MATLAB® Compiler™
User's Guide

MATLAB®

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

Revision History

Contents

Customizing a Compiler Project

4

MATLAB Code Deployment

5

Standalone Application Creation

6

Deployment Process

7

8

Work with the MATLAB Runtime

9

Distributing Code to an End User

10

Compiler Commands

11

Standalone Applications

12

Troubleshooting

Functions

16

MATLAB Compiler Quick Reference

A

Using MATLAB Compiler on Mac or Linux

B

Apps

17

Getting Started

- "MATLAB Compiler Product Description" on page 1-2
- "Appropriate Tasks for MATLAB Compiler Products" on page 1-3
- "Create Standalone Application from MATLAB" on page 1-5

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™.

Appropriate Tasks for MATLAB Compiler Products

MATLAB Compiler generates standalone applications and Excel add-ins. MATLAB Compiler SDK generates C/C++ shared libraries, deployable archives for use with MATLAB Production Server, Java® packages, .NET assemblies, and COM components.

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

Task	MATLAB Compiler and 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 applications for deployment to MATLAB Production Server	■				
Build non-MATLAB applications that include MATLAB functions	■				
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			■		

Task	MATLAB Compiler and MATLAB Compiler SDK	MATLAB Coder™	Simulink®	HDL Coder™	MATLAB External Interfaces
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 and 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.

Create Standalone Application from MATLAB

Supported Platform: Windows®, Linux®, macOS

This example shows how to use MATLAB Compiler to package the pre-written 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.

You can create standalone applications using the following options:

Option	Purpose
Application Compiler	Use this app to produce an installer that installs both the standalone application and all required dependencies on a target system.
`compiler.build.standaloneApplication`	Use this function to produce a standalone application that does not include MATLAB Runtime or an installer. To produce a standalone application that does not launch a Windows command shell, use `compiler.build.standaloneWindowsApplication`.
`compiler.package.installer`	Use this function to produce an installer that installs both the standalone application and all required dependencies on a target system.
`mcc`	Use this function to produce a standalone application that does not include MATLAB Runtime or an installer.

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.

```
function m = magicsquare(n)

if ischar(n)
    n=str2double(n);
end
m = magic(n);
```

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 Application Compiler App

1. On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Application Compiler**.
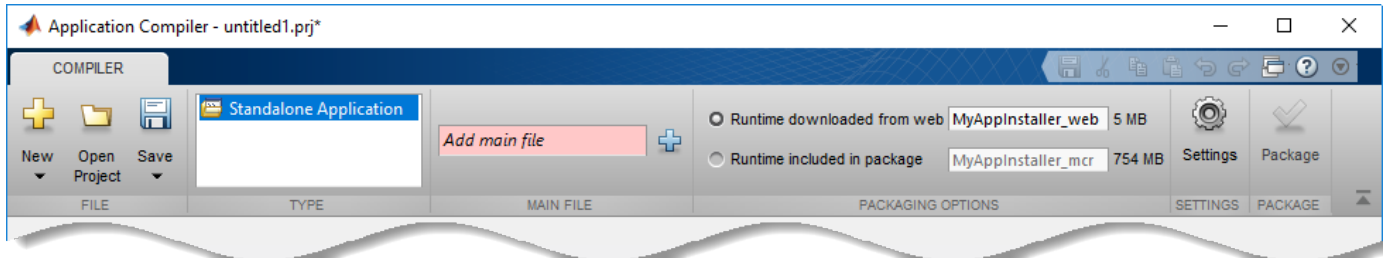
   Alternately, you can open the **Application Compiler** app by entering `applicationCompiler` at the MATLAB prompt.

2. In the **MATLAB Compiler** project window, specify the main file of the MATLAB application that you want to deploy.

   a. In the **Main File** section of the toolstrip, click .
   b. In the **Add Files** window, browse to *matlabroot*`\extern\examples\compiler`, and select `magicsquare.m`. Click **Open**.

      The function `magicsquare.m` is added to the list of main files.

3. Decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the two options in the **Packaging Options** section:

   - **Runtime downloaded from web** — Generates an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application.

   - **Runtime included in package** — Generates an installer that includes the MATLAB Runtime installer.

4. Customize the packaged application and its appearance:

- **Application information** — Editable information about the deployed application. You can also customize the standalone applications appearance by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata. See "Customize the Installer" on page 4-2.

- **Command line input type options** — Selection of input data types for the standalone application. For more information, see "Determine Data Type of Command-Line Input (For Packaging Standalone Applications Only)" on page 4-4.

- **Additional installer options** — Edit the default installation path for the generated installer and selecting custom logo. See "Change the Installation Path" on page 4-3 .

- **Files required for your application to run** — Additional files required by the generated application to run. These files are included in the generated application installer. See "Manage Required Files in Compiler Project" on page 4-4.

- **Files installed for your end user** — Files that are installed with your application. These files include:

- Generated `readme.txt`
- Generated executable for the target platform

See "Specify Files to Install with Application" on page 4-6.

- **Additional runtime settings** — Platform-specific options for controlling the generated executable. See "Additional Runtime Settings" on page 4-7.

Caution On Windows operating systems, when creating a console only application, uncheck the box **Do not display the Windows Command Shell (console) for execution**. By default, this box is checked. If the box is checked, output from your console only application is not displayed. Since this example is a console only application, the box must be unchecked.

---

5 To generate the packaged application, click **Package**.

In the Save Project dialog box, specify the location to save the project.

6 In the **Package** dialog box, verify that **Open output folder when process completes** is selected.

When the packaging process is complete, examine the generated output.

- Three folders are generated in the target folder location: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`.

  For further information about the files generated in these folders, see "Files Generated After Packaging MATLAB Functions" on page 3-11.

- `PackagingLog.html` — Log file generated by MATLAB Compiler.

7 To install your standalone application, see "Install Standalone Application" on page 1-11.

Create Standalone Application Using compiler.build.standaloneApplication

---

Note If you have already created a standalone application using the **Application Compiler** app, you can skip this section. However, if you want to know how to create a standalone application from the MATLAB command window using a programmatic approach, follow these instructions.

---

1 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');
```

2 Build the standalone application using the `compiler.build.standaloneApplication` function.

```
buildResults = compiler.build.standaloneApplication(appFile);
```

The `compiler.build.Results` object `buildResults` contains information on the build type, generated files, and build options.

3 The function generates the following files within a folder named `magicsquarestandaloneApplication` in your current working directory:

- `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 standalone executable does not include MATLAB Runtime or an installer.

4 Additional options can be specified by using one or more comma-separated pairs of name-value arguments in the `compiler.build` command.

- `'AdditionalFiles'` — Paths to additional files to include in the standalone application.
- `'AutoDetectDataFiles'` — Flag to automatically include data files.
- `'CustomHelpTextFile'` — Path to a file containing help text for the end user of the application.
- `'EmbedArchive'` — Flag to embed the standalone archive in the generated executable.
- `'ExecutableIcon'` — Path to a custom icon image.
- `'ExecutableName'` — Name of the generated application.
- `'ExecutableSplashScreen'` — Path to a custom splash screen image. This option is only used when you compile using the `compiler.build.standaloneWindowsApplication` function.
- `'ExecutableVersion'` — System level version of the generated application. This is only used on Windows systems.
- `'OutputDirectory'` — Path to the output directory that contains generated files.
- `'TreatInputsAsNumeric'` — Flag to interpret command line inputs as numeric MATLAB doubles.
- `'Verbose'` — Flag to display progress information indicating compiler output during the build process.

For example, you can specify the executable name and enable verbose output.

```
buildResults = compiler.build.standaloneApplication(appFile,...
    'ExecutableName','MyMagic','Verbose','On');
```

5 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`

6 To run your standalone application outside of MATLAB, see "Run Standalone Application" on page 1-11.

Create Standalone Application Installer Using the compiler.package.installer Function

---

Note If you have already created a standalone application installer using the **Application Compiler** app, you can skip this section. However, if you want to know how to create an installer for a standalone application from the MATLAB command prompt using a programmatic approach, follow these instructions.

---

1   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');
```

2   Build the standalone application using the `compiler.build.standaloneApplication` function and save the output in a `compiler.build.Results` object.

```
buildResults = compiler.build.standaloneApplication(appFile);
```

3   Create an installer by using the `Results` object `buildResults` as an input argument to the `compiler.package.installer` function.

```
compiler.package.installer(buildResults);
```

The function creates a new folder that contains the standalone application installer.

4   Additional options can be specified as one or more comma-separated pairs of name-value arguments in the `compiler.package.installer` command.

- `'ApplicationName'` — Name of the installed application.
- `'AuthorCompany'` — Name of company that created the application.
- `'AuthorEmail'` — Email address of the application author.
- `'AuthorName'` — Name of the application author.
- `'DefaultInstallationDir'` — Default directory in which the application is installed.
- `'Description'` — Detailed description of the application.
- `'InstallationNotes'` — Notes about additional requirements for using the application.
- `'InstallerIcon'` — Path to an image file used as the installer's icon.
- `'InstallerLogo'` — Path to an image file used as the installer's logo.
- `'InstallerName'` — Name of the generated installer.
- `'InstallerSplash'` — Path to an image file used as the installer splash screen.
- `'OutputDir'` — Path to the folder that contains the generated installer.
- `'RuntimeDelivery'` — Method to install MATLAB Runtime, specified as one of the following:
  - `'web'` (default) — Option for installer to download MATLAB Runtime during application installation.
  - `'installer'` — Option to include MATLAB Runtime within the installer so that it can be installed without connecting to the Internet.
- `'Shortcut'` — Path to a file or folder that the installer will create a shortcut to at install time. This is only used on Windows.
- `'Summary'` — Summary of the application.

- `'Version'` — Version number of the generated application.

For example, you can specify the installer name and include MATLAB Runtime in the installer.

```
compiler.package.installer(buildResults,...
    'InstallerName','MyMagic_Install','RuntimeDelivery','installer');
```

Install Standalone Application

- To install your application using an installer created by the **Application Compiler** app or the `compiler.package.installer` function, see "Install Deployed Application" on page 7-11.

Run Standalone Application

1 In your system command prompt, navigate to the folder containing your standalone executable.
2 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 Run-Time Deployment" on page 15-2.
3 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`

macOS	Using the shell script:
	
	`./run_magicsquare.sh <MATLAB_RUNTIME_INSTALL_DIR> 5`
	
	Using the executable:
	
	`./magicsquare.app/Contents/macOS/ magicsquare 5`

4 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
```

5 To create a command line shortcut for the application on Linux or macOS, use the `alias` command.

```
alias mymagic='/path/to/run_magicsquare.sh <MATLAB_RUNTIME_INSTALL_DIR>'
```

To run your application with the input argument 4, type `mymagic 4` in the terminal.

6   To make an alias permanent, append the command to the file `~/.bash_aliases` in a Bash shell or `~/.zshrc` in a Zsh shell.

```
echo "alias mymagic='~/MATLAB/apps/run_magicsquare.sh /usr/local/MATLAB/MATLAB_Runtime/v910'"
```

See Also

`applicationCompiler` | `compiler.build.standaloneApplication` | `compiler.build.standaloneWindowsApplication` | `compiler.package.installer` | `deploytool` | `mcc`

More About

- "Install and Configure MATLAB Runtime" on page 7-3
- "Set MATLAB Runtime Path for Run-Time Deployment" on page 15-2

MATLAB Runtime Additional Info

Differences Between MATLAB and MATLAB Runtime

MATLAB Runtime differs from MATLAB in several important ways:

- In MATLAB Runtime, MATLAB files are encrypted and immutable.

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

- The MATLAB Runtime is version-specific. You must run your applications with the version of the MATLAB Runtime associated with the version of MATLAB Compiler SDK with which it was created. For example, if you compiled an application using version 6.3 (R2016b) of MATLAB Compiler, users who do not have MATLAB installed must have version 9.1 of the MATLAB Runtime installed. Use `mcrversion` to return the version number of the 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.

Performance Considerations and MATLAB Runtime

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

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.

Calls into the MATLAB Runtime are serialized so calls into the MATLAB Runtime are threadsafe. This can impact performance.

Deploying Standalone Applications

Standalone Applications and Arguments

In this section...
"Overview" on page 3-2
"Pass File Names, Numbers or Letters, Matrices, and MATLAB Variables" on page 3-2
"Run Standalone Applications that Use Arguments" on page 3-2

Overview

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

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

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

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

To Pass....	Use This Syntax....	Notes
A file named `helpfile`	`filename helpfile`	
Numbers or letters	`filename 1 2 3 a b c`	Do *not* use commas or other separators between the numbers and letters you pass.
Matrices as input	`filename "[1 2 3]" "[4 5 6]"`	Place double quotes around input arguments to denote a blank space.

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

Run Standalone Applications that Use Arguments

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

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

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

Using SYSTEM, DOS, or UNIX

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

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

Using the ! (Bang) Operator

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

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

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

Using a Windows System

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

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

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

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

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

Using a MATLAB File You Plan to Deploy

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

- The input arguments you pass to your executable from a system prompt are received as character vector input. Thus, if you expect the data in a different format (for example, double), you must first convert the character vector input to the required format in your MATLAB code. For example, you can use STR2NUM to convert the character vector input to numerical data.

- You cannot return values from your standalone application to the user. The only way to return values from compiled code is to either display it on the screen or store it in a file.

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

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

Taking Input Arguments and Displaying to a Screen Using a MATLAB File

Here are two ways to use a MATLAB file to take input arguments and display data to the screen:

Method 1

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

if ischar(z)
z=str2num(z);
else
z=z;
end
x=2*z
y=z^2;
disp(y)
```

Method 2

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

if isdeployed
z=str2num(z);
end
x=2*z
y=z^2;
disp(y)
```

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, by default 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 a Cluster Profile

To export a cluster profile to an external file:

1 In the Home tab, in the **Environment** section, select **Parallel > Manage Cluster Profiles**.

2 In the **Cluster Profile Manager** dialog, select a profile, and in the **Manage** section, click **Export**.

Link to a 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 as 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 you application code, you can use the *myClusterProfile.mlsettings* file found within the MATLAB preferences folder as follows:

```
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 as follows:

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

Note that 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 as follows:

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

Note that this is a good practice if the user of your standalone application should supply their own cluster profile. Any file that you add with the `-a` flag when compiling with `mcc` is added to the

MATLAB search path. Therefore, you can also bundle a cluster profile with your application that is held externally. 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 need to 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

```
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;
}
```

Integrate Application with Mac OS X Finder

In this section...
"Overview" on page 3-7
"Installing the Mac Application Launcher Preference Pane" on page 3-7
"Configuring the Installation Area" on page 3-7
"Running the Application" on page 3-9

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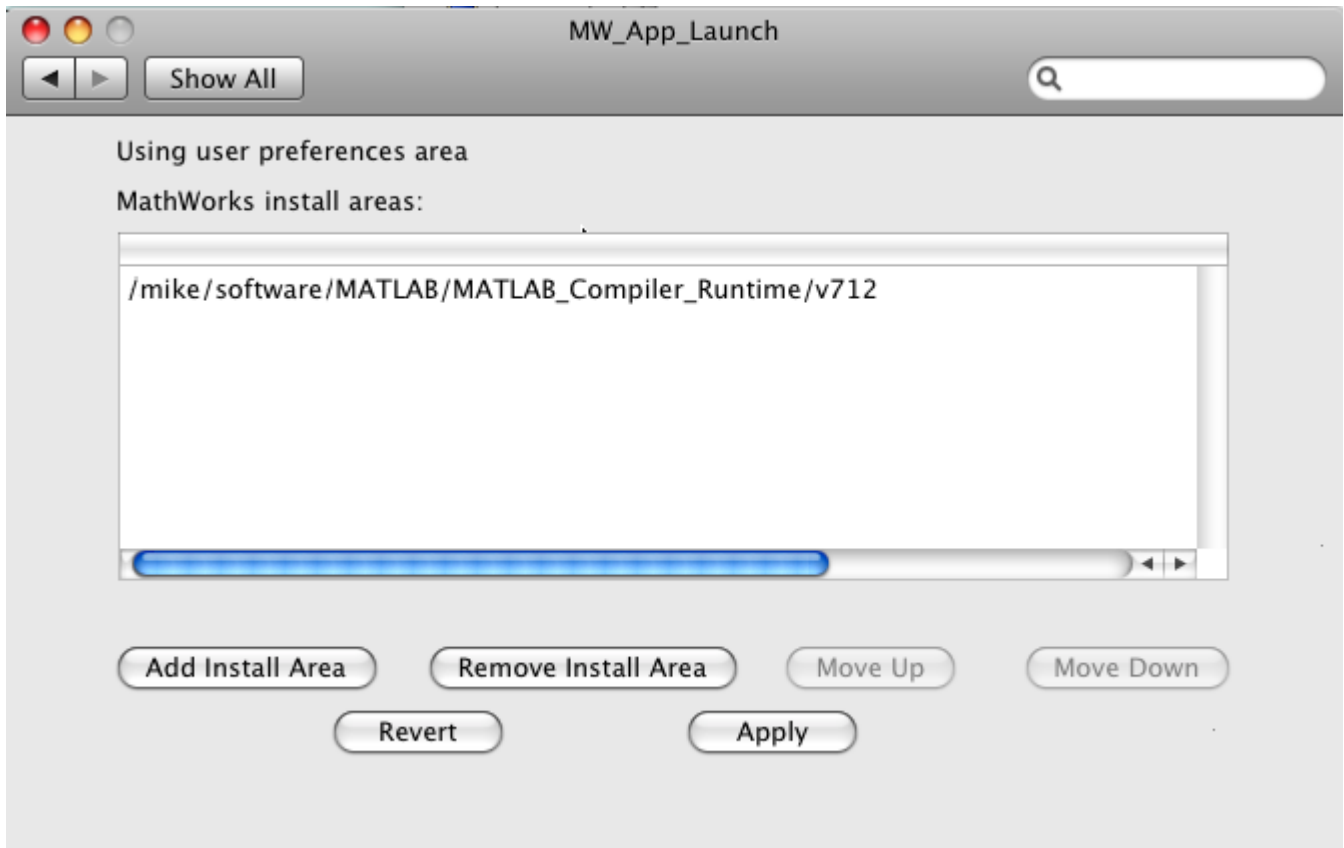
3   Define an installation area on your system by clicking **Add Install Area**.

