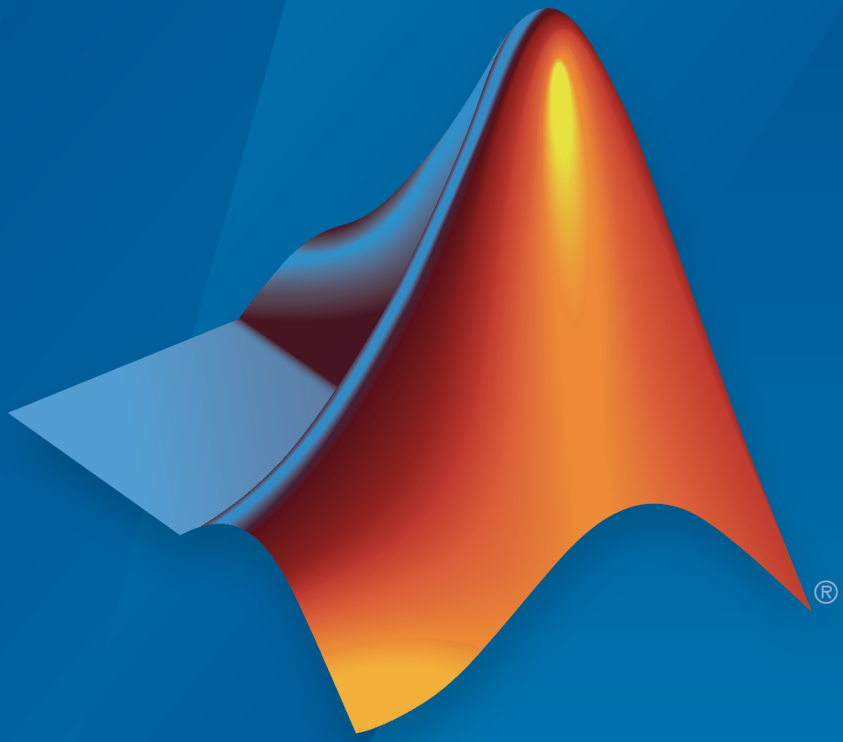
MATLAB® Compiler SDK™

MATLAB® Code Deployment Guide



MATLAB®

R2017a

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.
3 Apple Hill Drive
Natick, MA 01760-2098

MATLAB® Compiler SDK™ MATLAB® Code Deployment Guide

Revision History

March 2015	Online only	New for Version 6.0 (Release R2015a)
September 2015	Online only	Revised for Version 6.1 (Release 2015b)
October 2015	Online only	Rereleased for Version 6.0.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 6.2 (Release 2016a)
September 2016	Online only	Revised for Version 6.3 (Release R2016b)
March 2017	Online only	Revised for Version 6.3.1 (Release R2017a)

Contents

1 Overview

2 Write Deployable MATLAB Code

3 Compile a C/C++ Shared Library

4 Compile a .NET Assembly

5 Compile a Java Package

6 Compile a Python Package

7 Compile a Deployable Archive for MATLAB Production Server

8 Compile a COM Component

9 Customizing a Compiler Project

10 Advanced Uses of the Command Line Compiler

11 Work with the MATLAB Runtime

12 Limitations and Restrictions

1

Overview

Differences Between Compiler Apps and Command Line

You perform the same functions using either the compiler apps or the `mcc` command-line interface. The interactive menus and dialogs used in the compiler apps build `mcc` commands that are customized to your specification. As such, your MATLAB® code is processed the same way as if you were compiling it using `mcc`.

Compiler app advantages include:

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

How Does MATLAB Deploy Functions?

To deploy MATLAB functions, the compiler performs these tasks:

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

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

   For more information about how the compiler does dependency analysis, see "Dependency Analysis" on page 1-5.

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 1-4.

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

   For more details about deployable archives see "Deployable Archive" on page 1-7.

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.

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.

Dependency Analysis

In this section...

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

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

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 will not be able to determine the p-file's dependencies.
- Java classes and `.jar` files
- `.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`.

If you are using the compiler app, these data files are automatically added to the **Files required for your application to run** area of the app.

