrets stored in the server vault. To manage secrets on MATLAB Web App Server, the administrator can use one of these options:

- Use `webapps-secrets` at the command line.
- Use the graphical interface of the development version of MATLAB Web App Server. For details, see “Configure the Development Version of MATLAB Web App Server in MATLAB Compiler”.

Note **Security Considerations:** On MATLAB Web App Server, the vault file is configured by the Web App Server administrator, who has read and write permissions. Web app worker processes do not have access to this file. Server processes have read permission.

The MATLAB Web App Server also provides functionality to define attribute-based access control rules. These rules enable authenticated individuals to retrieve secrets from the server vault.

By activating policy-based access to secrets on the server, the server administrator can tailor secret access configurations for individual users. This feature is useful for managing secrets across various applications and their respective user bases. It allows web apps to access secret values at run time, for instance, to retrieve unique credentials on a per-user basis.

For information about secrets access control, see “Control Secrets Access in MATLAB Web App Server” (MATLAB Web App Server).

See Also

`setSecret` | `getSecret` | `isSecret` | `webapps-secrets`

Related Examples

- “Keep Sensitive Information Out of Code”
- “Access Sensitive Information in Standalone Application” on page 18-14
- “Configure the Development Version of MATLAB Web App Server in MATLAB Compiler”
- “Control Secrets Access in MATLAB Web App Server” (MATLAB Web App Server)