4   Define the default installation path by browsing to it.

5   Click **Open**.
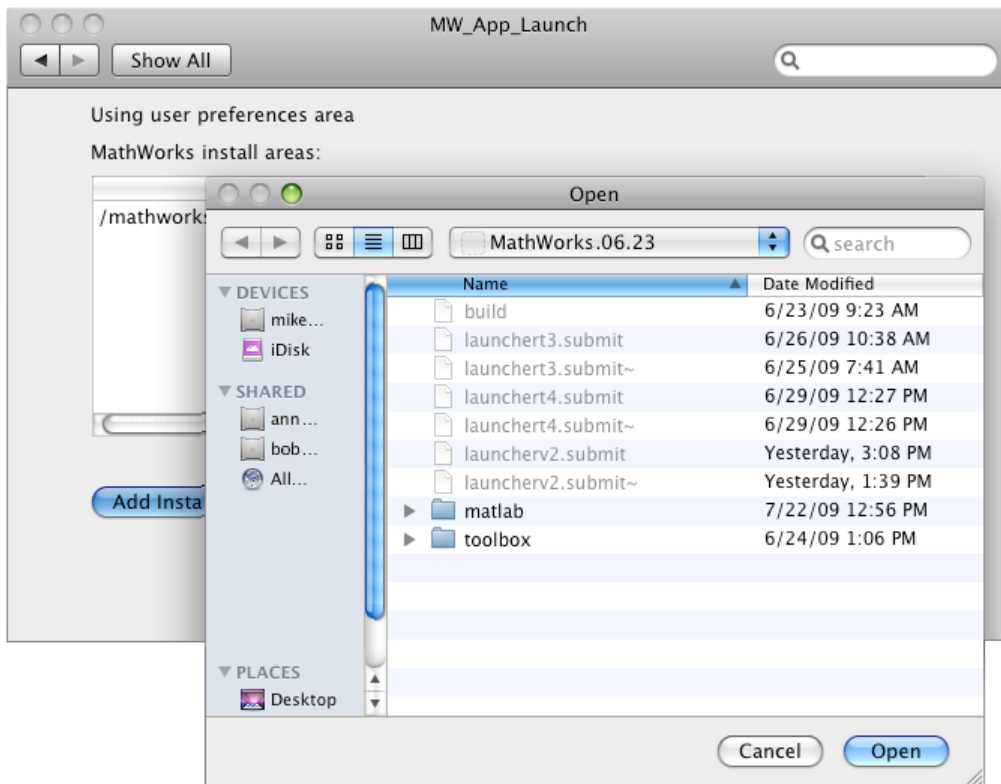
Modifying Your Installation Area

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 `applicationCompiler` to build the application:

- If you use `applicationCompiler`, the application bundle is placed in the `for_redistribution` folder of the packaged application.
- 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
applicationCompiler | compiler.build.standaloneApplication | mcc

More About
- "Create Standalone Application from MATLAB" on page 1-5

Files Generated After Packaging MATLAB Functions

When the packaging process is complete, three folders are generated in the target folder location: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`.

for_redistribution Folder

Distribute the `for_redistribution` folder to users who do not have MATLAB installed on their machines.

The folder contains the file `MyAppInstaller_web.exe` that installs the packaged application, MATLAB Runtime, and all the files that enable use of the application on the target platform with the target language in the target folder. It downloads MATLAB Runtime from the Internet if it is not included in the installer at the time of packaging.

for_redistribution_files_only Folder

Distribute the `for_redistribution_files_only` folder to users who do not have MATLAB installed on their machines. This folder contains specific files that enable use of the packaged application on the target platform with the target language.

Standalone Applications

File	Description
filename.exe	Standalone executable file.
readme.txt	Text file containing packaging information.
splash.png	When the executable starts, the file is read from the same folder where the executable is located, and the splash screen is displayed.

Excel Add-Ins

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`.
GettingStarted.html	HTML file with packaging and installation details.

for_testing Folder

Use the files in this folder to test your application. The folder contains all the intermediate and final artifacts such as binaries, `JAR` files, header files, and source files for a specific target. The final artifacts created during the packaging process are the same files as described in

"for_redistribution_files_only Folder" on page 3-11. For further information on how to test your packaged applications, see the following topics:

Target	Link
Standalone Application	"Install Standalone Application" on page 1-11
Excel Add-In	"Execute Functions and Create Macros"

The intermediate artifacts generated are a result of packaging of the MATLAB files. They are not significant to the user.

This folder also contains two text files. `mccExcludedFiles.txt` lists the files excluded from packaged application, and `requiredMCRProducts.txt` contains product IDs of products required by MATLAB Runtime to run the application.

See Also
`deploytool | mcc`

More About
- "Create Standalone Application from MATLAB" on page 1-5
- "Create Excel Add-In from MATLAB"

4

Customizing a Compiler Project

Customize an Application

You can customize an application in several ways: customize the installer, manage files in the project, or add a custom installer path using the **Application Compiler** app or the **Library Compiler** app.

Customize the Installer

Change Application Icon

To change the default icon, click the graphic to the left of the **Library name** or **Application name** field to preview the icon.

Click **Select icon**, and locate the graphic file to use as the application icon. Select the **Use mask** option to fill any blank spaces around the icon with white or the **Use border** option to add a border around the icon.

To return to the main window, click **Save and Use**.

Add Library or Application Information

You can provide further information about your application as follows:

- Library/Application Name: The name of the installed MATLAB artifacts. For example, if the name is `foo`, the installed executable is `foo.exe`, and the Windows start menu entry is **foo**. The folder created for the application is *InstallRoot*/`foo`.

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

- Version: The default value is 1.0.

- Author name: Name of the developer.

- Support email address: Email address to use for contact information.

- Company name: The full installation path for the installed MATLAB artifacts. For example, if the company name is `bar`, the full installation path would be *InstallRoot*/`bar`/*ApplicationName*.

- Summary: Brief summary describing the application.

- Description: Detailed explanation about the application.

All information is optional and, unless otherwise stated, is only displayed on the first page of the installer. On Windows systems, this information is also displayed in the Windows **Add/Remove Programs** control panel.

Change the Splash Screen

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

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

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

Change the Installation Path

This table lists the default path the installer uses when installing the packaged binaries onto a target system.
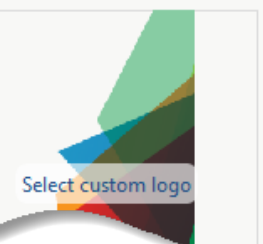
Windows	C:\Program Files*companyName**appName*
Mac OS X	/Applications/*companyName*/*appName*
Linux	/usr/*companyName*/*appName*

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

A text field specifying the path appended to the root folder is your installation folder. You can pick the root folder for the application installation folder. This table lists the optional custom root folders for each platform:

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

Change the Logo

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

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

Edit the Installation Notes

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

Determine Data Type of Command-Line Input (For Packaging Standalone Applications Only)

When an executable standalone application is run in the command prompt, the default input type is `char`. You can keep this default, or choose to interpret all inputs as numeric MATLAB doubles.

To pass inputs to the standalone application as MATLAB character vectors, select **Treat all inputs to the app as MATLAB character vectors**. In this case, you must include code to convert `char` to a numeric MATLAB type in the MATLAB function to be deployed as a standalone application.

To pass inputs to the standalone application as numeric MATLAB variables, select **Treat all inputs to the app as numeric MATLAB doubles**. option in the Application Compiler App. Thus, you do not need to include code to convert `char` to a numeric MATLAB type. Non numeric inputs to the application may result in an error.
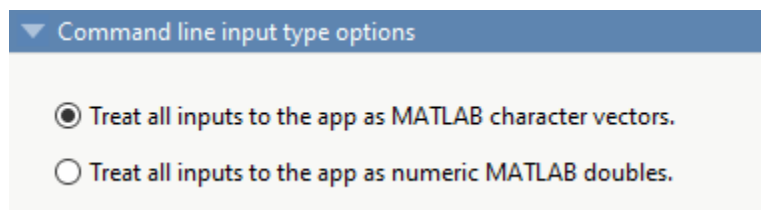
Manage Required Files in Compiler Project

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

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

To add files, click the plus button in the field, and select the file from the file explorer. To remove files, select the files, and press the **Delete** key.

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

Using mcc

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

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

Sample Driver File Creation

The following target types support sample driver file creation in MATLAB Compiler SDK:

- C++ shared library
- Java package
- .NET assembly
- Python® package
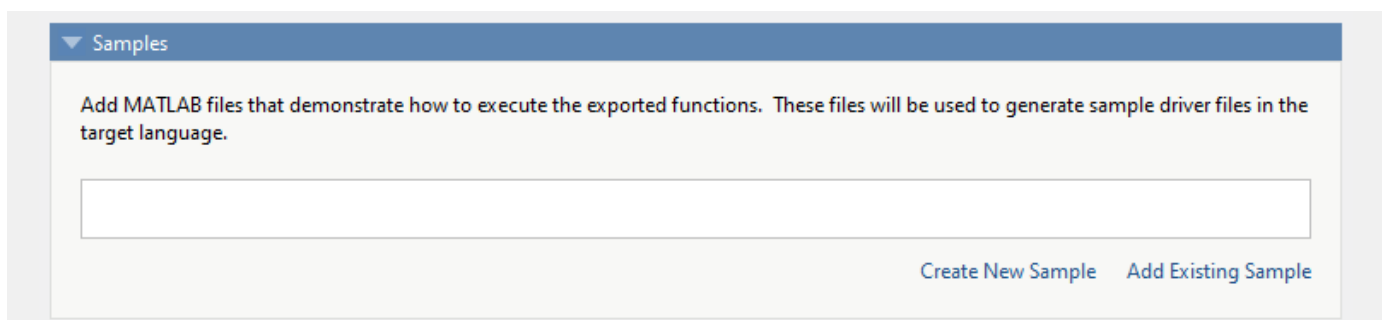
The sample driver file creation feature in **Library Compiler** uses MATLAB code to generate sample driver files in the target language. The sample driver files are used to implement the generated shared libraries into an application in the target language. In the app, click **Create New Sample** to automatically generate a new MATLAB script, or click **Add Existing Sample** to upload a MATLAB script that you have already written. After you package your functions, a sample driver file in the target language is generated from your MATLAB script and is saved in `for_redistribution_files_only\samples`. Sample driver files are also included in the installer in `for_redistribution`.

To automatically generate a new MATLAB file, click **Create New Sample**. This opens up a MATLAB file for you to edit. The sample file serves as a starting point, and you can edit it as necessary based on the behavior of your exported functions. The sample MATLAB files must follow these guidelines:

- The sample file code must use only exported functions.
- Each exported function must be in a separate sample file.
- Each call to the same exported function must be a separate sample file.
- The output of the exported function must be an n-dimensional numeric, char, logical, struct, or cell array.
- Data must be saved as a local variable and then passed to the exported function in the sample file code.
- Sample file code should not require user interaction.

Additional considerations specific to the target language are as follows:

- C++ mwArray API — `varargin` and `varargout` are not supported.
- .NET — Type-safe API is not supported.
- Python — Cell arrays and char arrays must be of size 1xN and struct arrays must be scalar. There are no restrictions on numeric or logical arrays, other than that they must be rectangular, as in MATLAB.

To upload a MATLAB file that you have already written, click **Add Existing Sample**. The MATLAB code should demonstrate how to execute the exported functions. The required MATLAB code can be only a few lines:

```
input1 = [1 4 7; 2 5 8; 3 6 9];
input2 = [1 4 7; 2 5 8; 3 6 9];
addoutput = addmatrix(input1,input2);
```

This code must also follow all the same guidelines outlined for the **Create New Sample** option.

You can also choose not to include a sample driver file at all during the packaging step. If you create your own driver code in the target language, you can later copy and paste it into the appropriate directory once the MATLAB functions are packaged.

Specify Files to Install with Application

The compiler packages files to install along with the ones it generates. By default, the installer includes a readme file with instructions on installing the MATLAB Runtime and configuring it.

These files are listed in the **Files installed for your end user** section of the app.

To add files to the list, click ➕, and select the file from the file explorer.

JAR files are added to the application class path as if you had called `javaaddpath`.

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

When installed on a target computer, the files listed in the **Files installed for your end user** are saved in the `application` folder.

Additional Runtime Settings

Type of Packaged Application	Description	Additional Runtime Settings Options
Standalone Applications	• **Do not display the Windows Command Shell (console) for execution** — If you select this option on a Windows platform, when you double-click the application from the file explorer, the application window opens without a command prompt. • **Create log file** — Generate a MATLAB log file for the application. The packaged application can't create a log file if installed in the `C:` folder on Windows because the application does not have write permission in that folder.	▼ Additional runtime settings ☑ Do not display the Windows Command Shell (console) for exec ☐ Create log file

Type of Packaged Application	Description	Additional Runtime Settings Options
Excel Add-Ins	• **Register the component for the current user (Recommended for non-admin users)** — This option enables registering the component for the current user account. It is provided for users without admin rights. • **Create log file** — Generate a MATLAB log file for the application. The packaged application can't create a log file if installed in the `C:` folder on Windows because the application does not have write permission in that folder.	▼ Additional runtime settings ☐ Register the component for the current user (Recommended fo ☐ Create log file

See Also
`applicationCompiler` | `libraryCompiler`

More About
- "Create Standalone Application from MATLAB" on page 1-5
- "Create Excel Add-In from MATLAB"
- "Generate a C++ mwArray API Shared Library and Build a C++ Application" (MATLAB Compiler SDK)
- "Generate a C++ MATLAB Data API Shared Library and Build a C++ Application" (MATLAB Compiler SDK)

Manage Support Packages

Using a Compiler App

Many MATLAB toolboxes use support packages to interact with hardware or to provide additional processing capabilities. If your MATLAB code uses a toolbox with an installed support package, the app displays a **Suggested Support Packages** section.
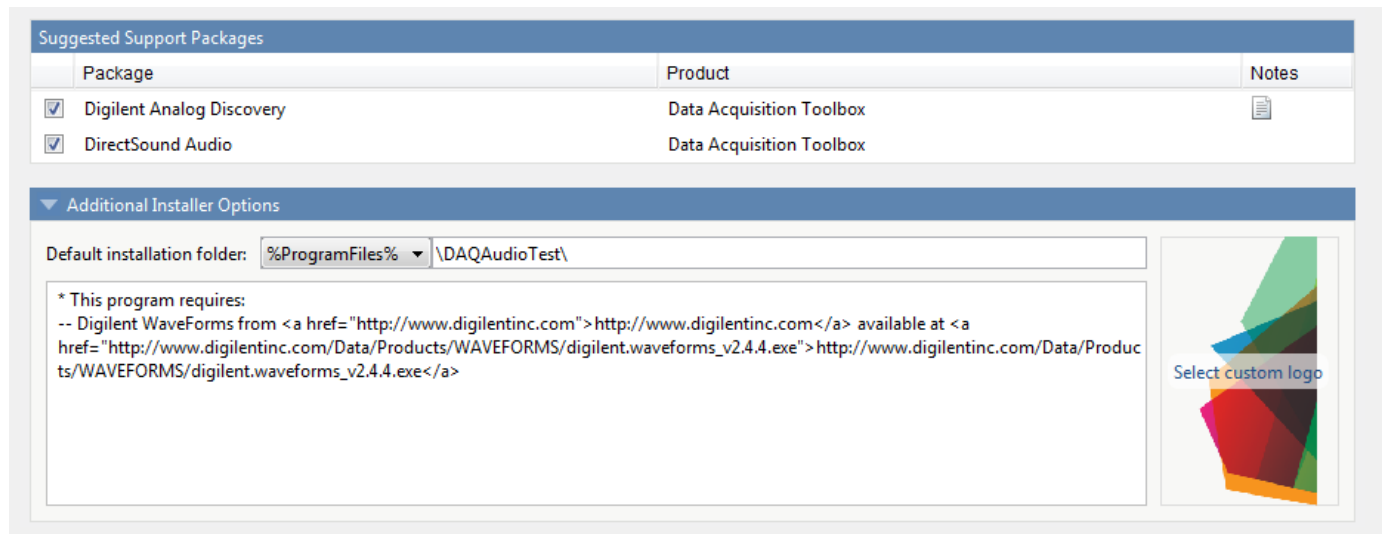
The list displays all installed support packages that your MATLAB code requires. The list is determined using these criteria:

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

Deselect support packages that are not required by your application.