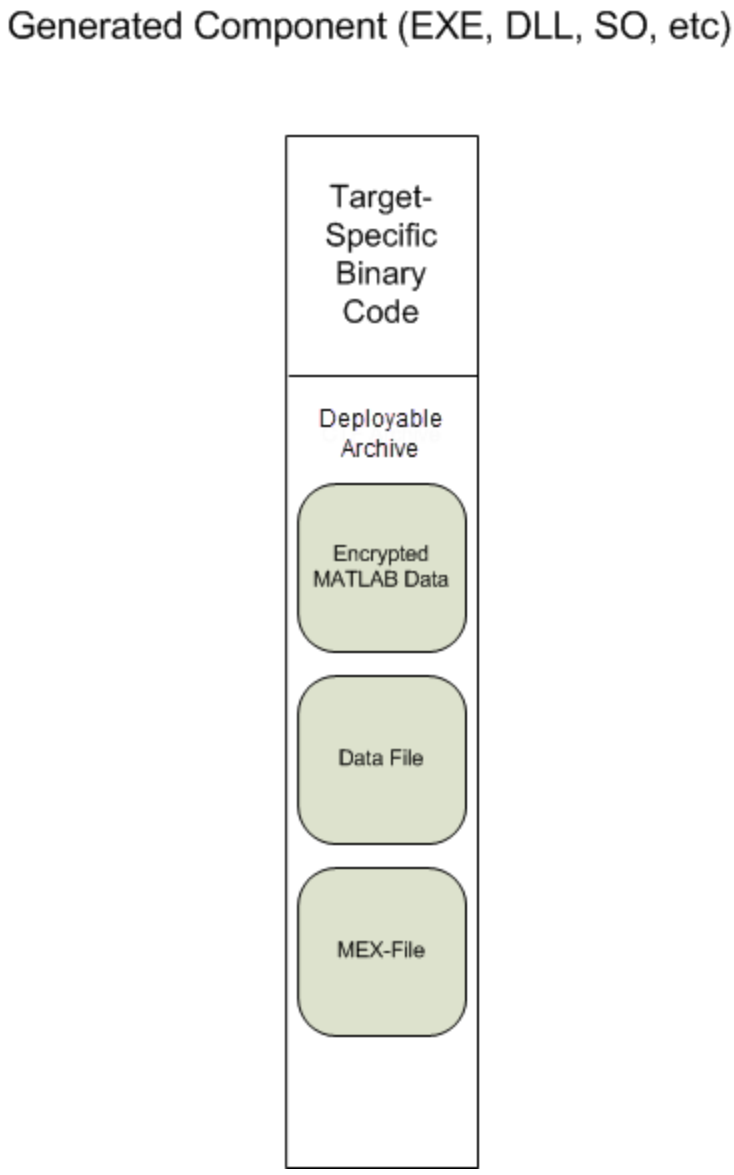
Deployable Archive

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 .NET integration	"MATLAB Runtime Component Cache and Deployable Archive Embedding"
MATLAB Compiler SDK Java integration	"Deployable Archive Embedding and Extraction"
MATLAB Compiler Excel® integration	"MATLAB Runtime Component Cache and Deployable Archive Embedding" (MATLAB Compiler)


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

Additional Details

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

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

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

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

Write Deployable MATLAB Code

Write Deployable MATLAB Code

In this section...

Compiled Applications Require Functions

Applications implemented with MATLAB Compiler SDK and MATLAB Production Server™ access MATLAB code through APIs generated from MATLAB functions. All MATLAB code compiled for use in these applications must be writen as a MATLAB function.

Compiled 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. 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.

The 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 the MATLAB Runtime.

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

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

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

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

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

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

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

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

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

If you find you cannot avoid placing `addpath` calls in your MATLAB code, use `ismcc` and `isdeployed`. See “Use isdeployed Functions To Execute Deployment-Specific Code Paths” on page 2-3 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 compiled and executed.

Gradually Refactor Applications That Depend on Noncompilable Functions

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

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

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

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

Do Not Create or Use Nonconstant Static State Variables

Avoid using the following:

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

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

When deploying applications, using persistent variables can cause problems because the 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 compiled MATLAB code, you should be aware that an instance of the 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 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.

State-Dependent Functions

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

Does My MATLAB Function Carry State?

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

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

Defensive Coding Practices

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

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

The following are examples of "defensive coding" practices:

Reset System-Generated Values in the Deployed Application

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

Validate Global or Persistent Variable Values

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

Ensure Access to Data Caches

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

Use Simple Data Types When Possible

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

Avoid Using MATLAB Callback Functions

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

Techniques for Preserving State

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

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

Caution: Using MATLAB `LOAD` and `SAVE` functions is often used to preserve state in MATLAB applications and workspaces. While this may be successful in some

circumstances, it is highly recommended that the data be validated and reset if needed, if not stored in a generic repository such as a database.

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 thatyou 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" (MATLAB).

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

See Also

`loadlibrary`

Related Examples

- "Call Functions in Shared Libraries" (MATLAB)

MATLAB Data Files in Compiled Applications

In this section...

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" on page 1-5.

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 Neural Network 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 1-7.

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 excryption 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');
```

Share MATLAB Runtime Instances

In this section...

What Is a Singleton MATLAB Runtime?

You create an instance of the MATLAB Runtime that can be shared among all subsequent class instances within a component. This is commonly called a shared MATLAB Runtime instance or a *Singleton runtime*.

Advantages and Disadvantages of Using a Singleton

In most cases, a singleton MATLAB Runtime will provide many more advantages than disadvantages. Following are examples of when you might and might not create a shared MATLAB Runtime instance.

When You Should Use a Singleton

If you have multiple users running from a specific instance of MATLAB, using a singleton will most likely:

- Utilize system memory more efficiently
- Decrease MATLAB Runtime start-up or initialization time

When You Might Avoid Using a Singleton

Using a singleton may not benefit you if your application uses a large number of global variables. This causes crosstalk.

Compile a C/C++ Shared Library

Install an ANSI C or C++ Compiler

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

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

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

Supported ANSI C and C++ Windows Compilers

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

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

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

Supported ANSI C and C++ UNIX Compilers

MATLAB Compiler and MATLAB Compiler SDK support the native system compilers on:

- Linux®
- Linux x86-64
- Mac OS X

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

Common Installation Issues and Parameters

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

Windows Operating System

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

UNIX Operating System

Issue	Comment
Determine which C or C++ compiler is available on your system.	See your system administrator.
Determine the path to your C or C++ compiler.	See your system administrator.

Issue	Comment
Installing on `Maci64`	Install X Code from installation DVD.

Compile C/C++ Shared Libraries with Library Compiler App

To compile MATLAB code into a shared library:

1. Open the Library Compiler app.

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

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

   Note: You can also launch the Library Compiler app using the `libraryCompiler` function.

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

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

3. Specify the MATLAB files you want deployed in the shared library.

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

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

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

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

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

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

You can select one or both of the following options:

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

Note: Selecting both options creates two installers.

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

5 Specify the name of any generated installers.

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

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

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

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

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

For more information see "Customize the Installer" on page 9-2.

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

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

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

For more information see “Manage Required Files in Compiler Project” on page 9-6.

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

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

This section automatically lists:

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

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

For more information see “Specify Files to Install with Application” on page 9-8.

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

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

Compile C/C++ Shared Libraries from Command Line

In this section...

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

- `deploytool` invokes the compiler to execute presaved compiler projects
- `mcc` invokes the command line compiler

Execute Compiler Projects with deploytool

The `deploytool` command has two flags to invoke one of the compiler apps without opening a window.

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

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

Compile a Shared Library with mcc

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

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

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

Compiler Shared Library Options

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

Distribute C/C++ Shared Libraries to Application Developers

Distribute the following to the application developer integrating the shared library:

- Function signatures of the deployed MATLAB functions
- Generated shared library and header file
- MATLAB Runtime installer

The Library Compiler app generates an installer that packages all of the binary artifacts required for distributing a shared library. The installer is located in the `for_redistribution` folder of the compiler project.

Compile a .NET Assembly

Create a .NET Assembly

This example shows how to transform a MATLAB function into a .NET assembly. The example compiles a MATLAB function, `makesquare`, which computes a magic square, into a .NET assembly.

1. In MATLAB, create the function that you want to deploy as a shared library.

   This example uses the sample function, called `makesquare.m`, included in the *`matlabroot`*`\toolbox\dotnetbuilder\Examples\VS`*`Version`*`\NET\MagicSquareExample\MagicSquareComp` folder, where *`matlabroot`* represents the name of your MATLAB installation folder.

   ```
   function y = makesquare(x)