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

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

Using the Command Line

Many MATLAB toolboxes use support packages to interact with hardware or to provide additional processing capabilities. If your MATLAB code uses a toolbox with an installed support package, use the `-a` flag with `mcc` command when packaging your MATLAB code to specify supporting files in the

support package folder. For example, if your function uses the `OS Generic Video Interface` support package, run the following command:

```
mcc -m -v test.m -a C:\MATLAB\SupportPackages\R2016b\toolbox\daq\supportpackages\daqaudio -a 'C:\
```

Some support packages require third-party drivers that the compiler cannot package. In this case, you are responsible for downloading and installing the required drivers.

MATLAB Code Deployment

How Does MATLAB Deploy Functions?

To deploy MATLAB functions, the compiler performs these tasks:

1 Analyzes files for dependencies using a dependency analysis function. Dependencies are files included in the generated package and originate from functions called by the file. Dependencies are affected by:

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

   For more information about dependency analysis, see "Dependency Analysis Using MATLAB Compiler" on page 5-3.

2 Validates MEX-files. In particular, `mexFunction` entry points are verified.

   For more details about MEX-file processing, see "MEX-Files, DLLs, or Shared Libraries" on page 5-5.

3 Creates a deployable archive from the input files and their dependencies.

   For more details about deployable archives, see "Deployable Archive" on page 5-6.

4 Generates target-specific wrapper code.

5 Generates target-specific binary package.

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

Dependency Analysis Using MATLAB Compiler

In this section...
"Function Dependency" on page 5-3
"Data File Dependency" on page 5-3
"Exclude Files From Package" on page 5-4

MATLAB Compiler uses a dependency analysis function to determine the list of necessary files to include in the generated package. Sometimes, this process generates a large list of files, particularly when MATLAB object classes exist in the compilation and the dependency analyzer cannot resolve overloaded methods at package time. 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 required for your application** in the compiler app or use the AdditionalFiles option in a compiler.build function.

Function Dependency

The dependency analyzer searches for executable content such as:

- MATLAB files
- P-files

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

- .fig files
- MEX-files

Data File Dependency

In addition to executable content listed above, 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.

To ensure that a specific file is included, specify the full path to the file as a character array in the function.

```
fileread('D:\Work\MATLAB\Project\myfile.ext')
```

The compiler app automatically adds these data files to the **Files required for your application to run** area.

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.

See Also
applicationCompiler | compiler.build.standaloneApplication | mcc

More About
- Application Compiler

MEX-Files, DLLs, or Shared Libraries

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

- Since the dependency analyzer cannot examine MEX-files, DLLs, or shared libraries to determine their dependencies, explicitly include all executable files these files require. To do so, use either the `mcc -a` option or the **Files required for your application to run** field in the compiler app.

- If you have any doubts that the dependency analyzer can find a MATLAB function called by a MEX-file, DLL, or shared library, then manually include that function. To do so, use either the `mcc -a` option or the **Files required for your application to run** field in the compiler app.

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

Deployable Archive

Each application or shared library you produce using the compiler has an embedded deployable archive. The archive contains all the MATLAB based content (MATLAB files, MEX-files, and so on). All MATLAB files in the deployable archive are encrypted using the Advanced Encryption Standard (AES) cryptosystem.

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

Information on Deployable Archive Embedding/Extraction and Component Cache

Product	Refer to
MATLAB Compiler SDK C/C++ integration	"MATLAB Runtime Component Cache and Deployable Archive Embedding" (MATLAB Compiler SDK)
MATLAB Compiler SDK .NET integration	"MATLAB Runtime Component Cache and Deployable Archive Embedding" (MATLAB Compiler SDK)
MATLAB Compiler SDK Java integration	"Deployable Archive Embedding and Extraction" (MATLAB Compiler SDK)
MATLAB Compiler Excel integration	"MATLAB Runtime Component Cache and Deployable Archive Embedding"

Generated Component (EXE, DLL, SO, etc)
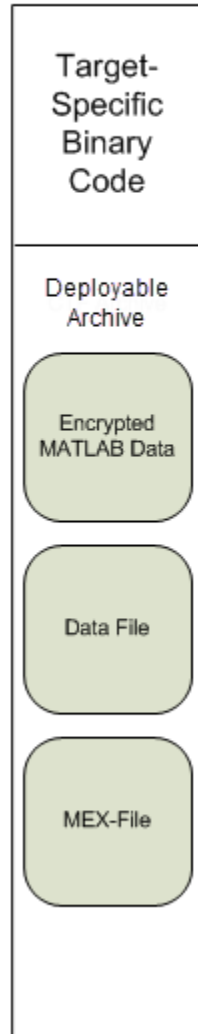
Additional Details

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.

Write Deployable MATLAB Code

In this section...

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, including the generated function.

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

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

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

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

For example, in a deployed application, the MATLAB and Java paths are fixed and cannot change. Therefore, any 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`. See "Use isdeployed Functions To Execute Deployment-Specific Code Paths" on page 5-10 for details.

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
```

Gradually Refactor Applications That Depend on Noncompilable Functions

Over time, refactor, streamline, and modularize MATLAB code containing non-compilable or non-deployable functions that use `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 undeployable code.

Do Not Create or Use Nonconstant Static State Variables

Avoid using the following:

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

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

When deploying applications, using persistent variables can cause problems because the 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`.

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.

See Also
`isdeployed` | `ismcc`

More About
- MATLAB Compiler support for MATLAB and toolboxes

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 Application Compiler or Library Compiler apps, 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 C Functions in Shared Libraries"

MATLAB Data Files in Compiled Applications

In this section...
"Explicitly Including MATLAB Data files Using the %#function Pragma" on page 5-13
"Load and Save Functions" on page 5-13

Explicitly Including MATLAB Data files Using the %#function Pragma

The compiler excludes MATLAB data files (MAT-files) from dependency analysis by default. See "Dependency Analysis Using MATLAB Compiler" on page 5-3.

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

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

Load and Save Functions

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

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

For more information about deployable archives, see "Deployable Archive" on page 5-6.

See the ctfroot reference page for more information about ctfroot.

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

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

The following example specifies three MATLAB data files:

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

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

2 Compile ex_loadsave.m with the following mcc command:

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

ex_loadsave.m

```
function ex_loadsave
% This example shows how to work with the
% "load/save" functions on data files in
% deployed mode. There are three source data files
% in this example.
%    user_data.mat
%    userdata\extra_data.mat
%    ..\externdata\extern_data.mat
%
% Compile this example with the mcc command:
%     mcc -m ex_loadsave.m -a 'user_data.mat' -a
%     '.\userdata\extra_data.mat'
%         -a '..\externdata\extern_data.mat'
% All the folders under the current main MATLAB file directory will
%    be included as
% relative path to ctfroot; All other folders will have the
%    folder
% structure included in the deployable archive file from root of the
%     disk drive.
%
% If a data file is outside of the main MATLAB file path,
%     the absolute path will be
% included in deployable archive and extracted under ctfroot. For example:
%   Data file
%     "c:\$matlabroot\examples\externdata\extern_data.mat"
%     will be added into deployable archive and extracted to
%  "$ctfroot\$matlabroot\examples\externdata\extern_data.mat".
%
% All mat/data files are unchanged after mcc runs. There is
% no encryption on these user included data files. They are
% included in the deployable archive.
%
% The target data file is:
%    .\output\saved_data.mat
%   When writing the file to local disk, do not save any files
%     under ctfroot since it may be refreshed and deleted
%   when the application isnext started.

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

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

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

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

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

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

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 the MATLAB Runtime" on page 7-2*
- *"Install and Configure MATLAB Runtime" on page 7-3*
- *"Run Applications Using a Network Installation of MATLAB Runtime" on page 7-9*
- *"MATLAB Runtime on Big Data Platforms" on page 7-10*
- *"Install Deployed Application" on page 7-11*

About the MATLAB Runtime

In this section...
"How is the MATLAB Runtime Different from MATLAB?" on page 7-2
"Performance Considerations and the MATLAB Runtime" on page 7-2

The MATLAB Runtime is a standalone 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 SDK require access to an appropriate version of the MATLAB Runtime to run.

End-users of compiled artifacts without access to MATLAB must install the MATLAB Runtime on their computers or know the location of a network-installed MATLAB Runtime. The installers generated by the compiler apps may include the MATLAB Runtime installer. If you compiled your artifact using `mcc`, you should direct your end-users to download the MATLAB Runtime installer from the website `https://www.mathworks.com/products/compiler/mcr`.

See "Install and Configure MATLAB Runtime" on page 7-3 for more information.

How is the MATLAB Runtime Different from MATLAB?

The MATLAB Runtime differs from MATLAB in several important ways:

- In the MATLAB Runtime, MATLAB files are encrypted and immutable.

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

- The MATLAB Runtime is version-specific. You must run your applications with the version of the MATLAB Runtime associated with the version of MATLAB Compiler SDK with which it was created. For example, if you compiled an application using version 6.3 (R2016b) of MATLAB Compiler, users who do not have MATLAB installed must have version 9.1 of the MATLAB Runtime installed. Use `mcrversion` to return the version number of the 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.

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

Calls into the MATLAB Runtime are serialized so calls into the MATLAB Runtime are threadsafe. This can impact performance.

Install and Configure MATLAB Runtime

Supported Platforms: Windows, Linux, macOS

MATLAB Runtime contains the libraries needed to run MATLAB applications on a target system without a licensed copy of MATLAB.

Download MATLAB Runtime Installer

Download MATLAB Runtime 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.

Install MATLAB Runtime Interactively

To install MATLAB Runtime:

1. Extract the archive containing the MATLAB Runtime installer.

Platform	Steps
Windows	Unzip the MATLAB Runtime installer. Right-click the ZIP file `MATLAB_Runtime_R2021a_win64.zip` and select **Extract All**.
Linux	Unzip the MATLAB Runtime installer at the terminal using the `unzip` command. For example, if you are unzipping the R2021a MATLAB Runtime installer, at the terminal, type: `unzip MATLAB_Runtime_R2021a_glnxa64.zip`
macOS	Unzip the MATLAB Runtime installer at the terminal using the `unzip` command. For example, if you are unzipping the R2021a MATLAB Runtime installer, at the terminal, type: `unzip MATLAB_Runtime_R2021a_maci64.zip`

Note The release part of the installer file name (*_R2021a_*) changes from one release to the next.

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` **Note** The `-H` option sets the `HOME` environment variable to the home directory of the root user and should be used for graphical applications such as installers.
macOS	At the terminal, type: `./install` **Note** You may need to enter an administrator user name and password after you run `./install`.

Note If you are running the MATLAB Runtime installer on a shared folder, be aware that other users of the share may need to alter their system configuration.

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 in which you want to install MATLAB Runtime.

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

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 information on setting environment variables, see "Set MATLAB Runtime Path for Run-Time Deployment" on page 15-2.

7   Click **Finish** to exit the installer.

The default MATLAB Runtime installation directory for **R**2021**a** is specified in the following table:

Operating System	MATLAB Runtime Installation Directory
Windows	`C:\Program Files\MATLAB\MATLAB Runtime\v910`
Linux	`/usr/local/MATLAB/MATLAB_Runtime/v910`
macOS	`/Applications/MATLAB/MATLAB_Runtime/v910`

Install MATLAB Runtime Noninteractively

To install MATLAB Runtime without having to interact with the installer dialog boxes, use one of these noninteractive modes:

- Silent — The installer runs as a background task and does not display any dialog boxes.
- Automated — The installer displays the dialog boxes but does not wait for user interaction.

When run in silent or automated mode, the MATLAB Runtime installer uses default values for installation options. You can override these values by using MATLAB Runtime installer command-line options or an installer control file.

Note When running in silent or automated mode, the installer overwrites the installation location.

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 `-mode silent` and `-agreeToLicense yes` options on the command line.

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

Platform	Command
Windows	`setup -mode silent -agreeToLicense yes`
Linux	`./install -mode silent -agreeToLicense yes`
macOS	`./install -mode silent -agreeToLicense yes`

Note If you do not include the `-agreeToLicense yes` option, the installer does not install MATLAB Runtime.

4 View a log of the installation.

On Windows systems, the MATLAB Runtime 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.

5 On Linux and macOS systems, the MATLAB Runtime installer displays the log information at the command prompt and also saves it to a file if you use the `-outputFile` option.

Customize a Noninteractive Installation

When run in one of the noninteractive modes, the installer uses the default values unless you specify otherwise. Like the MATLAB installer, the MATLAB Runtime installer accepts a number of command-line options that modify the default installation properties.

Option	Description
`-destinationFolder`	Specifies where MATLAB Runtime is installed.
`-outputFile`	Specifies where the installation log file is written.
`-tmpdir`	Specifies where temporary files are stored during installation. **Caution** The installer deletes everything inside the specified folder.
`-automatedModeTimeout`	Specifies how long, in milliseconds, that the dialog boxes are displayed when run in automatic mode.
`-inputFile`	Specifies an installer control file that contains your command-line options and values. Omit the dashes and put each option and value on a separate line.

Note The MATLAB Runtime installer archive includes an example installer control file called `installer_input.txt`. This file contains all of the options available for a full MATLAB installation. The options listed in this section are valid for the MATLAB Runtime installer.

Install MATLAB Runtime without Administrator Rights

To install MATLAB Runtime as a user without administrator rights on Windows:

1   Use the MATLAB Runtime installer to install it 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 Run-Time 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.

Install 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, however, 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 Run-Time Deployment" on page 15-2.

Uninstall MATLAB Runtime

The method you use to uninstall MATLAB Runtime from your computer varies depending on your platform.

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>*\uninstall\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 Run-Time Deployment" on page 15-2

Run Applications Using a Network Installation of MATLAB Runtime

Local clients on a network can access MATLAB Runtime on a network drive.

On Linux systems, distributing to a network file system is the same as distributing to a local file system. After installing MATLAB Runtime, set the LD_LIBRARY_PATH environment variable or use shell scripts that point to the MATLAB Runtime installation. For information on setting the library path, see "Set MATLAB Runtime Path for Run-Time Deployment" on page 15-2.

On Windows systems, complete the following steps to run applications using a network install of MATLAB Runtime:

1 Run the mcrinstaller function to obtain name and location of the MATLAB Runtime installer.

2 Copy the entire MATLAB Runtime folder onto a network drive.

3 Copy the compiled application into a separate folder on the network drive and add the directory *<MATLAB_RUNTIME_INSTALL_DIR>**<VERSION>*\runtime*<ARCH>* to the path on all client machines. All network clients can then execute the application.

4 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
.NET assemblies to create COM objects	mwcomutil.dll

To register the DLLs, at the system command prompt enter:

mwregsvr *<fully_qualified_pathname\dllname*.dll>

These DLLs are located in *<<MATLAB_RUNTIME_INSTALL_DIR>>**<VERSION>*\bin\win64.

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

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 by Cloudera Manager.

Download URL

```
https://www.mathworks.com/supportfiles/downloads/R2021a/deployment_files/
R2021a/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

MATLAB Runtime is available for distribution as an Apache Ambari stack.

You can distribute MATLAB Runtime across a Hadoop® cluster using Apache Ambari.

Download URL

```
https://www.mathworks.com/supportfiles/downloads/R2021a/deployment_files/
R2021a/ambari/matlab-runtime-2021a-service.tgz
```

```
https://www.mathworks.com/supportfiles/downloads/R2021a/deployment_files/
R2021a/ambari/matlab-runtime-2021a-service.sha1
```

For more information, see the Apache Ambari documentation.

Azure HDInsight

MATLAB Runtime is available for distribution as an Azure HDInsight script action. You can distribute MATLAB Runtime across an Azure HDInsight cluster using script action.

Download URL

```
https://www.mathworks.com/supportfiles/downloads/R2021a/deployment_files/
R2021a/hdinsight
```

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 `magicsquare` interactively using `MyAppInstaller`.

1 Start the installer.

Platform	Steps
Windows	Double-click the file `MyAppInstaller.exe`.
Linux	In the terminal, type: `sudo -H ./MyAppInstaller` **Note** The `-H` option sets the `HOME` environment variable to the home directory of the root user and should be used for graphical applications.
macOS	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 Run-Time 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\magicsquare`
macOS	`/Applications/magicsquare`
Linux	`/usr/magicsquare`

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.

To install an application without having to interact with the installer dialog boxes, use one of the following noninteractive modes:

- Silent — The installer runs as a background task and does not display any dialog boxes.
- Automated — The installer displays the dialog boxes but does not wait for user interaction.

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 in automated or silent mode, 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 Run-Time Deployment" on page 15-2.

Note When running in silent or automated mode, the installer overwrites the installation location.

Run Installer in Silent Mode

To install the application in silent mode:

1. Run the installer on the command line and specify the options `-mode silent` and `-agreeToLicense yes`.

Platform	Command
Windows	`MyAppInstaller.exe -mode silent -agreeToLicense yes`
Linux	`./MyAppInstaller -mode silent -agreeToLicense yes`
macOS	`./MyAppInstaller -mode silent -agreeToLicense yes`

   Note If you do not include the `-agreeToLicense yes` option, the installer will not install the application.

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.

   On Linux and macOS systems, the installer displays the log information at the command prompt and will also save it to a file if you use the `-outputFile` option.

Customize Noninteractive Installation

When run in one of the noninteractive modes, 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. Separate each option and its value with a space.

Option	Description	Comment
`-destinationFolder`	Specifies where the application and MATLAB Runtime is installed.	In the destination folder, MATLAB Runtime is installed in a folder named after the version number, for example, v910.
`-outputFile`	Specifies where the installation log file is written.	
`-automatedModeTimeout`	Specifies how long, in milliseconds, that the dialog boxes are displayed when run in automated mode.	The default value is 1000 milliseconds.
`-desktopShortcut true|false`	Specifies whether to create a desktop shortcut icon for the installed application.	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 is only used on Windows.
`-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 on a separate line.

See Also

More About

- "Create Standalone Application from MATLAB" on page 1-5
- "Install and Configure MATLAB Runtime" on page 7-3
- "Set MATLAB Runtime Path for Run-Time 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 8-4
- "Display MATLAB Runtime Initialization Messages" on page 8-6

MATLAB Runtime Startup Options

Set MATLAB Runtime Options

For a standalone executable, set MATLAB Runtime options by specifying the -R switch and arguments. You can set options from either of the following:

- The **Additional Runtime Settings** area of the compiler apps.
- The mcc command.

Note Not all options are available for all compilation targets.

Use a Compiler App

In the **Additional Runtime Settings** area of the compiler apps, you can set the following options.

MATLAB Runtime Startup Option	Description	Setting
-nojvm	Disable the Java Virtual Machine (JVM™), which is enabled by default. This can help improve the MATLAB Runtime performance.	Select the **No JVM** check box.
-nodisplay	On Linux, open the MATLAB Runtime without display functionality.	In the **Settings** box, enter -R -nodisplay.
-logfile	Write information about the MATLAB Runtime startup to a logfile.	Select the **Create log file** check box. Enter the path to the log file, including the log file name, in the **Log File** box.
-startmsg	Specify message to be displayed when the MATLAB Runtime begins initialization.	In the **Settings** box, enter -R 'startmsg, *message text*'.
-completemsg	Specify message to be displayed when the MATLAB Runtime completes initialization.	In the **Settings** box, enter -R 'completemsg, *message text*'.

Set MATLAB Runtime Startup Options Using the mcc Command Line

When you use the command line, specify the -R switch to invoke the MATLAB Runtime startup options you want to use.

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

Set -nojvm

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

Set -nodisplay (Linux Only)

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

Set -logfile

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

Set -nojvm, -nodisplay, and -logfile with One Command

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

Using MATLAB Runtime User Data Interface

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

- You need to supply run-time profile information to a client running an application created with the Parallel Computing Toolbox. You supply and change profile information on a per-execution basis. For example, two instances of the same application may run simultaneously with different profiles. For more information, see "Use Parallel Computing Toolbox in Deployed Applications" on page 3-5.
- You want to set up a global workspace, a global variable, or variables that MATLAB and your client can access.
- You want to store the state of any variable or group of variables.

The API consists of:

- Two MATLAB functions callable from within deployed application MATLAB code
- Four external C functions callable from within deployed application wrapper code

MATLAB Functions

Use the MATLAB functions `getmcruserdata` and `setmcruserdata` from deployed MATLAB applications. They are loaded by default only in applications created with the MATLAB Compiler or MATLAB Compiler SDK products.

You can include `setmcruserdata` and `getmcruserdata` in your packaged application using `mcc` as follows:

```
mcc -g -W cpplib:<lib> -T link:lib ... setmcruserdata.m getmcruserdata.m
```

You can also use the `%#` function in your MATLAB file to include `setmcruserdata` and `getmcruserdata`. Doing so ensures inclusion of these functions in the packaged application when you use `deploytool`.

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 about the `isdeployed` function, see the `isdeployed` reference page.

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

`getmcruserdata` | `setmcruserdata`

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

Some examples of different ways to invoke this option follow:

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

Best Practices

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

- When calling `mcc` in the MATLAB command window, place the comma inside the single quote.

  `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.

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 "Install and Configure MATLAB Runtime" on page 7-3.

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 Run-Time 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

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 Run-Time 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

`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.

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 the MATLAB Runtime?

All shared libraries required for your standalone executable or shared library are contained in the MATLAB Runtime. Installing the 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 `mclmcrrt7x.dll` or `mclmcrrt7x.so` are generally caused by incorrect installation of the MATLAB Runtime. It is also possible that the MATLAB Runtime is installed correctly, but that the `PATH`, `LD_LIBRARY_PATH`, or `DYLD_LIBRARY_PATH` variables is set incorrectly. For information on installing the MATLAB Runtime on a deployment machine, see "Install and Configure MATLAB Runtime" on page 7-3.

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

Does your system's graphics card support the graphics application?

In situations where the existing hardware graphics card does not support the graphics application, you should use software OpenGL. OpenGL libraries are visible for an application by appending `matlab/sys/opengl/lib/arch` to the `LD_LIBRARY_PATH`. For example:

`setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:matlab/sys/opengl/lib/`*arch*

For issues with MATLAB graphics in Linux, set the environment variable `LD_LIBRARY_PATH` to:

`setenv LD_LIBRARY_PATH $MATLAB/sys/opengl/lib/glnxa64:$LD_LIBRARY_PATH`

Is OpenGL properly installed on your system?

When searching for OpenGL libraries, the MATLAB Runtime first looks on the system library path. If OpenGL is not found there, it will use the `LD_LIBRARY_PATH` environment variable to locate the libraries. If you are getting failures due to the OpenGL libraries not being found, you can append the location of the OpenGL libraries to the `LD_LIBRARY_PATH` environment variable. For example:

`setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:matlab/sys/opengl/lib/`*arch*

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 the MATLAB Runtime on their machines. The MATLAB Runtime includes a set of shared libraries that provides support for all features of MATLAB. If your application fails during end-user deployment, the following questions in the column to the right may help you isolate the problem.

Note There are a number of reasons why your application might not deploy to end users, after running successfully in a test environment. For a detailed list of guidelines for writing MATLAB code that can be consumed by end users, see "Write Deployable MATLAB Code" on page 5-9

Is MATLAB Runtime installed?

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. See "Install and Configure MATLAB Runtime" on page 7-3 for complete information.

If running on UNIX or Mac, did you update the dynamic library path after installing the MATLAB Runtime?

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

Do you receive an error message about a missing DLL?

Error messages indicating missing DLLs such as `mclmcrrt7x.dll` or `mclmcrrt7x.so` are generally caused by incorrect installation of MATLAB Runtime. It is also possible that MATLAB Runtime is installed correctly, but that the `PATH`, `LD_LIBRARY_PATH`, or `DYLD_LIBRARY_PATH` variables are set incorrectly. For information on installing the MATLAB Runtime on a deployment machine, see "Install and Configure MATLAB Runtime" on page 7-3. For information on setting the library path, see "Set MATLAB Runtime Path for Run-Time Deployment" on page 15-2.

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

Do you have write access to the directory the application is installed in?

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

Are you executing a newer version of your application?

When deploying a newer version of an executable, the executable needs to be redeployed, since it also contains the embedded deployable archive. The deployable archive is keyed to a specific compilation session. Every time an application is recompiled, a new, matched deployable archive is

created. Delete the existing *application-name*_mcr 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
- "Functions Not Supported for Compilation by MATLAB Compiler and MATLAB Compiler SDK " on page 13-7

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, go to the MathWorks website.

Compiled applications can run only on the same platform on which they were developed, with the following exceptions:

- Web apps, which can be deployed to MATLAB Web App Server™ running on any compatible platform.
- C++ libraries compiled using the MATLAB Data API that do not contain platform-specific files.
- .NET Assemblies compiled using .NET Core that do not contain platform-specific files.
- Java packages that do not contain platform-specific files.
- Python packages that do not contain platform-specific files.

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 13-7.

Fixing Callback Problems: Missing Functions

When MATLAB Compiler creates a standalone application, it packages the MATLAB files that you specify on the command line. In addition, it includes any other MATLAB files that your packaged MATLAB files call. MATLAB Compiler uses a dependency analysis, which determines all the functions on which the supplied MATLAB files, MEX-files, and P-files depend.

Note If the MATLAB file associated with a p-file is unavailable, the dependency analysis 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 analyzer 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
- Using the `-a` option

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.

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

Finding Missing Functions in a MATLAB File

To find functions in your application that 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 the UNIX System

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

To suppress the `libjvm.so` warning, set the dynamic library path properly for your platform. See "Set MATLAB Runtime Path for Run-Time Deployment" on page 15-2.

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

Cannot Use Graphics with the -nojvm Option

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

Cannot Create the 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
```

Packaging a Function with which 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 the `pwd` function to explicitly point to the file in the current folder, as follows:

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

- Rather than using the general `open` function, use `load` or other specialized functions for your particular file type, as `load` explicitly checks for the file in the current folder. For example:

  ```
  load myFile.mat
  ```

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

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

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

For instance, if you are working with the following array:

```
[1 2 3 4]
```

you cannot use `SetData` to increase the size of the array to a length of five elements.

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 Application	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.
Library Compiler	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	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.

See Also

More About

- "Functions Not Supported for Compilation by MATLAB Compiler and MATLAB Compiler SDK" on page 13-7

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.

List of Unsupported Functions and Programs

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

new_system

open

open_system

pack

pcode

plotbrowser

plotedit

plottools

profile

profsave

propedit

propertyeditor

publish

rehash

restoredefaultpath

run

segment

set_param

sldebug

type

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.

Prerequisites

1   Verify that you have Docker installed on your Linux machine by typing `docker` in a console. 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   Verify that the 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 a message stating that the MATLAB Runtime installer exists and specifies 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 need to 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 computer that is not connected to the Internet. 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 with the file name `mymagic.m`.

```
function y = mymagic(x)

y = magic(x);
```

Test the function at the MATLAB command prompt.

```
out = mymagic(5)

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
```

Create Standalone Application

Make 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]
```

Once the build is complete, the function creates a folder named `mymagicstandaloneApplication` in your current directory to store the standalone application. The `Results` object `res` returned at the MATLAB command prompt contains information about the build.

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'
     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 in 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.

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 console 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/r2020b/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 console, 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.build.standaloneApplication | compiler.package.DockerOptions | compiler.package.docker | compiler.runtime.download

Reference Information

Set MATLAB Runtime Path for Run-Time Deployment

In this section...
"Path for Java Applications on All Platforms" on page 15-2
"Windows" on page 15-2
"Linux" on page 15-3
"macOS" on page 15-4

After you install MATLAB Runtime, add the run-time directories to the system library path according to the instructions for your operating system and shell environment.

Note

- Ensure that the MATLAB Runtime directories are not already present in the path before adding them.
- Save the value of your current library path as a backup before modifying it.
- 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.

Path for Java Applications on All Platforms

When you run Java applications that contain compiled MATLAB code, you must set the path so that the system can find MATLAB Runtime.

When you deploy a Java application to end users, they must set the class path on the target machine. For their system to locate the `.jar` files containing the MATLAB libraries, users must specify a `classpath` either in the `javac` command or in their system environment variables. For more information, see "Configure Your Java Environment" (MATLAB Compiler SDK).

Path Modifications Required for Accessibility

In order to use some screen-readers or assistive technologies, such as JAWS®, you must add the following DLLs to your Windows path:

```
matlabroot\sys\java\jre\arch\jre\bin\JavaAccessBridge.dll
matlabroot\sys\java\jre\arch\jre\bin\WindowsAccessBridge.dll
```

You may not be able to use such technologies without doing so.

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.

Graphical Approach

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 directory *<MATLAB_RUNTIME_INSTALL_DIR>*`\runtime\`*<arch>*. For example, if you are using MATLAB Runtime R2021a located in the default installation directory on 64-bit Windows, add `C:\Program Files\MATLAB\MATLAB Runtime\v910\runtime \win64`.

4   Click **OK** to apply the change.

Command-Line Approach Using PowerShell

1   Execute the following command using Windows PowerShell with elevated privileges to save the current system path as a variable named `mypath` and display it in the console.

```
($mypath = (Get-Item "HKLM:\System\CurrentControlSet\Control\Session Manager\Environment").Ge
```

2   Append your MATLAB Runtime directory to the `mypath` variable. If there is a semicolon at the end of your current path or the path is empty, then delete the semicolon from the following command.

```
$mypath = "${mypath};<MATLAB_RUNTIME_INSTALL_DIR>\runtime\<arch>"
```

For example, if you are using MATLAB Runtime R2021a located in the default installation directory on 64-bit Windows, use the following command:

```
$mypath = "${mypath};C:\Program Files\MATLAB\MATLAB Runtime\v910\runtime\win64"
```

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

```
echo %mypath%
```

4   Set the system path to the value of `mypath`.

```
Set-ItemProperty -Path 'Registry::HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session
```

5   Sign out of Windows to apply the change.

Linux

In the terminal, display the `SHELL` variable to determine your current shell environment.

```
echo $SHELL
```

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

Bash Shell

1   Save your current library path as a variable named `mypath` and display it in the console.

```
mypath=$LD_LIBRARY_PATH && echo $mypath
```

2   Append the MATLAB Runtime directories to the `mypath` variable. The following command must be entered as a single line.

```
mypath="${mypath:+${mypath}:}
<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"
```

Note If you are using OpenGL®, replace /extern/bin/glnxa64 with /sys/opengl/lib/ glnxa64.

For example, if you are using MATLAB Runtime R2021a located in the default installation directory, use the following command entered on a single line:

```
mypath="${mypath:+${mypath}:}/usr/local/MATLAB/MATLAB_Runtime/v910/runtime/glnxa64:
/usr/local/MATLAB/MATLAB_Runtime/v910/bin/glnxa64:/usr/local/MATLAB/MATLAB_Runtime/v910/exter
```

3 Display the value of mypath to ensure the path is correct.

```
echo $mypath
```

4 Set the LD_LIBRARY_PATH variable for the current session.

```
export LD_LIBRARY_PATH=$mypath
```

5 To set the variable permanently, append the export command to a file in your home directory named .bashrc.

```
echo "export LD_LIBRARY_PATH=$mypath" >> ~/.bashrc
```

6 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.

7 To apply changes to the current session, type source ~/.bashrc.

macOS

In the terminal, display the SHELL variable to determine your current shell environment.

```
echo $SHELL
```

Zsh Shell (macOS Catalina 10.15 or Later)

1 Save your current library path as a variable named mypath and display it in the console.

```
mypath=$DYLD_LIBRARY_PATH && echo $mypath
```

2 Append the MATLAB Runtime directories to the mypath variable. The following command must be entered as a single line.

```
mypath="${mypath:+${mypath}:}
<MATLAB_RUNTIME_INSTALL_DIR>/runtime/maci64:
<MATLAB_RUNTIME_INSTALL_DIR>/bin/maci64:
<MATLAB_RUNTIME_INSTALL_DIR>/sys/os/maci64"
```

For example, if you are using MATLAB Runtime R2021a located in the default installation directory, use the following command entered on a single line:

```
mypath="${mypath:+${mypath}:}/Applications/MATLAB/MATLAB_Runtime/v910/runtime/maci64:
/Applications/MATLAB/MATLAB_Runtime/v910/bin/maci64:/Applications/MATLAB/MATLAB_Runtime/v910/
```

3 Display the value of mypath to ensure the path is correct.

```
echo $mypath
```

4   Set the DYLD_LIBRARY_PATH variable for the current session.

```
setenv DYLD_LIBRARY_PATH $mypath
```

5   To set the variable permanently, append the `setenv` command to a file in your home directory named `.zshrc`.

```
echo "setenv DYLD_LIBRARY_PATH $mypath" >> ~/.zshrc
```

6   To apply changes to the current session, type `source ~/.zshrc`.

Bash Shell (macOS Mojave 10.14 or Earlier)

1   Save your current library path as a variable named `mypath` and display it in the console.

```
mypath=$DYLD_LIBRARY_PATH && echo $mypath
```

2   Append the MATLAB Runtime directories to the `mypath` variable. The following command must be entered as a single line.

```
mypath="${mypath:+${mypath}:}
<MATLAB_RUNTIME_INSTALL_DIR>/runtime/maci64:
<MATLAB_RUNTIME_INSTALL_DIR>/bin/maci64:
<MATLAB_RUNTIME_INSTALL_DIR>/sys/os/maci64"
```

For example, if you are using MATLAB Runtime R2021a located in the default installation directory, use the following command entered as a single line:

```
mypath="${mypath:+${mypath}:}/Applications/MATLAB/MATLAB_Runtime/v910/runtime/maci64
:/Applications/MATLAB/MATLAB_Runtime/v910/bin/maci64:/Applications/MATLAB/MATLAB_Runtime/v910
```

3   Display the value of $mypath to ensure the path is correct.

```
echo $mypath
```

4   Set the DYLD_LIBRARY_PATH variable for the current session.

```
export DYLD_LIBRARY_PATH=$mypath
```

5   To set the variable permanently, append the `export` command to a file in your home directory named `.profile`.