   y = magic(x);
   ```

   Run the example in MATLAB.

   ```
   makesquare(5)
   ```

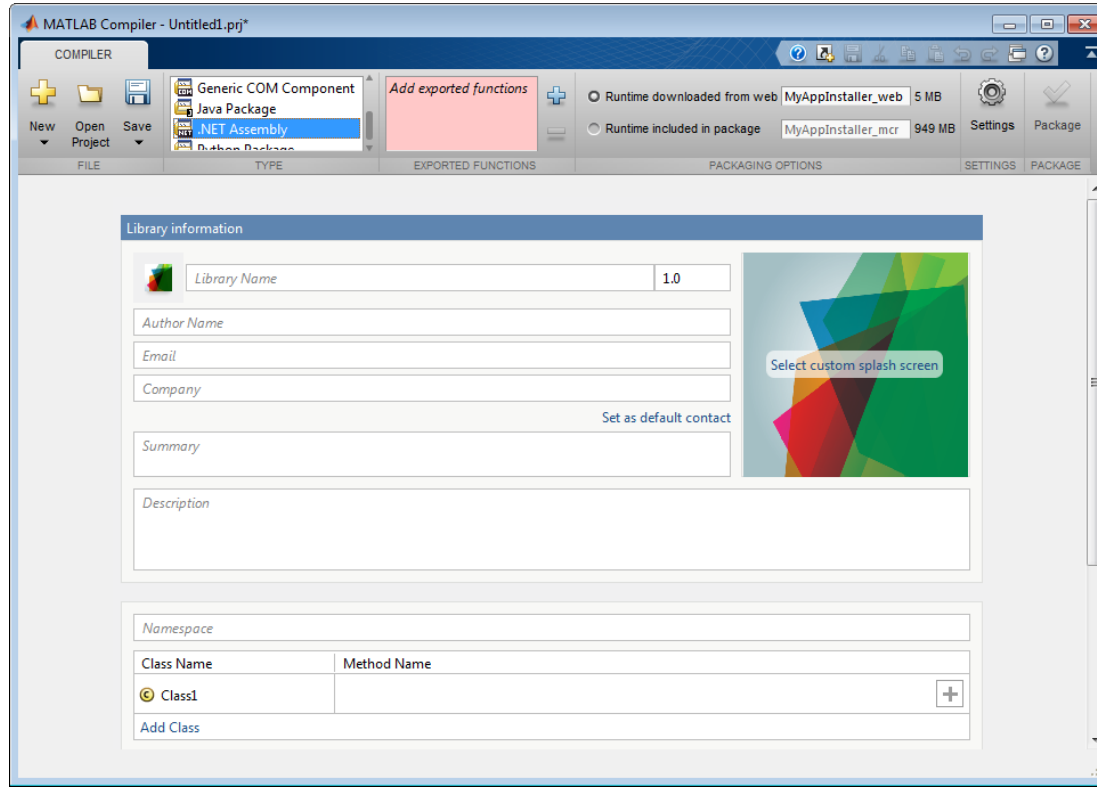
   ```
   ans =

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

2. Open the Library Compiler app.

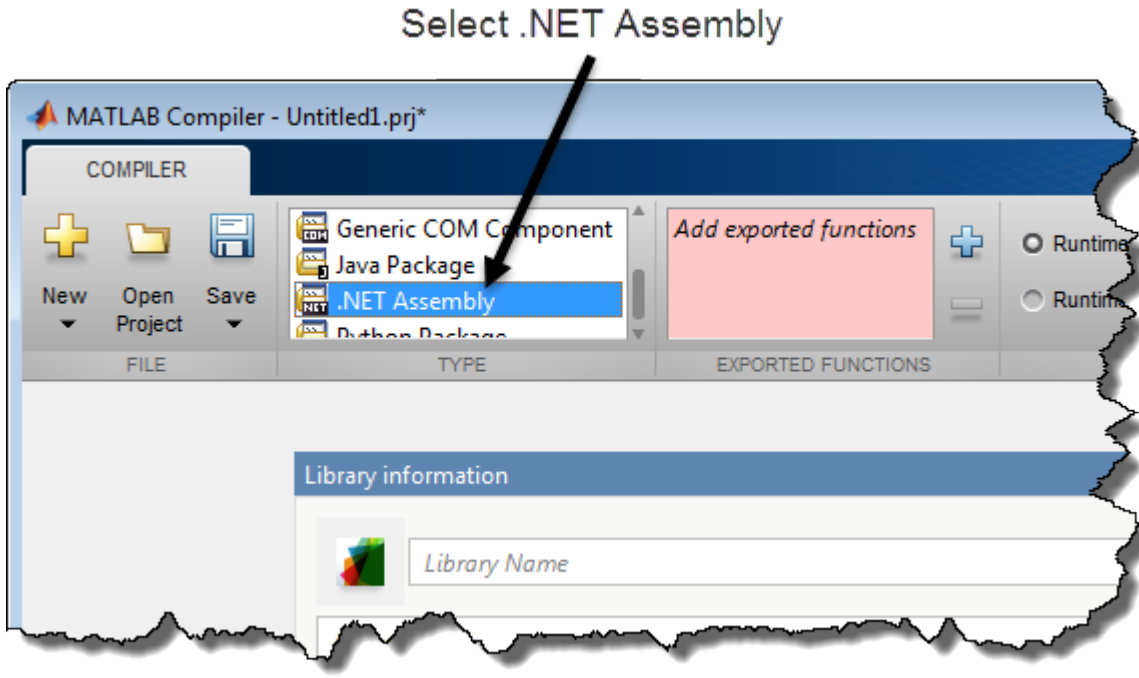
   a. On the toolstrip, select the **Apps** tab.
   b. Click the arrow at the far right of the tab to open the apps gallery.
   c. Click **Library Compiler**.

   Note: You can also call the `libraryCompiler` command.



3 In the **Type** section of the toolstrip, select **.NET Assembly** from the list.

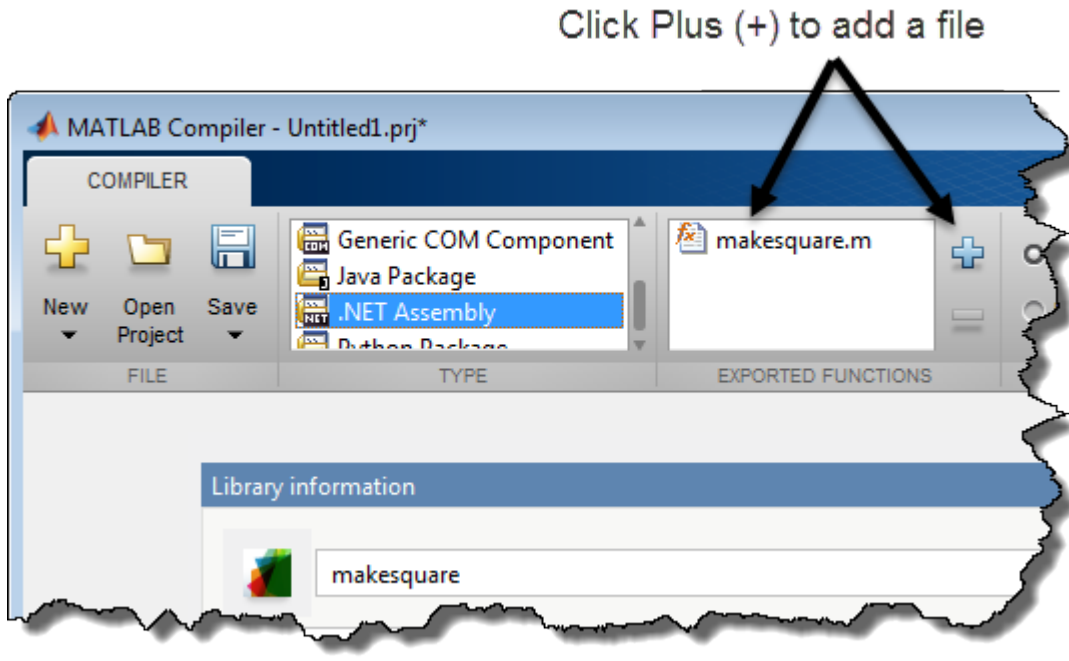
Note: If the **Type** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.



4 Specify the MATLAB functions that you want to deploy.

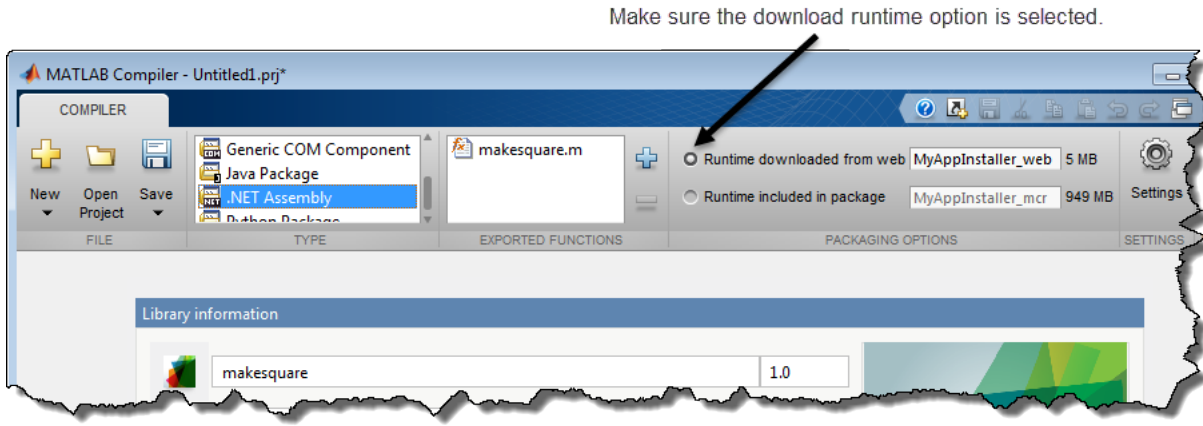
a In the **Exported Functions** section of the toolstrip, click the plus button.
b In the file explorer that opens, locate and select the `makesquare.m` file.
c Click **Open** to select the file and close the file explorer.

The Library Compiler app adds **makesquare.m** to the list of files and a minus button appears under the plus button. The Library Compiler app uses the name of the file as the name of the deployment project file (`.prj`), shown in the title bar, and as the name of the assembly, shown in the first field of the Library Information area. The project file saves all of the deployment settings so that you can re-open the project.



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

This option creates an application installer that automatically downloads the MATLAB Runtime and installs it along with the deployed add-in.



6 In the top field of **Library Information**, replace `makesquare` with `MagicSquareComp`.
7 In the **Class Name** column of the class browser, replace `Class1` with `MLTestClass`.
8 Click **Save** to save the project.
9 Click **Package**.

   The Package window opens while the library is being generated. Select the **Open output folder when process completes** check box. The packaging process generates a self-extracting file that *automatically registers* the DLL and unpacks all deployable deliverables.
10 When the deployment process is complete, a file explorer opens and displays the generated output.

   It should contain:

   - `for_redistribution` — A folder containing the installer to distribute the generated assembly
   - `for_testing` — A folder containing the raw files generated by the compiler
   - `for_redistribution_files_only` — A folder containing only the files needed to redistribute the assembly
   - `PackagingLog.txt` — A log file generated by the compiler
11 Click **Close** on the Package window.

Compile .NET Assemblies from Command Line

In this section...

Command-Line Syntax Description

Instead of using the Library Compiler app to create .NET assemblies, you can use the `mcc` command.