```
echo "export DYLD_LIBRARY_PATH=$mypath" >> ~/.profile
```

6   To apply changes to the current session, type `source ~/.profile`.

See Also

More About

- "Install and Configure MATLAB Runtime" on page 7-3
- "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 functions or solution is ultimately integrated. Typically, the end goal for the deployment customer is integration of a deployed MATLAB function into a larger enterprise environment application. The deployment products prepare the MATLAB function for integration by wrapping MATLAB code with enterprise-compatible source code, such as C, C++, C# (.NET), F#, and Java code.

Assembly — An executable bundle of code, especially in .NET.

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. See "Additional Details" on page 5-7.

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` (MATLAB Production Server).

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 `for_redistribution` subfolder. In addition to the installer, the compiler apps generate a number of lose artifacts that can be used for testing or building a custom installer.

PID File — See *Process Identification File (PID File)*.

Pool — A pool of threads, in the context of server management using MATLAB Production Server software. Servers created with the software do not allocate a unique thread to each client connection. Rather, when data is available on a connection, the required processing is scheduled on a pool, or group, of available threads. The server configuration file option `--num-threads` sets the size of that pool (the number of available request-processing threads) in the master server process.

Process Identification File (PID File) — A file that documents informational and error messages relating to a running server instance of MATLAB Production Server software.

Program — A bundle of code that is executed to achieve a purpose. Programs usually are written to automate repetitive operations through computer processing. Enterprise system applications usually consist of hundreds or even thousands of smaller programs.

Properties — For this definition in the context of .NET, see *Fields and Properties*.

Proxy — A software design pattern typically using a class, which functions as an interface to something else. For example, `MWArray` is a proxy for programmers who need to access the underlying type `mxArray`.

S

Server Instance — See MATLAB Production Server Server Instance.

Shared Library — Groups of files that reside in one space on disk or memory for fast loading into Windows applications. Dynamic-link libraries (DLLs) are Microsoft's implementation of the shared library concept 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.

Functions

%#exclude

Ignore a file or function dependency during dependency analysis while executing the mcc command

Syntax

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

Description

%#exclude fileOrFunction1 [fileOrFunction2 ... fileOrFunctionN] pragma informs the mcc command that the specified file(s) or function(s) need to be excluded from dependency analysis during compilation.

Examples

Using %#exclude Within a MATLAB Function

Create a MATLAB function named testExclusion that includes a %#exclude pragma to determine which files are included and which ones are excluded while executing the mcc command with various options.

```
function testExclusion()

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

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

- 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

The -a option in the mcc command is used to add files. The %#function pragma is used to inform the mcc command that the specified function(s) should be included in the compilation.

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

See Also

`mcc`

Introduced in R2020a

%#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 "MATLAB Data Files in Compiled Applications".

Introduced before R2006a

applicationCompiler

Build and package functions into standalone applications

Syntax

```
applicationCompiler
applicationCompiler project_name
```

Description

`applicationCompiler` opens the MATLAB standalone compiler for the creation of a new compiler project. For more information on the Application Compiler app, see Application Compiler.

`applicationCompiler project_name` opens the MATLAB standalone compiler app with the project preloaded.

Examples

Create a New Standalone Application Project

Open the application compiler to create a new project.

```
applicationCompiler
```

Input Arguments

project_name — name of the project to be compiled
character array or string

Specify the name of a previously saved MATLAB Compiler project. The project must be on the current path.

Compatibility Considerations

-build and -package options will be removed
Not recommended starting in R2020a

The `-build` and `-package` options will be removed. To build applications, use the `compiler.build.standaloneApplication` function or the `mcc` command, and to package and create an installer, use the `compiler.package.installer` function.

See Also
`compiler.package.installer` | `deploytool` | `mcc`

Introduced in R2013b

compiler.build.Results

Compiler build results object

Description

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

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

For information on results from compiling C/C++ shared libraries, .NET assemblies, COM components, Java packages, Python packages, or MATLAB Production Server deployable archives, see `compiler.build.Results` for MATLAB Compiler SDK.

Creation

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

- Create a standalone application using `compiler.build.standaloneApplication` (example on page 16-8).
- Create a standalone Windows application using `compiler.build.standaloneWindowsApplication` (example on page 16-9).
- Create a web app archive using `compiler.build.webAppArchive` (example on page 16-9).
- Create an Excel add-in using `compiler.build.excelAddIn` (example on page 16-10).

Properties

BuildType — Build type
`'standaloneApplication'` | `'standaloneWindowsApplication'` | `'webAppArchive'` | `'excelAddIn'`

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.excelAddIn	'excelAddIn'

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'}`

`'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'}`

`'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.

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

Data Types: `cell`

Options — Build options
StandaloneApplicationOptions | WebAppArchiveOptions | ExcelAddInOptions

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`
`'excelAddIn'`	`ExcelAddInOptions`

Examples

Get Build Information from Standalone Application

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

Compile using the file `magicsquare.m` located in *matlabroot*\extern\examples\compiler.

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

results =

  Results with properties:

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

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, and build options to a `compiler.build.Results` object.

Compile using the file `Mortgage.mlapp` located in *matlabroot*\examples\matlab\main.

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

results =

  Results with properties:

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

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, and build options to a `compiler.build.Results` object.

Compile using the file `Mortgage.mlapp` located in *matlabroot*\examples\matlab\main.

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

results =

  Results with properties:
```

```
      BuildType: 'webAppArchive'
          Files: {'D:\Documents\MATLAB\work\MortgagewebAppArchive\Mortgage.ctf'}
        Options: [1×1 compiler.build.WebAppArchiveOptions]
```

The `Files` property contains the path to the deployable archive file `Mortgage.ctf`.

Get Build Information from Excel Add-In

Create an Excel add-in and save information about the build type, generated files, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m` located in *matlabroot*\extern\examples\compiler.

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

results =

  Results with properties:

        BuildType: 'excelAddIn'
            Files: {2×1 cell}
          Options: [1×1 compiler.build.ExcelAddInOptions]
```

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.

See Also
`compiler.build.excelAddIn` | `compiler.build.standaloneApplication` | `compiler.build.standaloneWindowsApplication` | `compiler.build.webAppArchive`

Introduced in R2020b

compiler.build.standaloneApplication

Create a 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` located in *matlabroot*\extern\examples\compiler.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','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:

- `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`

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 1-11.

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')
```

The function generates the following files within a folder named `MagicDiagAppstandaloneApplication` in your current working directory:

- `MagicDiagApp.exe` or `MagicDiagApp`
- `run_MagicDiagApp.sh` (only on Linux and macOS systems)
- `mccExcludedFiles.log`
- `readme.txt`
- `requiredMCRProducts.txt`

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.

For this example, use the file `magicsquare.m` located in *matlabroot*\extern\examples\compiler.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
```

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\R2021a\toolbox\compiler\resources\default_ic
                ExecutableName: 'magicsquare'
        ExecutableSplashScreen: 'C:\Program Files\MATLAB\R2021a\toolbox\compiler\resources\default_sp
             ExecutableVersion: '1.0.0.0'
                      AppFile: 'C:\Program Files\MATLAB\R2021a\extern\examples\compiler\magicsquare
         TreatInputsAsNumeric: on
               AdditionalFiles: {}
          AutoDetectDataFiles: on
                      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, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m` located in *matlabroot*\extern\examples\compiler.

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

results =

  Results with properties:

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

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 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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'EmbedArchive','on'`

AdditionalFiles — Additional files
character vector | string scalar | cell array of character vectors | string array

Additional files to include in the standalone application, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File 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.

- 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 standalone archive
`'on'` (default) | on/off logical value

Flag to embed the standalone 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 standalone executable.
- If you set this property to `'off'`, then the function generates the standalone 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 `.ico`, `.jpg`, `.jpeg`, `.png`, `.bmp`, and `.gif`.

The default path is:

`'matlabroot\toolbox\compiler\resources\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\toolbox\compiler\resources\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`

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`

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.

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.

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:

- Build type, which is `'standaloneApplication'`
- Paths to the compiled files
- Build options, specified as a `StandaloneApplicationOptions` 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')
  ```

See Also
`applicationCompiler` | `compiler.build.StandaloneApplicationOptions` | `compiler.build.standaloneWindowsApplication` | `compiler.package.installer` | `mcc`

Topics
"Create Standalone Application from MATLAB" on page 1-5

Introduced in R2020b

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\R2021a\toolbox\compiler\resources\default_i
           ExecutableName: 'magicsquare'
    ExecutableSplashScreen: 'C:\Program Files\MATLAB\R2021a\toolbox\toolbox\compiler\resources\de
        ExecutableVersion: '1.0.0.0'
                  AppFile: 'C:\Program Files\MATLAB\R2021a\extern\examples\compiler\magicsquare
      TreatInputsAsNumeric: off
           AdditionalFiles: {}
        AutoDetectDataFiles: on
                  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\R2021a\toolbox\compiler\resources\default_ic
            ExecutableName: 'magicsquare'
    ExecutableSplashScreen: 'C:\Program Files\MATLAB\R2021a\toolbox\toolbox\compiler\resources\de
         ExecutableVersion: '1.0.0.0'
                   AppFile: 'C:\Program Files\MATLAB\R2021a\extern\examples\compiler\magicsquare
        TreatInputsAsNumeric: off
            AdditionalFiles: {}
        AutoDetectDataFiles: on
                    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\R2021a\toolbox\toolbox\compiler\resources\de
         ExecutableVersion: '2.0'
                   AppFile: 'D:\Documents\MATLAB\work\mymagic.m'
        TreatInputsAsNumeric: on
            AdditionalFiles: {}
        AutoDetectDataFiles: on
                    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'
```

```
opts =

  StandaloneApplicationOptions with properties:

          CustomHelpTextFile: ''
                 EmbedArchive: on
                ExecutableIcon: 'D:\Documents\MATLAB\work\images\magicicon.png'
               ExecutableName: 'mymagic'
      ExecutableSplashScreen: 'C:\Program Files\MATLAB\R2021a\toolbox\toolbox\compiler\resources\de
            ExecutableVersion: '2.0'
                      AppFile: 'D:\Documents\MATLAB\work\mymagic.m'
        TreatInputsAsNumeric: on
               AdditionalFiles: {}
          AutoDetectDataFiles: on
                      Verbose: on
                    OutputDir: 'D:\Documents\MATLAB\work\MagicApp'
```

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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and
`Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and
value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'EmbedArchive','on'`

AdditionalFiles — Additional files
character vector | string scalar | cell array of character vectors | string array

Additional files to include in the standalone application, specified as a character vector, a string
scalar, a string array, or a cell array of character vectors. File 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.

- 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 standalone archive
`'on'` (default) | on/off logical value

Flag to embed the standalone 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 standalone executable.

- If you set this property to `'off'`, then the function generates the standalone 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 `.ico`, `.jpg`, `.jpeg`, `.png`, `.bmp`, and `.gif`.

The default path is:

`'`*matlabroot*`\toolbox\compiler\resources\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\toolbox\compiler\resources\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`

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`

`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.

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.

Example: `'Verbose','On'`

Data Types: `logical`

Output Arguments

`opts` — Standalone application options object
`StandaloneApplictionOptions` object

Standalone application build options, returned as a `StandaloneApplictionOptions` object.

See Also
`compiler.build.standaloneApplication` |
`compiler.build.standaloneWindowsApplication` | `deploytool` | `mcc`

Introduced in R2020b

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:

- `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.
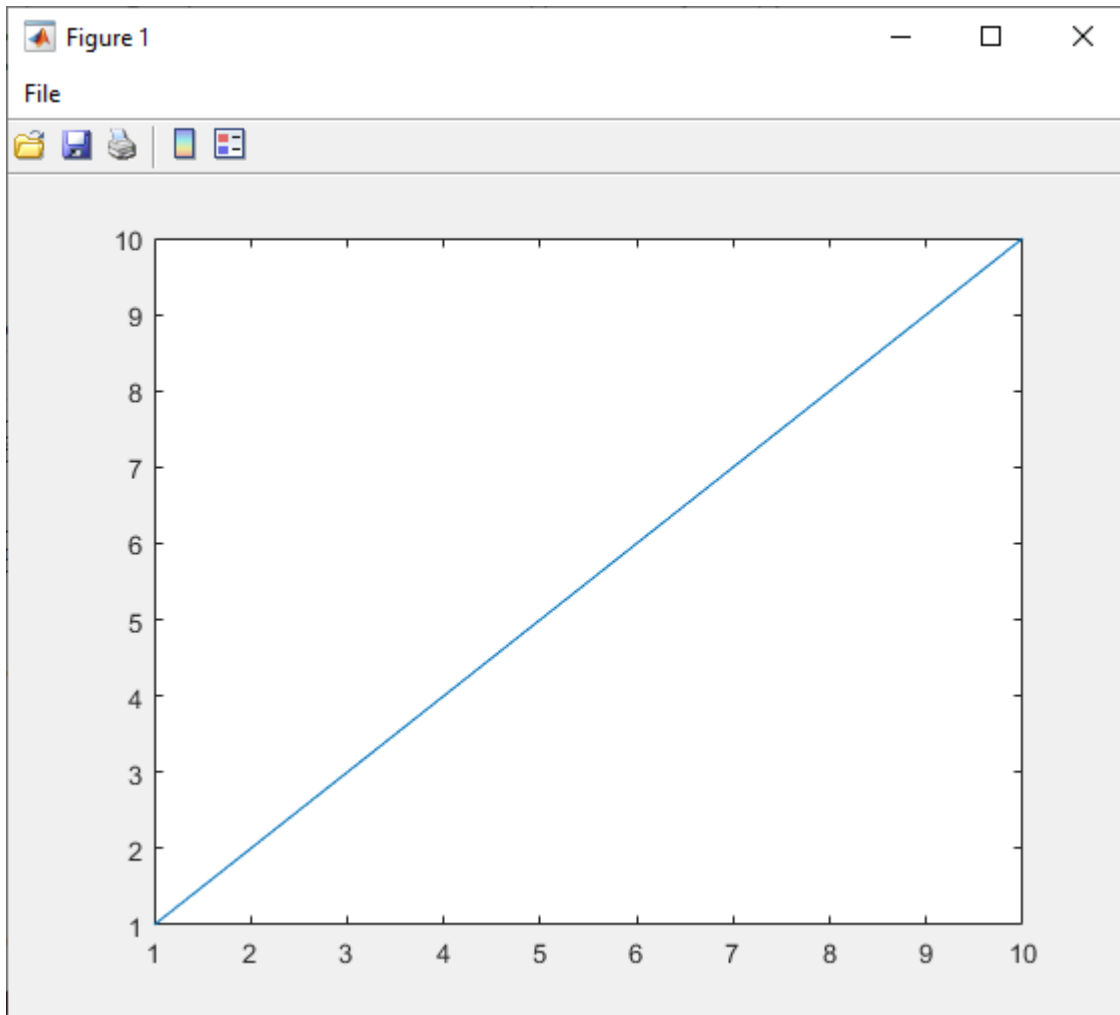
Figure 1 (myPlot.exe)

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)
```

16-27

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:

- `SineWaveApp.exe`
- `mccExcludedFiles.log`
- `readme.txt`
- `requiredMCRProducts.txt`
- `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.
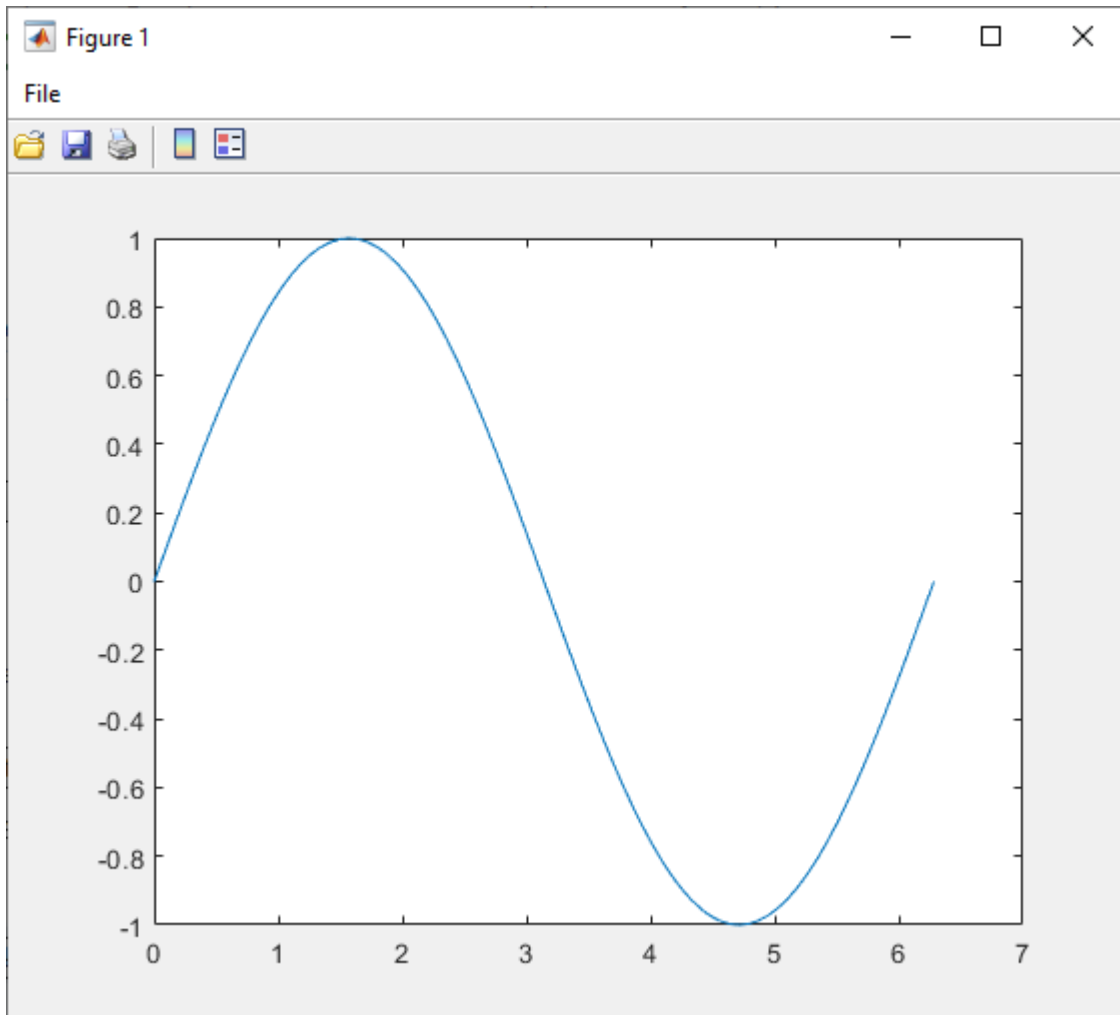
Figure 1 (SineWaveApp.exe)

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:

           ExecutableName: 'myPlot'
       CustomHelpTextFile: ''
             EmbedArchive: on
           ExecutableIcon: 'C:\Program Files\MATLAB\R2021a\toolbox\compiler\resources\default_i
   ExecutableSplashScreen: 'C:\Program Files\MATLAB\R2021a\toolbox\toolbox\compiler\resources\de
        ExecutableVersion: '1.0.0.0'
                  AppFile: 'myPlot.m'
       TreatInputsAsNumeric: on
          AdditionalFiles: {}
       AutoDetectDataFiles: on
                OutputDir: 'D:\Documents\MATLAB\work\WindowsApps'
                  Verbose: on
```

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, and build options to a `compiler.build.Results` object.

Compile using the file `Mortgage.mlapp` located in *matlabroot*\examples\matlab\main.

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

results =

  Results with properties:

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

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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'EmbedArchive','on'`

AdditionalFiles — Additional files
character vector | string scalar | cell array of character vectors | string array

Additional files to include in the standalone application, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File 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.

- 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 standalone archive
`'on'` (default) | on/off logical value

Flag to embed the standalone 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 standalone executable.

- If you set this property to `'off'`, then the function generates the standalone 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 `.ico`, `.jpg`, `.jpeg`, `.png`, `.bmp`, and `.gif`.

The default path is:

`'`*matlabroot*`\toolbox\compiler\resources\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\toolbox\compiler\resources\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`

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`

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.

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.

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:

- Build type, which is `'standaloneApplication`
- Paths to the following files:

    - *ExecutableName*`.exe`
    - `splash.png`
    - `readme.txt`
- 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')
  ```

See Also
`applicationCompiler` | `compiler.build.StandaloneApplicationOptions` | `compiler.build.standaloneApplication` | `compiler.package.installer` | `mcc`

Introduced in R2020b

compiler.package.docker

Create a Docker image for files generated by MATLAB Compiler on Linux operating systems

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

Caution This function is only supported on Linux operating systems.

`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);
```

The `Results` object is passed 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 Pairs

Customize a standalone application 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 the Docker image using the `Results` object and specify additional options as name-value pair 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 `hello-world.m` and save the build results to a `compiler.build.Results` object.

```
buildResults = compiler.build.standaloneApplication('hello-world.m');
```

Create a `DockerOptions` object to specify additional build options, such as the image name.

```
opts = compiler.package.DockerOptions(buildResults,
    'ImageName','hellodocker',...
    'ExecuteDockerBuild','Off');
```

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);
```

Create Docker Image Using Files and Name Value Pairs

Create a Docker image using files generated by the MATLAB Compiler and specify the image name on a Linux system.

Build a standalone application using the `mcc` command.

```
mcc -o runmyapp -m myapp.m
```

Build the Docker image by passing the generated files to the `compiler.package.docker` function.

```
compiler.package.docker('runmyapp','requiredMCRProducts.txt',...
    'ImageName','launchapp','EntryPoint','runmyapp');
```

Customize Docker Image Using Files and Options Object

Customize a Docker image using files generated by the MATLAB Compiler and a `DockerOptions` object on a Linux system.

Create a standalone application using `helloworld.m` and save the build results to a `compiler.build.Results` object..

```
buildResults = compiler.build.standaloneApplication('helloworld.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: 'helloworld'
    ExecuteDockerBuild: on
             ImageName: 'helloworld'
         DockerContext: './DockerImages'
```

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';
```

Build the Docker image by passing the generated files to the `compiler.package.docker` function.

```
cd helloworldstandaloneApplication

compiler.package.docker('helloworld','requiredMCRProducts.txt',...
    '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 the MATLAB Compiler product and can also

include any additional files and folders required by the installed application to run. Files generated by the MATLAB Compiler product 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 requiredMCRProducts.txt file
character vector | string scalar

Path to the `requiredMCRProducts.txt` file, specified as a character vector or string scalar. This file is generated by the MATLAB Compiler product. The path can be relative to the current working directory or absolute.

Example: `'/home/mluser/Documents/MATLAB/magicsquare/requiredMCRProducts.txt'`

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: `'hello-world'`

Data Types: `char` | `string`

opts — Docker options
`DockerOptions` object

Docker options, specified as a `DockerOptions` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'ExecuteDockerBuild','on'`

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*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`

Limitations

- Only standalone applications can be packaged into Docker images as of R2020b.

See Also
`compiler.build.Results` | `compiler.build.standaloneApplication` | `compiler.package.DockerOptions`

Topics
"Package MATLAB Standalone Applications into Docker Images" on page 14-2

Introduced in R2020b

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'
    ExecuteDockerBuild: on
             ImageName: 'magic-square-'
         DockerContext: './Docker/MagicSquare/magic-square-docker'
```

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: ''
    ExecuteDockerBuild: on
             ImageName: 'helloworld'
         DockerContext: './helloworlddocker'
```

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: ''
      ExecuteDockerBuild: off
             ImageName: 'helloworld'
          DockerContext: './helloworlddocker'
```

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'
      ExecuteDockerBuild: on
             ImageName: 'myapp-'
          DockerContext: './Docker/MyDockerApp'
```

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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'ExecuteDockerBuild','on'`

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`

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.

See Also

compiler.build.Results | compiler.build.standaloneApplication |
compiler.package.docker

Introduced in R2020b

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's name is specified by `appName`. The installer's extension is determined by the operating system you are running the function from.

`compiler.package.installer(files,filePath,'ApplicationName',appName, Name,Value)` creates an installer for files generated by the `mcc` command. The installed application's 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`.

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.

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. 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.

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.

16-47

```
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('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\R2021a\toolbox\toolbox\compiler\resources\de
              InstallerIcon: 'C:\Program Files\MATLAB\R2021a\toolbox\compiler\resources\default_ic
              InstallerLogo: 'C:\Program Files\MATLAB\R2021a\toolbox\compiler\resources\default_lc
                 AuthorName: 'Frog'
                AuthorEmail: ''
              AuthorCompany: 'Boston Common'
                    Summary: 'Generates a magic square.'
                Description: ''
          InstallationNotes: ''
                   Shortcut: ''
                    Version: '1.0'
              InstallerName: 'MagicSquare_Installer'
            ApplicationName: 'MagicSquare_Generator'
                  OutputDir: '.\MagicSquare_Generatorinstaller'
      DefaultInstallationDir: 'C:\Program Files\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

mymagic.exe
mccExcludedFiles.log
readme.txt
requiredMCRProducts.txt
```

Create an installer for the standalone application using the `compiler.package.installer` function.

```
compiler.package.installer(...
    'mymagic.exe','D:\Documents\MATLAB\work\MagicSquare\requiredMCRProducts.txt',...
    'ApplicationName','MagicSquare_Generator')
```

This generates an installer named `MyAppInstaller.exe` within a folder named `MagicSquare_Generatorinstaller`.

Customize Installer Using Files

Customize an installer for a standalone application on a Windows system using name-value arguments.

```
compiler.package.installer('mymagic.exe','requiredMCRProducts.txt',...
    'ApplicationName','MagicSquare_Generator',...
    'AuthorCompany','Boston Common',...
    'AuthorName','Frog',...
    'InstallerName','MagicSquare_Installer',...
    'Summary','Generates a magic square.')
```

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.')

opts =

  InstallerOptions with properties:

            RuntimeDelivery: 'web'
            InstallerSplash: 'C:\Program Files\MATLAB\R2021a\toolbox\toolbox\compiler\resources\de
              InstallerIcon: 'C:\Program Files\MATLAB\R2021a\toolbox\compiler\resources\default_ic
              InstallerLogo: 'C:\Program Files\MATLAB\R2021a\toolbox\compiler\resources\default_lc
                 AuthorName: 'Frog'
                AuthorEmail: ''
              AuthorCompany: 'Boston Common'
                    Summary: 'Generates a magic square.'
                Description: ''
          InstallationNotes: ''
                   Shortcut: ''
                    Version: '1.0'
              InstallerName: 'MagicSquare_Installer'
            ApplicationName: 'MagicSquare_Generator'
                  OutputDir: '.\MagicSquare_Generator'
    DefaultInstallationDir: 'C:\Program Files\MagicSquare_Generator'
```

Pass the `InstallerOptions` object as an input to the function.

16-49

```
compiler.package.installer('mymagic.exe','requiredMCRProducts.txt','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.

- 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'`

Data Types: `char` | `string`

`filePath` — Path to requiredMCRProducts.txt file
character vector | string scalar

Path to the `requiredMCRProducts.txt` file generated by MATLAB Compiler.

Example: `'D:\Documents\MATLAB\work\MagicSquare\requiredMCRProducts.txt'`

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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Version','9.5'` specifies the version of the installed application.

ApplicationName — Application name
`''` (default) | character vector | string scalar

Name of installed application, specified as a character vector or a string scalar.

Example: `'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: `'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: `'frog@bostoncommon.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: `'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: On Windows: `C:\Program Files\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: `'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: `'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\resources\default_icon_48.png'`

Example: `'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's logo, specified as a character vector or a string scalar. The logo will be resized to 150 pixels by 340 pixels.

The default path is:

`'`*matlabroot*`\toolbox\compiler\resources\default_logo.png'`

Example: `'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: `'MagicSquare_Installer'`

InstallerSplash — Path to splash screen image
character vector | string scalar

Path to an image file used as the installer's 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\toolbox\compiler\resources\default_splash.png'`

Example: `'D:\Documents\MATLAB\work\images\mySplash.png'`

`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: `'D:\Documents\MATLAB\work\MagicSquare'`

`RuntimeDelivery` — MATLAB Runtime delivery option
`'web'` (default) | `'installer'`

Choice on how the MATLAB Runtime is made available to the installed application.

- `'web'`—Option for installer to download MATLAB Runtime from MathWorks website during application installation. This is the default option.
- `'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 may not have access to the Internet.

Example: `'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: `'.\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: `'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: '2.0'

Data Types: char | string

See Also
compiler.package.InstallerOptions | mcc

Introduced in R2020a

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)
```

```
results =

  Results with properties:
```

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

```
opts = compiler.package.InstallerOptions(results,'AuthorName','Frog')
```

```
opts =

  InstallerOptions with properties:

            RuntimeDelivery: 'web'
            InstallerSplash: 'C:\Program Files\MATLAB\R2021a\toolbox\toolbox\compiler\resources\de
              InstallerIcon: 'C:\Program Files\MATLAB\R2021a\toolbox\compiler\resources\default_ic
              InstallerLogo: 'C:\Program Files\MATLAB\R2021a\toolbox\compiler\resources\default_lc
                 AuthorName: 'Frog'
                AuthorEmail: ''
              AuthorCompany: ''
                    Summary: ''
                Description: ''
          InstallationNotes: ''
                   Shortcut: ''
                    Version: ''
              InstallerName: 'MyAppInstaller'
            ApplicationName: 'magicsquare'
                  OutputDir: '.\magicsquare'
      DefaultInstallationDir: 'C:\Program Files\magicsquare'
```

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 `MyMagicInstaller.exe` within a folder named `magicsquareinstaller`.

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.')
```

```
opts =

  InstallerOptions with properties:

            RuntimeDelivery: 'web'
            InstallerSplash: 'C:\Program Files\MATLAB\R2021a\toolbox\toolbox\compiler\resources\de
              InstallerIcon: 'C:\Program Files\MATLAB\R2021a\toolbox\compiler\resources\default_ic
              InstallerLogo: 'C:\Program Files\MATLAB\R2021a\toolbox\compiler\resources\default_lc
```

```
             AuthorName: 'Frog'
            AuthorEmail: ''
          AuthorCompany: 'Boston Common'
                Summary: 'Generates a magic square.'
            Description: ''
      InstallationNotes: ''
               Shortcut: ''
                Version: '1.0'
          InstallerName: 'MagicSquare_Installer'
        ApplicationName: 'MagicSquare_Generator'
              OutputDir: '.\MagicSquare_Generator'
   DefaultInstallationDir: 'C:\Program Files\MagicSquare_Generator'
```

You can modify the property values of an existing `InstallerOptions` object using dot notation. For example, set the installation notes to `Windows installer`.

`opts.InstallationNotes = 'Windows installer'`

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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Version','9.5'` specifies the version of the installed application.

`ApplicationName` — Application name
`''` (default) | character vector | string scalar

Name of installed application, specified as a character vector or a string scalar.

Example: `'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: `'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: `'frog@bostoncommon.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: `'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: On Windows: `C:\Program Files\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: `'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: `'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\resources\default_icon_48.png'

Example: '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's logo, specified as a character vector or a string scalar. The logo will be resized to 150 pixels by 340 pixels.

The default path is:

'*matlabroot*\toolbox\compiler\resources\default_logo.png'

Example: '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: 'MagicSquare_Installer'

`InstallerSplash` — Path to splash screen image
character vector | string scalar

Path to an image file used as the installer's 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\toolbox\compiler\resources\default_splash.png'

Example: 'D:\Documents\MATLAB\work\images\mySplash.png'

`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: `'D:\Documents\MATLAB\work\MagicSquare'`

RuntimeDelivery — MATLAB Runtime delivery option
`'web'` (default) | `'installer'`

Choice on how the MATLAB Runtime is made available to the installed application.

- `'web'`—Option for installer to download MATLAB Runtime from MathWorks website during application installation. This is the default option.
- `'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 may not have access to the Internet.

Example: `'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: `'.\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: `'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: `'2.0'`

Data Types: `char` | `string`

Output Arguments

opts — Installer options object
`InstallerOptions` object

Installer options, returned as an `InstallerOptions` object.

See Also
`compiler.package.installer` | `mcc`

Introduced in R2020a

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..

Introduced in R2006a

deploytool

Open a list of application deployment apps

Syntax

```
deploytool
deploytool project_name
```

Description

`deploytool` opens a list of application deployment apps.

`deploytool project_name` opens the appropriate deployment app with the project preloaded.

Examples

Open a List of Application Deployment Apps

Open the list of apps.

```
deploytool
```

Input Arguments

project_name — name of the project to be opened
character array or string

Name of the project to be opened by the appropriate deployment app, specified as a character array or string. The project must be on the current path.

Compatibility Considerations

-build and -package options will be removed
Not recommended starting in R2020a

The `-build` and `-package` options will be 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.

Introduced in R2006b

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.

See Also
setmcruserdata

Introduced in R2008b

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 a standalone executable or shared library that runs outside of a MATLAB session using MATLAB Runtime libraries.

Examples

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 uses `addpath` before deployment.

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

Send Data to Printer

Deployed applications must use `deployprint`, rather than `print`, to send data to a printer.

```
if ~isdeployed
    print
else
   deployprint
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
```

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
deploytool | ismcc | mcc

Topics
"Write Deployable MATLAB Code" on page 5-9

Introduced before R2006a

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
```

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

Introduced in R2008b

libraryCompiler

Open the Library Compiler app

Syntax

```
libraryCompiler
libraryCompiler project_name
```

Description

`libraryCompiler` opens the Library Compiler app for the creation of a new compiler project

`libraryCompiler project_name` opens the Library Compiler app with the project preloaded.

Examples

Create a New Project

Open the Library Compiler app to create a new project.

```
libraryCompiler
```

Input Arguments

project_name — name of the project to be compiled
character array or string

Specify the name of a previously saved project. The project must be on the current path.

Compatibility Considerations

-build and -package options will be removed
Not recommended starting in R2020a

The `-build` and `-package` options will be 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.

Introduced in R2013b

mcc

Compile MATLAB functions for deployment

Syntax

```
mcc options mfilename1 mfilename2...mfilenameN

mcc -m options mfilename
mcc -e options mfilename

mcc -W 'excel:addin_name,className,version=version_number' -T link:lib
options mfilename1 mfilename2...mfilenameN

mcc -H -W hadoop:archiveName,CONFIG:configFile mfilename

mcc -m options mfilename
```

Description

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.