The following command defines the complete `mcc` command syntax with all required and optional arguments used to create a .NET assembly. Brackets indicate optional parts of the syntax.

```
mcc -W 'dotnet:component_name,class_name, 0.0|framework_version,
Private|Encryption_Key_Path,local|remote' file1
[file2...fileN][class{class_name:file1 [,file2,...,fileN]},...[-d
output_dir_path] -T link:lib
```

.NET Bundle

You can simplify the command line used to create .NET assemblies. To do so, use the bundle named `dotnet`. Using this bundle still requires that you pass in the five parts (including `local|remote`) of the `-W` argument text string; however, you do not have to specify the `-T` option.

The following example creates a .NET assembly called `mycomponent` containing a single .NET class named `myclass` with methods `foo` and `bar`.

```
mcc -B 'dotnet:mycomponent,myclass,2.0,
    encryption_keyfile_path,local'
    foo.m bar.m
```

In this example, the compiler uses the .NET Framework version 2.0 to compile the component into a shared assembly using the key file specified in `encryption_keyfile_path` to sign the shared component.

Creating a .NET Namespace

The following example creates a .NET assembly from two MATLAB files `foo.m` and `bar.m`.

```
mcc -B
'dotnet:mycompany.mygroup.mycomponent,myclass,0.0,Private,local'
 foo.m bar.m
```

The example creates a .NET assembly named `mycomponent` that has the following namespace: `mycompany.mygroup`. The component contains a single .NET class, `myclass`, which contains methods `foo` and `bar`.

To use `myclass`, place the following statement in your code:

```
using mycompany.mygroup;
```

Adding Multiple Classes to an Assembly

The following example creates a .NET assembly that includes more than one class. This example uses the optional `class{...}` argument to the `mcc` command.

```
mcc -B 'dotnet:mycompany.mycomponent,myclass,2.0,Private,local' foo.m bar.m
class{myclass2:foo2.m,bar2.m}
```

The example creates a .NET assembly named `mycomponent` with two classes:

- `myclass` has methods `foo` and `bar`
- `myclass2` has methods `foo2` and `bar2`

See for a list of supported framework versions.

Execute Compiler Projects with deploytool

The `deploytool` command has two flags to invoke one of the compiler apps without opening a window.

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

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

Distribute .NET Assemblies to Application Developers

Distribute the following to the application developer integrating the .NET assembly:

- Function signatures of the deployed MATLAB functions
- *assemblyName*`.xml` — generated documentation files
- *assemblyName*`.dll` — generated assembly file
- *assemblyName*`.pdb` — optionally generated program database file containing debugging information
- MATLAB Runtime installer

The Library Compiler app generates an installer that packages all of the binary artifacts required for distributing a .NET assembly. The installer is located in the `for_redistribution` folder of the compiler project.

Compile a Java Package

Configure Your Java Environment

In this section...

Before you can compile MATLAB functions into Java packages or use the generated Java packages in a Java development environment, you need to ensure that your Java environment is properly configured. You must verify that:

- Your system uses the same version of the Java Developer’s Kit (JDK™) as MATLAB.
- `JAVA_HOME` is set to the folder containing the system’s JDK installation.
- `CLASSPATH` contains all of the MATLAB library JAR files and the JAR files for the packages containing your compiled MATLAB code.
- The MATLAB native library paths are properly configured.

Note: For updated Java system requirements, including versions of Java Developer's Kit (JDK) and Java Runtime Environment (JRE), see the supported compiler page at http://www.mathworks.com/support/compilers/current_release/.

Install the Required JDK

To install the proper version of the JDK:

1. Verify the version of Java your MATLAB installation is using by running the following MATLAB command:

   ```
   version -java
   ```
2. Download the matching version Java Developer's Kit (JDK) from http://www.oracle.com/technetwork/java/javase/downloads/index.html.
3. Install the JDK, following the instructions provided by Oracle®.

Note: If you are not developing applications or compiling MATLAB code, you can use the Java Runtime Environment (JRE) instead of the JDK.

Set JAVA_HOME

1. Set the system environment variable, `JAVA_HOME`, to point to your JDK installation.
2. At the MATLAB command prompt, type `getenv JAVA_HOME` to verify that MATLAB is reading the correct version of `JAVA_HOME`.
3. Verify that the folder containing your Java installation has been added to your system `PATH` environment variable.

Set the CLASSPATH

To build and run a Java application that uses a MATLAB Compiler SDK generated package, the system must locate:

- JAR files containing the MATLAB libraries
- Packages that you have developed and built with the compiler

Java classes generated by the MATLAB Compiler SDK software use classes contained in the `com.mathworks.toolbox.javabuilder` package. To use the compiled classes, you need to include a file called `javabuilder.jar` on the Java class path. You can find this file in one of the following folders:

MATLAB installed on your system	*`matlabroot`*`/toolbox/javabuilder/jar`
MATLAB Runtime installed on your system	*`mcrroot`*`/toolbox/javabuilder/jar`

Note: *`matlabroot`* refers to the root folder into which the MATLAB installer has placed the MATLAB files. *`mcrroot`* refers to the root folder under which MATLAB Runtime is installed.

In addition, you need to add to the JAR files created by the compiler to the class path.

Configure the Native Library Path Variables

The operating system uses the native library path to locate native libraries that are needed to run your Java class. See the following list of variable names according to operating system:

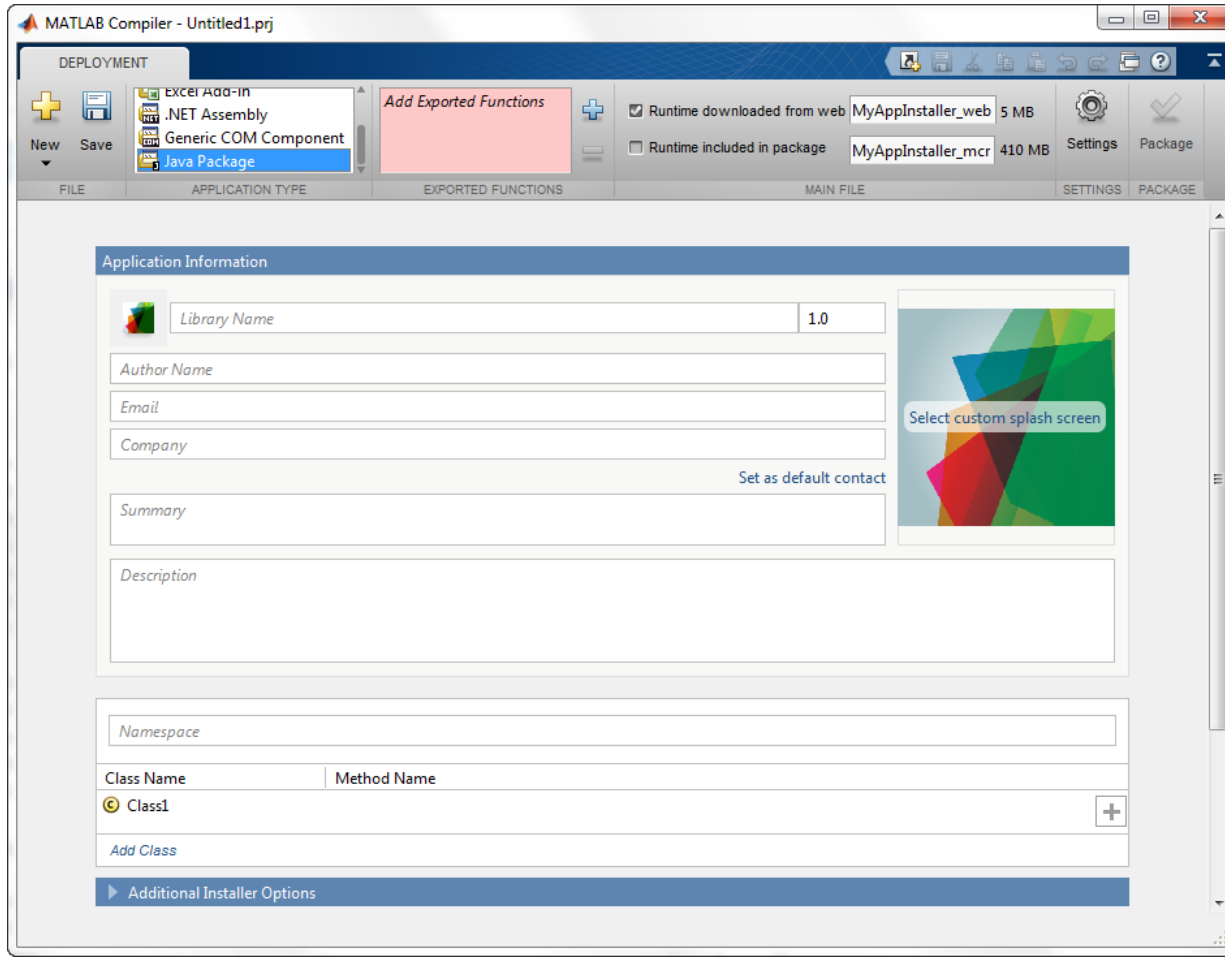
Windows	`PATH`
Linux	`LD_LIBRARY_PATH`
Macintosh	`DYLD_LIBRARY_PATH`

The native MATLAB or MATLAB Runtime files needed to execute the compiled MATLAB functions called from the Java code must be included on the paths listed by your system's native library path variable.

Compile Java Packages with Library Compiler App

To compile MATLAB code into a Java package:

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



Note: You can also start the Shared Library Compiler app using the `libraryCompiler` function.

2 In the **Application Type** section of the toolstrip, select **Java Package**.

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

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

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

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

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

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

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

4 Verify that the functions defined in the selected files are properly mapped into classes.

For more information, see “Map Functions to Java Class Methods” on page 5-12.

5 In the **Packaging Options** section of the toolstrip, specify how the installer will deliver the MATLAB Runtime with the package.

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

You can select one or both of the following options:

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

Note: Selecting both options creates two installers.

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

6 Specify the name of any generated installers.

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

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

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

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

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

For more information, see "Customize the Installer" on page 9-2.

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

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

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

For more information, see "Manage Required Files in Compiler Project" on page 9-6.

9 In the **Files installed for your end user** section of the app, verify that any additional non-MATLAB files you want installed with the application are listed.

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

This section automatically lists:

- Generated package
- `doc` folder containing the Javadoc for the generated classes
- Readme file

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

For more information, see "Specify Files to Install with Application" on page 9-8.

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

Note: To create a package that uses a singleton MATLAB Runtime, pass the `-S` flag to the compiler. For more information, see "Share MATLAB Runtime Instances" on page 2-14.

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

12 Verify that the generated output contains:

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

Compile Java Packages from Command Line

In this section...

You can compile Java packages from both the MATLAB command line and the system terminal command line:

- `deploytool` invokes one of the compiler apps to execute a presaved compiler project
- `mcc` invokes the command line compiler

Execute Compiler Projects with deploytool

The `deploytool` command has two flags to invoke one of the compiler apps without opening a window.

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

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

Compile a Java Package with mcc

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

To invoke the compiler to generate a Java package use the `-W java:`*`packageName`*`,`*`className`* flag with `mcc`. This flag creates a Java package named *`packageName`*. The package contains a class *`className`* with methods for each of the provided MATLAB functions.

For compiling Java packages, you can also use the following options.

Compiler Java Options

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

Map Functions to Java Class Methods

In this section...

Map Functions to Java Classes with the Library Compiler App

The Library Compiler app presents a visual class mapper for mapping MATLAB functions to Java classes. The class mapper is located between the **Application Information** and the **Additional Installer Options** sections of the app.

The **Namespace** field at the top of the class browser specifies the name of the package into which the generated classes are placed. By default, the name of the first listed MATLAB file is used as the package name. You can change the package name to fit the naming conventions used by your organization.

The table used to match functions to classes is below the package name. The **Class Name** column specifies the name of the generated Java class. The **Method Name** column specifies the list of MATLAB functions that are mapped into methods of the generated class.

Add a New Class to a Java Package

To add a class to a Java package:

1. Click **Add Class**.
2. Rename the class as described in “Rename a Java Class” on page 5-13.
3. Add one or more methods to the class as described in “Add a Method to a Java Class” on page 5-13.