For information on compiling:

- C/C++ shared libraries, .NET assemblies, Java packages, or Python packages, see `mcc` for MATLAB Compiler SDK
- MATLAB Production Server deployable archives or Excel add-ins for MATLAB Production Server, see `mcc` for MATLAB Compiler SDK

Standalone Application

`mcc -m options mfilename` compiles the function into a standalone application.

This is equivalent to `mcc -W main -T link:exe`.

`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.

This syntax is equivalent to `-W WinMain -T link:exe`.

Excel Add-In

`mcc -W 'excel:addin_name,className,version=version_number' -T link:lib options mfilename1 mfilename2...mfilenameN` creates a Microsoft Excel add-in from the specified files.

- *addin_name* — Specifies the name of the add-in.

- *className* — 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. If specified, *className* needs to be different from *mfilename*.

- *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 0.

  - *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.

Note Excel add-ins can be created only in MATLAB running on Windows.

Note Remove the single quotes around 'excel:addin_name,className,version' when executing the mcc command from a DOS prompt.

MapReduce Applications on Hadoop

mcc -H -W hadoop:*archiveName*,CONFIG:*configFile* mfilename generates a deployable archive from mfilename that can be run as a job by Hadoop.

- *archiveName* — Specifies the name of the generated archive.

- *configFile* — 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".

Tip You can issue the mcc command either at the MATLAB command prompt or the Windows or Linux system command-line.

Simulink Simulations (Requires Simulink Compiler)

mcc -m options mfilename compiles a MATLAB application containing a Simulink simulation into a standalone application. For more information, see "Create and Deploy a Script with Simulink Compiler" (Simulink Compiler).

Examples

Create a standalone application

```
mcc -m magic.m
```

Create a standalone application that does not open the Command shell (Windows only)

```
mcc -e magic.m
```

Create a standalone application with a system-level file version number (Windows only)

Create a standalone application in Windows with version number 3.4.1.5.

```
mcc -W 'main:mymagic,version=3.4.1.5' -T link:exe mymagic.m
```

Create an Excel add-in

```
mcc -W 'excel:myAddin,myClass,1.0' -T link:lib magic.m
```

Create an Excel add-in with a system-level file version number (Windows only)

Create an Excel add-in in Windows with version number 5.2.1.7.

```
mcc -W 'excel:myAddin,myClass,version=5.2.1.7' -T link:lib -b mymagic.m
```

Create an Excel add-in for MATLAB Production Server

```
mcc -W 'mpsxl:myDeployableArchvie,myExcelClass,version=1.0' -T link:lib mymagic.m
```

Create a Standalone Application for a Simulink Simulation (Requires Simulink Compiler)

To create a standalone application for a Simulink simulation:

Create a Simulink model using Simulink. This example uses the model `sldemo_suspn_3dof` that ships with Simulink.

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
```

Input Arguments

`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.

options — Options for customizing the output
-a | -b | -B | -c | -C | -d | -f | -g | -G | -I | -K | -m | -M | -n | -N | -o | -p | -r | -R | -S | -T | -u | -U | -v | -w | -W | -X | -Y

Options for customizing the output, specified as a list of character vectors or string scalars.

- **-a**

  Add files to the deployable archive using `-a path` to specify the files to be added. Multiple `-a` options are permitted.

  If a file name is specified with `-a`, the compiler looks for these files on the MATLAB path, so specifying the full path name is optional. These files are not passed to `mbuild`, so you can include files such as data files.

  If 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 filename includes a 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,

  ```
  mcc -m hello.m -a ./testdir/*
  ```

  specifies that all files in `./testdir` are added to the deployable archive and subfolders under `./testdir` are not processed recursively.

  ```
  mcc -m hello.m -a ./testdir/*.m
  ```

  specifies that all files with the extension `.m` under `./testdir` are added to the deployable archive and subfolders of `./testdir` are not processed recursively.

  Note `*` is the only supported wildcard.

  When you add files to the archive using `-a` that do not appear on the MATLAB path at the time of compilation, a path entry is added to the application's run-time path so that they appear on the path when the deployed code executes.

  When you use the `-a` option to specify a full path to a resource, the basic path is preserved, with some modifications, but relative to a subdirectory of the runtime cache directory, not to the user's local folder. The cache directory is created from the deployable archive the first time the application is executed. You can use the `isdeployed` function to determine whether the application is being run in deployed mode, and adjust the path accordingly. The `-a` option also creates a `.auth` file for authorization purposes.

> **Caution** If you use the -a flag to include a file that is not on the MATLAB path, the folder containing the file is added to the MATLAB dependency analysis path. As a result, other files from that folder might be included in the compiled application.

> **Note** 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.

- **-b**

  Generate a Visual Basic file (`.bas`) containing the Microsoft Excel Formula Function interface to the COM object generated by MATLAB Compiler. When imported into the workbook Visual Basic code, this code allows the MATLAB function to be seen as a cell formula function.

- **-B**

  Replace the file on the `mcc` command line with the contents of the specified file. Use

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

  The bundle `filename` should contain only `mcc` command-line options and corresponding arguments and/or other file names. The file might contain other -B options. A bundle can include replacement parameters for compiler options that accept names and version numbers. See "Using Bundles to Build MATLAB Code" (MATLAB Compiler SDK).

- **-c**

  When used in conjunction with the -l option, suppresses compiling and linking of the generated C wrapper code. The -c option cannot be used independently of the -l option.

- **-C**

  Do not embed the deployable archive in binaries.

> **Note** The -C flag is ignored for Java libraries.

- **-d**

  Place output in a specified folder. Use

  ```
  -d outFolder
  ```

  to direct the generated files to *outFolder*. The specified folder must already exist.

- **-e**

  Use -e in place of the -m option to 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.

- **-f**

  Override the default options file with the specified options file. It specifically applies to the C/C++ shared libraries, COM, and Excel targets. Use

```
-f filename
```

to specify `filename` as the options file when calling `mbuild`. This option lets you use different ANSI compilers for different invocations of the compiler. This option is a direct pass-through to `mbuild`.

- **-g, -G**

  Include debugging symbol information for the C/C++ code generated by MATLAB Compiler SDK. It also causes `mbuild` to pass appropriate debugging flags to the system C/C++ compiler. The debug option lets you backtrace up to the point where you can identify if the failure occurred in the initialization of MATLAB Runtime, the function call, or the termination routine. This option does not let you debug your MATLAB files with a C/C++ debugger.

- **-I**

  Add a new folder path to the list of included folders. Each `-I` option appends the folder to the end of the list of paths to search. For example,

  ```
  -I <directory1> -I <directory2>
  ```

  sets up the search path so that `directory1` is searched first for MATLAB files, followed by `directory2`. This option is important for standalone compilation where the MATLAB path is not available.

  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.

- **-K**

  Direct `mcc` to not delete output files if the compilation ends prematurely due to error.

  The default behavior of `mcc` is to dispose of any partial output if the command fails to execute successfully.

- **-m**

  Direct `mcc` to generate a standalone application.

- **-M**

  Define compile-time options. Use

  ```
  -M string
  ```

  to pass `string` directly to `mbuild`. This option provides a useful 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 line	-M CFLAGS='$CFLAGS /bigobj'

- **-n**

  The -n option automatically identifies numeric command line inputs and treats them as MATLAB doubles.

- **-N**

  Passing -N 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
  - *matlabroot*\toolbox\shared\bigdata

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

  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.

- **-o**

  Specify the name of the final executable (standalone applications only). Use

  -o outputfile

  to name the final executable output of MATLAB Compiler. A suitable platform-dependent extension is added to the specified name (for example, .exe for Windows standalone applications).

- **-p**

  Use in conjunction with the option -N to add specific folders and subfolders under *matlabroot* \toolbox to the compilation MATLAB path. The files are added in the same order in which they appear in the MATLAB path. Use the syntax

  -N -p *directory*

  where directory is the folder to be included. If directory is not an absolute path, it is assumed to be under the current working folder.

  - 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 the same order.
  - If a folder is included with -p that is not on the original MATLAB path, that folder is ignored. (You can use -I to force its inclusion.)

- **-r**

  Embed resource icon in binary. The syntax is as follows:

  -r '*path/to/my_icon.ico*'

- **-R**

  Provide MATLAB Runtime options.

  Note This option is relevant only when building standalone applications using MATLAB Compiler.

  The syntax is as follows:

  -R *option*

Option	Description	Target
'-logfile, *filename*'	Specify a log file name. Option must be in single quotes.	MATLAB Compiler
-nodisplay	Suppress the MATLAB nodisplay run-time warning.	MATLAB Compiler
-nojvm	Do not use the Java Virtual Machine (JVM).	MATLAB Compiler
-startmsg	Customizable user message displayed at initialization time.	MATLAB Compiler Standalone Applications
-completemsg	Customizable user message displayed when initialization is complete.	MATLAB Compiler Standalone Applications
-singleCompThread	Limit MATLAB to a single computational thread.	MATLAB Compiler

  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.

  Note If you specify the -R option for libraries created from MATLAB Compiler SDK, mcc still compiles without errors and generates the results. But the -R option doesn't apply to these libraries and does not do anything.

- **-S**

  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 by the following products on these specific targets:

Target supported by Singleton MATLAB Runtime	Create a Singleton MATLAB Runtime by….
Excel add-in	Default behavior for target is singleton MATLAB Runtime. You do not need to perform other steps.
.NET assembly	Default behavior for target is singleton MATLAB Runtime. You do not need to perform other steps.
COM component	• Using the Library Compiler app, click **Settings** and add `-S` to the **Additional parameters passed to MCC** field.
Java package	• Using `mcc`, pass the `-S` flag.

- **-T**

  Specify the output target phase and type.

  Use the syntax `-T target` to define the output type.

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.
`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.

- **-u**

  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.

- **-U**

  Build deployable archive (`.ctf` file) for MATLAB Production Server.

- **-v**

  Display the compilation steps, including:

  - MATLAB Compiler version number
  - The source file names as they are processed
  - The names of the generated output files as they are created

- The invocation of `mbuild`

The `-v` option passes the `-v` option to `mbuild` and displays information about `mbuild`.

- **-w**

Display warning messages. Use the syntax

```
-w option [:<msg>]
```

to control the display of warnings.

Syntax	Description
`-w list`	List the compile-time warnings that have abbreviated identifiers, together with their status.
`-w enable`	Enable all compile-time warnings.
`-w disable[:<string>]`	Disable specific compile-time warnings associated with *<string>*. Omit the optional *<string>* to apply the `disable` action to all compile-time warnings.
`-w enable[:<string>]`	Enable specific compile-time warnings associated with *<string>*. Omit the optional *<string>* to apply the `enable` action to all compile-time warnings.
`-w error[:<string>]`	Treat specific compile-time and runtime warnings associated with *<string>* as an error. Omit the optional *<string>* to apply the `error` action to all compile-time and runtime warnings.
`-w off[:<string>]`	Turn off warnings for specific error messages defined by *<string>*. Omit the optional *<string>* to apply the `off` action to all runtime warnings.
`-w on[:<string>]`	Turn on runtime warnings associated with *<string>*. Omit the optional *<string>* to apply the `on` action to all runtime warnings. This option is enabled by default.

You can also turn warnings on or off in your MATLAB code.

For example, to turn off warnings for deployed applications (specified using `isdeployed`) in `startup.m`, you write:

```
if isdeployed
    warning off
end
```

To turn on warnings for deployed applications, you write:

```
if isdeployed
    warning on
end
```

You can also specify multiple `-w` options.

For example, if you want to disable all warnings except `repeated_file`, you write:

```
-w disable -w enable:repeated_file
```

When you specify multiple -w options, they are processed from left to right.

- **-W**

  Control the generation of function wrappers. Use the syntax

  ```
  -W type
  ```

  to control the generation of function wrappers for a collection of MATLAB files generated by the compiler. You provide a list of functions, and the compiler generates the wrapper functions and any appropriate global variable definitions.

Target	Syntax
Standalone Application	`-W 'main:appName,version=version'`
Standalone Application (no Windows console)	`-W 'WinMain:appName,version=version'`
Excel Add-In	`-W 'excel:addinName,className,version=version'`
Hadoop MapReduce Application	`-W 'hadoop:archiveName,CONFIG:configFile'`
Spark Application	`-W 'spark:appName,version'`

  Note Replace single quotes with double when executing the command from a Windows Command Prompt.

- -X

  Use -X to ignore data files read by common MATLAB file I/O functions during dependency analysis. For more information, see "Dependency Analysis Using MATLAB Compiler" on page 5-3. For examples on how to use the -X option, see `%#exclude`.

- **-Y**

  Use

  ```
  -Y license.lic
  ```

  to override the default license file with the specified argument.

  Note The -Y flag works only with the command-line mode.

  ```
  >>!mcc -m foo.m -Y license.lic
  ```

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 a standalone application with a system-level file version number (Windows only)" on page 16-72.

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. For supported targets in MATLAB Compiler SDK, see the **Tips** section in mcc.

See Also

Introduced before R2006a

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.

See "Install and Configure MATLAB Runtime" (MATLAB Compiler SDK)for more information about the MATLAB Runtime installer.

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)`).

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

Topics
"Install and Configure MATLAB Runtime" (MATLAB Compiler SDK)

Introduced in R2009a

mcrversion

Return the MATLAB Runtime version number matching the version of MATLAB

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`

See Also

`compiler.runtime.download` | `mcrinstaller`

Topics
"Install and Configure MATLAB Runtime" (MATLAB Compiler SDK)

Introduced in R2008a

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.

See Also
getmcruserdata

Introduced in R2008a

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 R2021a.

```
compiler.runtime.download
```

Downloading MATLAB Runtime installer. It may take several minutes...

MATLAB Runtime installer has been downloaded to:
 "C:\Users*username*\AppData\Local\Temp*username*\MCRInstaller9.10\MATLAB_Runtime_R2021a_win64.zip"

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 R2021a:

```
compiler.runtime.download
```

An existing MATLAB Runtime installer was found at:
 "C:\Users*username*\AppData\Local\Temp*username*\MCRInstaller9.10\MATLAB_Runtime_R2021a_win64.zip"

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.

Using MATLAB R2021a with no updates, display the download URL for MATLAB Runtime.

```
compiler.runtime.download
```

16-87

```
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/R2021a/Release/0/deployment_files/installer/comp
and update the runtime location in Compiler Settings.
```

The URL corresponds to the version and update level of MATLAB, as indicated by *<release>*/
Release/*<update_level>*.

See Also
mcrinstaller | mcrversion

Introduced in R2018a

MATLAB Compiler Quick Reference

mcc Command Arguments Listed Alphabetically

Option	Description	Comment
`-?`	Display help message.	Cannot be used in a `deploytool` app.
`-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.
`-b`	Generate Excel compatible formula function.	Requires MATLAB Compiler for Excel add-ins. Cannot be used in a `deploytool` app.
`-B` `filename[:arg[,arg]]`	Replace `-B filename` on the `mcc` command line with the contents of `filename`.	The file should contain only `mcc` command-line options. These are MathWorks included options files: • `-B csharedlib:foo` (C shared library) • `-B cpplib:foo` (C++ library) Cannot be used in a `deploytool` app.
`-c`	Generate C wrapper code.	Equivalent to `-T codegen`.
`-C`	Direct `mcc` to not embed the deployable archive in generated binaries.	
`-d` `directory`	Place output in specified folder.	The specified folder must already exist. Cannot be used in a `deploytool` app.
`-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`. Cannot be used in a `deploytool` app. The standalone app compiler suppresses the MS-DOS command window by default. To enable it, deselect **Do not require Windows Command Shell (console) for execution** in the **Additional Runtime Settings** area.
`-f` `filename`	Use the specified options file, `filename`, when calling `mbuild`.	`mbuild -setup` is recommended.
`-g`	Generate debugging information.	
`-G`	Same as `-g`.	
`-I` `directory`	Add folder to search path for MATLAB files.	
`-K`	Directs `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 function library.	Equivalent to `-W lib -T link:lib`. Cannot be used in a `deploytool` app.

Option	Description	Comment
`-m`	Generate a standalone application.	Equivalent to `-W main -T link:exe`. Cannot be used in a `deploytool` app.
`-M string`	Pass string to `mbuild`.	Use to define compile-time options.
`-n`	Automatically treat numeric inputs as MATLAB doubles.	Cannot be used in a `deploytool` app.
`-N`	Clear the 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 outputfile`	Specify name of final output file.	Adds appropriate extension. Cannot be used in a `deploytool` app.
`-p directory`	Add folder to compilation path in an order-sensitive context.	Requires `-N` option.
`-R option`	Specify run-time options for MATLAB Runtime.	Valid only for standalone applications using MATLAB Compiler. *option* = `-nojvm`, `-nodisplay`, `'-logfile` *filename*`'`, `-startmsg`, and `-completemsg` *filename*
`-S`	Create singleton MATLAB Runtime.	Default for generic COM components. Valid for Microsoft Excel and Java packages.
`-T`	Specify the output target phase and type.	Default is `codegen`. 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.
`-v`	Verbose; display compilation steps.	
`-w option`	Display warning messages.	*option* = `list`, *level*, or *level*:*string* where *level* = `disable`, `enable`, `error`, `off`:*string*, or `on`:*string*
`-W type`	Control the generation of function wrappers.	*type* = `main cpplib:<string> lib:<string> none com:compname,clname,version` Cannot be used in a `deploytool` app.
`-X`	Ignore data files detected by dependency analysis.	For more information, see "Dependency Analysis Using MATLAB Compiler" on page 5-3.

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`

Packaging Log and Output Folders

By default, the deployment app places the packaging log and the **Testing Files**, **End User Files**, and **Packaged Installers** folders 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 *filename*	Add `filename` to the deployable archive.	None
-C	Directs `mcc` to not embed the deployable archive in C/C++ and main/Winmain shared libraries and standalone binaries by default.	None

Debugging

Option	Description	Comment
-g	Generate debugging information.	None
-G	Same as `-g`	None
-K	Directs `mcc` to not delete output files if the compilation ends prematurely, due to error.	`mcc`'s default behavior is to dispose of any partial output if the command fails to execute successfully.
-v	Verbose; display compilation steps.	None
-W *type*	Control the generation of function wrappers.	`type` = `main` `cpplib:<string>` `lib:<string>` `none` `com:compname,clname,version`
-?	Display help message.	None

Dependency Function Processing

Option	Description	Comment
-a *filename*	Add `filename` to the deployable archive.	None

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`

MATLAB Compiler for Excel Add-Ins

Option	Description	Comment
-b	Generate Excel compatible formula function.	Requires MATLAB Compiler
-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.	MATLAB path is automatically included when running from MATLAB, but not when running from a DOS/UNIX shell.
-N	Clear the path of all but a minimal, required set of folders.	None
-p directory	Add `directory` to compilation path in an order-sensitive context.	Requires `-N` option

mbuild

Option	Description	Comment
-f filename	Use the specified options file, `filename`, when calling `mbuild`.	`mbuild -setup` is recommended.
-M string	Pass string to `mbuild`.	Use to define compile-time options.

MATLAB 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

Override Default Outputs

Option	Description	Comment
`-d directory`	Place output in specified folder.	None
`-o outputfile`	Specify name of final output file.	Adds appropriate extension
`-e`	Suppresses appearance of the MS-DOS Command Window when generating a standalone application.	Use `-e` in place of the `-m` option. Available for Windows only. Use with `-R` option to generate error logging. Equivalent to `-W WinMain -T link:exe` The standalone app compiler suppresses the MS-DOS command window by default. To unsuppress it, unselect **Do not require Windows Command Shell (console) for execution** in the app's **Additional Runtime Settings** area.

Wrappers and Libraries

Option	Description	Comment
`-c`	Generate C wrapper code.	Equivalent to `-T codegen`
`-l`	Macro to create a function library.	Equivalent to `-W lib -T link:lib`
`-m`	Macro to generate a standalone application.	Equivalent to `-W main -T link:exe`
`-W type`	Control the generation of function wrappers.	`type = main cpplib:<string> lib:<string> none com:compname,clname,version`

Using MATLAB Compiler on Mac or Linux

Problems Setting MATLAB Runtime Paths

In this section...
"Running SETENV on Mac Failed" on page B-2
"Mac Application Fails with "Library not loaded" or "Image not found"" on page B-2

When you build applications, associated shell scripts (run_*application*.sh) are automatically generated in the same folder as your binary. By running these scripts, you can conveniently set the path to your MATLAB Runtime location.

Running SETENV on Mac Failed

If the setenv command fails with a message similar to setenv: command not found or setenv: not found, you are not using a C Shell command interpreter (such as csh or tcsh).

Set the environment variables using the export command using the format export *my_variable*=*my_value*.

For example, to set DYLD_LIBRARY_PATH, run the following command:

export DYLD_LIBRARY_PATH=*mcr_root*/v910/runtime/maci64:*mcr_root*/ ...

Mac Application Fails with "Library not loaded" or "Image not found"

If you set your environment variables, you may still receive the following message when you run your application:

```
dyld: Library not loaded: @rpath/libmwlaunchermain.dylib
Referenced from: /Applications/magicsquare/application/
magicsquare.app/Contents/MacOS/magicsquare
  Reason: image not found
Trace/BPT trap: 5
```

You may have set your environment variables initially, but they were not set up as persistent variables. Do the following:

1   In your home directory, open a file such as .bashrc or .profile file in your log-in shell.

2   In either of these types of log-in shell files, add commands to set your environment variables so that they persist. For example, to set DYLD_LIBRARY_PATH in this manner, you enter the following in your file:

```
# Setting PATH for MCR

DYLD_LIBRARY_PATH=MCR_ROOT/v910/runtime/maci64:
MCR_ROOT/v910/sys/os/maci64:
MCR_ROOT/v910/bin/maci64
export DYLD_LIBRARY_PATH

?
```

Note The DYLD_LIBRARY_PATH= statement is one statement that must be entered as a single line. The statement is shown on different lines, in this example, for readability only.
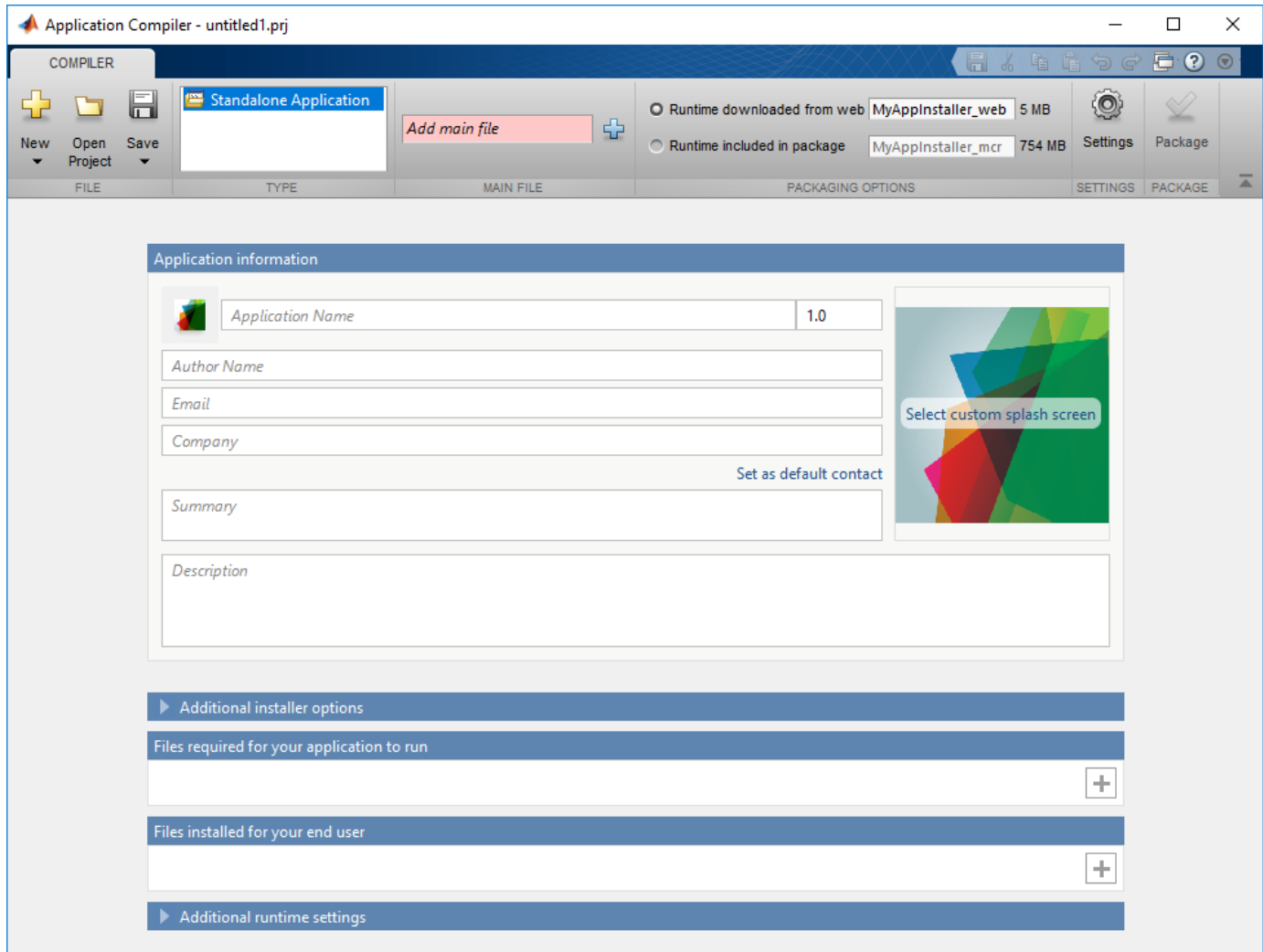
Apps

Application Compiler

Package MATLAB programs for deployment as standalone applications

Description

The **Application Compiler** app packages MATLAB programs into applications that can run outside of MATLAB.

Open the Application Compiler App

- MATLAB toolstrip: On the **Apps** tab, under **Application Deployment**, click the app icon.
- MATLAB command prompt: Enter `applicationCompiler`.

Examples

*   "Create Standalone Application from MATLAB" on page 1-5

Parameters

main file — name of the function to package
character vector

Name of the function to package as a character vector. The selected function is the entry point for the packaged application.

packaging options — method for installing the MATLAB Runtime with the packaged application
MATLAB Runtime downloaded from web (default) | MATLAB Runtime included in package

You can decide whether to include the MATLAB Runtime fallback for MATLAB Runtime installer in the generated application by selecting one of the two options in the **Packaging Options** section. Including the MATLAB Runtime installer in the package significantly increases the size of the package.

Runtime downloaded from web — Generates an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application.

Runtime included in package — Generates an installer that includes the MATLAB Runtime installer.

The first time you select this option, you are prompted to download the MATLAB Runtime installer or obtain a CD if you do not have internet access.

Files required for your application to run — files that must be included with application
list of files

Files that must be included with application as a list of files.

Files installed for your end user — files installed on the end user's machine when the application is installed
list of files

Optional files installed with application as a list of files.

Additional runtime settings — execution options for the application
check options

Check the appropriate boxes if you don't want a command window to show up during execution or if you want a log file to be created.

Settings

Additional parameters passed to MCC — flags controlling the behavior of the compiler
character vector

Flags controlling the behavior of the compiler as a character vector.

`Testing Files` — Folder where files for testing are stored
character vector

Folder where files for testing are stored as a character vector.

`End User Files` — Folder where files for building a custom installer are stored
character vector

Folder where files for building a custom installer are stored as a character vector.

`Packaged Installers` — Folder where generated installers are stored
character vector

Folder where generated installers are stored as a character vector.

Application information

`Application Name` — name of the installed application
character vector

Name of the installed application as a character vector.

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

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

`Version` — version of the generated application
character vector

Version of the generated application as a character vector.

`splash screen` — image displayed on installer
image

Image displayed on installer as an image.

`Author Name` — name of the application author
character vector

Name of the application author as a character vector.

`Email` — Email address used to contact application support
character vector

Email address used to contact application support as a character vector.

`Summary` — brief description of application
character vector

Brief description of application as a character vector.

`Description` — detailed description of application
character vector

Detailed description of application as a character vector.

Additional installer options

Default installation folder — Folder where application is installed
character vector

Folder where the application is installed as a character vector.

Installation notes — notes about additional requirements for using application
character vector

Notes about additional requirements for using application as a character vector.

Programmatic Use

applicationCompiler

See Also

Topics
"Create Standalone Application from MATLAB" on page 1-5

Introduced in R2013b

Hadoop Compiler

Package MATLAB programs for deployment to Hadoop clusters as MapReduce programs

---

Note The **Hadoop Compiler** app will be removed in a future release. To create standalone MATLAB® MapReduce applications, or deployable archives from MATLAB map and reduce functions, use the `mcc` command. For details, see "Compatibility Considerations".

Description

The **Hadoop Compiler** app packages MATLAB map and reduce functions into a deployable archive. You can incorporate the archive into a Hadoop mapreduce job by passing it as a payload argument to job submitted to a Hadoop cluster.
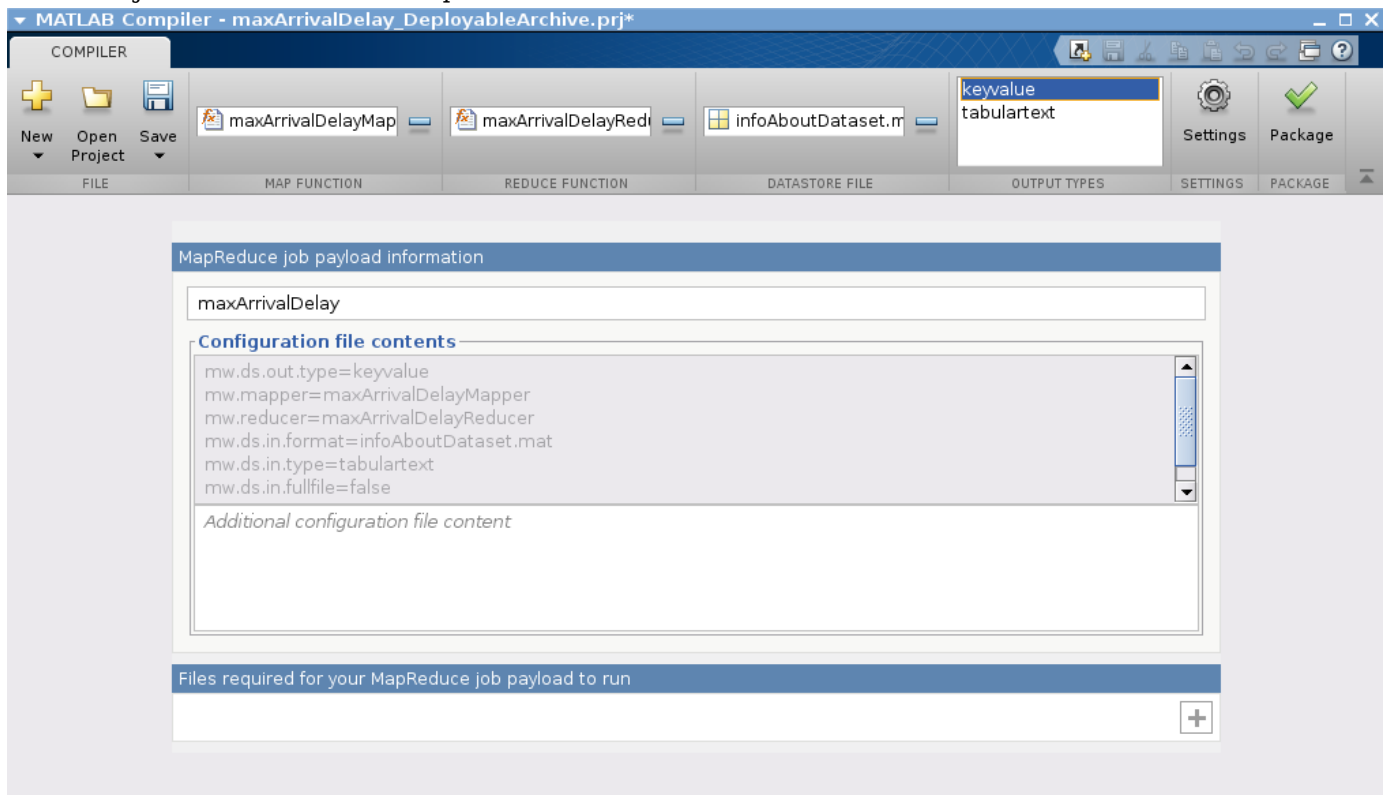
Open the Hadoop Compiler App

- MATLAB Toolstrip: On the **Apps** tab, under **Application Deployment**, click the app icon.
- MATLAB command prompt: Enter `hadoopCompiler`.

Parameters

`map function` — mapper file
character vector

Function for the mapper, specified as a character vector.

`reduce function` — reducer file
character vector

Function for the reducer, specified as a character vector.

`datastore file` — file containing a datastore representing the data to be processed
character vector

A file containing a datastore representing the data to be processed, specified as a character vector.

In most cases, you will start off by working on a small sample dataset residing on a local machine that is representative of the actual dataset on the cluster. This sample dataset has the same structure and variables as the actual dataset on the cluster. By creating a datastore object to the dataset residing on your local machine you are taking a snapshot of that structure. By having access to this datastore object, a Hadoop job executing on the cluster will know how to access and process the actual dataset residing on HDFS™.

`output types` — format of output
keyvalue (default) | tabulartext

Format of output from Hadoop mapreduce job, specified as a keyvalue or tabular text.

`additional configuration file content` — additional parameters configuring how Hadoop executes the job
character vector

Additional parameters to configure how Hadoop executes the job, specified as a character vector. For more information, see "Configuration File for Creating Deployable Archive Using the mcc Command".

`files required for your MapReduce job payload to run` — files that must be included with generated artifacts
list of files

Files that must be included with generated artifacts, specified as a list of files.

Settings

`Additional parameters passed to MCC` — flags controlling the behavior of the compiler
character vector

Flags controlling the behavior of the compiler, specified as a character vector.

`testing files` — folder where files for testing are stored
character vector

Folder where files for testing are stored, specified as a character vector.

packaged files — folder where generated artifacts are stored
character vector

Folder where generated artifacts are stored, specified as a character vector.

Compatibility Considerations

Hadoop Compiler will be removed
Not recommended starting in R2020a

Hadoop Compiler app will be removed in a future release. To create standalone MATLAB MapReduce applications, or deployable archives from MATLAB map and reduce functions, use the mcc command.

Introduced in R2014b