Rename a Java Class

To rename a Java class:

1. Select the name of the class to be renamed.
2. Open the context menu.
3. Select **Rename**.
4. Enter the new class name.

   The class name must follow the Java naming guidelines. It cannot contain any special characters, dots, or spaces.

Delete a Class from a Java Package

To delete a class from a Java package:

1. Select the name of the class to be deleted.
2. Open the context menu.
3. Select **Delete**.

Add a Method to a Java Class

To add a method to a Java class:

1. In the **Method Name** column of the row for the class to which the method is being added, click the plus button.
2. Select the name of the function to add.

Delete a Method from a Java Class

To delete a method from a Java class:

1. Select the name of the function to be deleted.
2. Open the context menu.
3. Select **Delete**.

Tip: You can also delete the method using the **Delete** key.

Map Functions to Java Classes with mcc

When using `mcc` to generate Java packages, you map your MATLAB functions into Java classes based on the list into which they are placed on the command line. Class groupings

are specified by adding one or more `class{`*`className`*`:`*`mfilename`*`...}` entries to the command line. All of the files not specifically included in a class grouping are added to the class specified by the `-W java:`*`packageName`*`,`*`className`* flag.

For example, `mcc –W java:myPackage,MyClass fun1.m fun2.m fun3.m` generates a Java package `myPackage` that contains a single class `MyClass`. `MyClass` has three methods: `fun1`, `fun2`, and `fun3`.

However, `mcc –W java:myPackage,MyClass fun1.m fun2.m class{MyOtherClass:fun3.m}` generates a Java package `myPackage` that contains two classes: `MyClass` and `MyOtherClass`. `MyClass` has two methods: `fun1` and `fun2`. `MyOtherClass` has one method `fun3`.

Distribute Java Packages to Application Developers

Distribute the following to the application developer integrating the package:

- Function signatures of the deployed MATLAB functions
- Generated package
- MATLAB Runtime installer

The Library Compiler app generates an installer that packages all of the binary artifacts required for distributing a Java package. The installer is located in the `for_redistribution` folder of the compiler project.

6

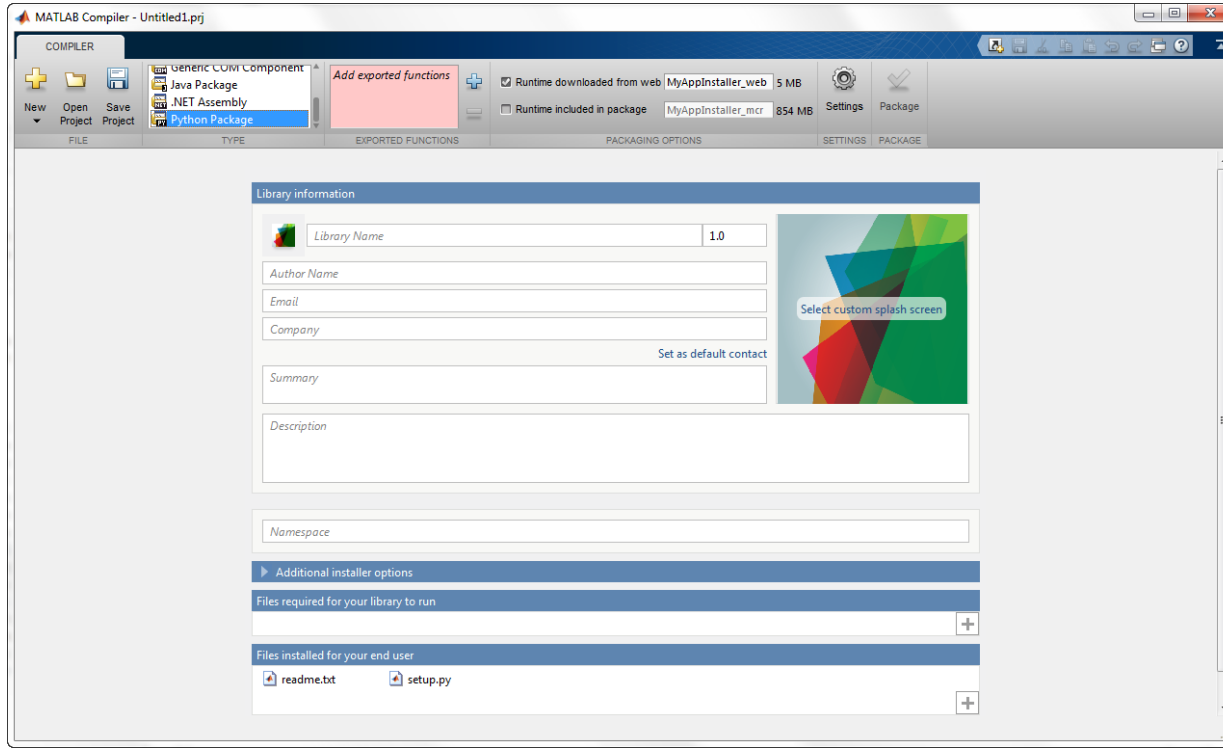
Compile a Python Package

Compile Python Packages with Library Compiler App

Note: MATLAB Compiler SDK cannot compile MATLAB code that uses the MATLAB Python® interface.

To compile MATLAB code into a Python package:

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



Note: You can also start the Library Compiler app using the `libraryCompiler` function.

2 In the **Application Type** section of the toolstrip, select **Python Package**.

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

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

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

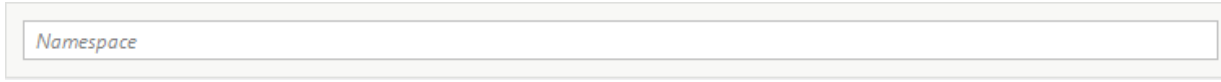
Note: If the **Exported Functions** section of the toolstrip is collapsed, expand it by clicking the down arrow.

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

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

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

4 Verify that the function defined in the selected files is properly mapped into a *namespace*.



5 In the **Packaging Options** section of the toolstrip, specify how the installer will deliver the MATLAB Runtime with the package.

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

You can select one or both of the following options:

- **Runtime downloaded from web** — Generates an installer that downloads the MATLAB Runtime installer from the web.

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

Note: Selecting both options creates two installers.

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

6 Specify the name of any generated installers.

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

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

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

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

All the provided information is displayed as the installer runs.

For more information, see "Customize the Installer" on page 9-2.

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

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

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

For more information, see “Manage Required Files in Compiler Project” on page 9-6.

9 In the **Files installed for your end user** section of the app, verify that any additional non-MATLAB files you want installed with the application are listed.

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

This section automatically lists:

- Generated package
- Python setup script
- Readme file

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

For more information, see “Specify Files to Install with Application” on page 9-8.

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

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

12 Verify that the generated output contains:

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

Compile Python Packages from Command Line

In this section...

Note: MATLAB Compiler SDK cannot compile MATLAB code that uses the MATLAB Python interface.

You can compile Python packages from both the MATLAB command line and the system terminal command line:

- `deploytool` invokes one of the compiler apps to execute a presaved compiler project
- `mcc` invokes the command line compiler

Execute Compiler Projects with deploytool

The `deploytool` command has two flags to invoke one of the compiler apps without opening a window.

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

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

Compile a Python Package with mcc

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

To invoke the compiler to generate a Python package use the `-W python:`*`namespace.packageName`* flag with `mcc`. This flag creates a Python package named *`packageName`* with methods for each of the provided MATLAB functions.

For compiling Python packages, you can also use the following options.

Compiler Python Options

Option	Description
`-a` *`filePath`*	Add any files on the path to the generated binary.
`-d` *`outFolder`*	Specify the folder into which the results of compilation are written.

Distribute Python Packages to Application Developers

Distribute the following to the application developer integrating the package:

- Function signatures of the deployed MATLAB functions
- Generated package
- Generated `setup.py`
- MATLAB Runtime installer

The Library Compiler app generates an installer that packages all the binary artifacts required for distributing a Python package. The installer is located in the `for_redistribution` folder of the compiler project.

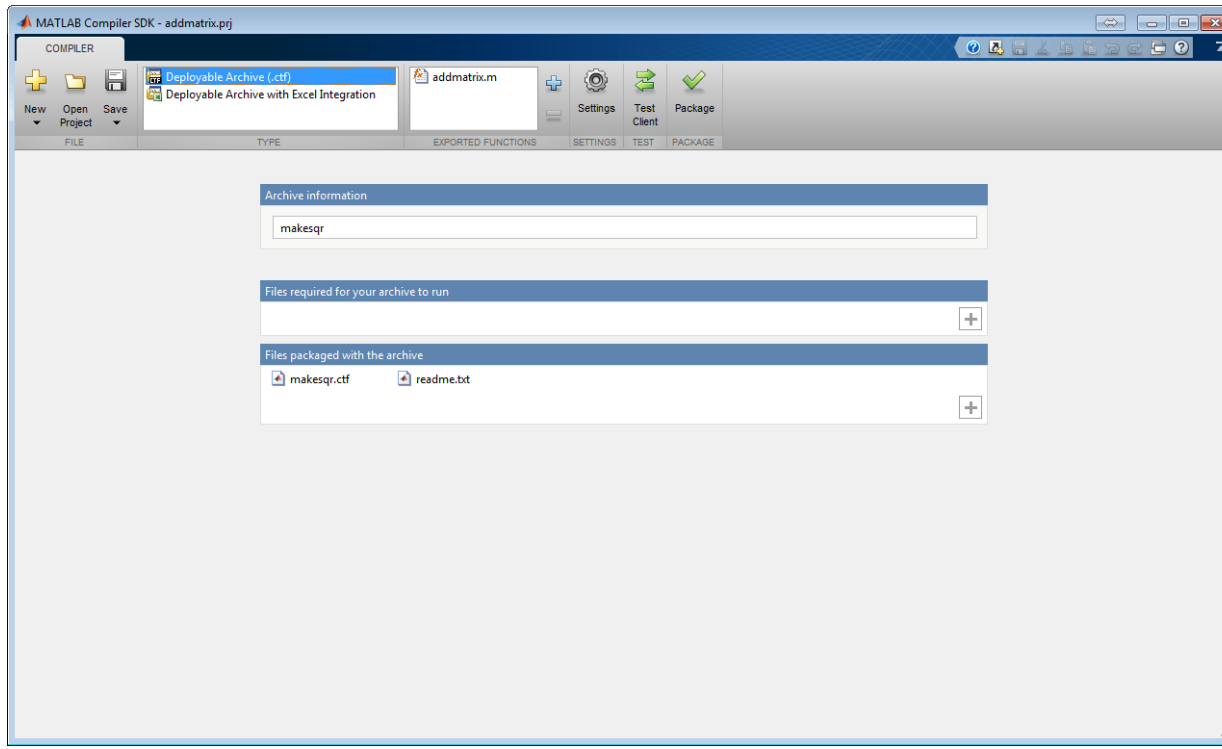
7

Compile a Deployable Archive for MATLAB Production Server

Compile Deployable Archives with Production Server Compiler App

To compile MATLAB code into a deployable archive:

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



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

Note: You can also launch the Production Server Compiler app using the `productionServerCompiler` function.

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

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

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

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

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

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

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

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

4 In the **Archive Information** section of the app, specify the name of the archive.

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

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

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

For more information see “Manage Required Files in Compiler Project” on page 9-6.

6 In the **Files packaged with your archive** section of the app, verify that any additional non-MATLAB files you want packaged with the archive are listed.

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

This section automatically lists:

- Generated deployable archive
- Readme file

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

For more information see “Specify Files to Install with Application” on page 9-8.

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

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

9 Verify that the generated output contains:

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

Compile Deployable Archives from Command Line

In this section...

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

- `deploytool` invokes one of the compiler apps to execute a presaved compiler project
- `mcc` invokes the command line compiler

Execute Compiler Projects with deploytool

The `deploytool` command has two flags to invoke one of the compiler apps without opening a window.

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

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

Compile a Deployable Archive with mcc

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

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

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

Compiler Options

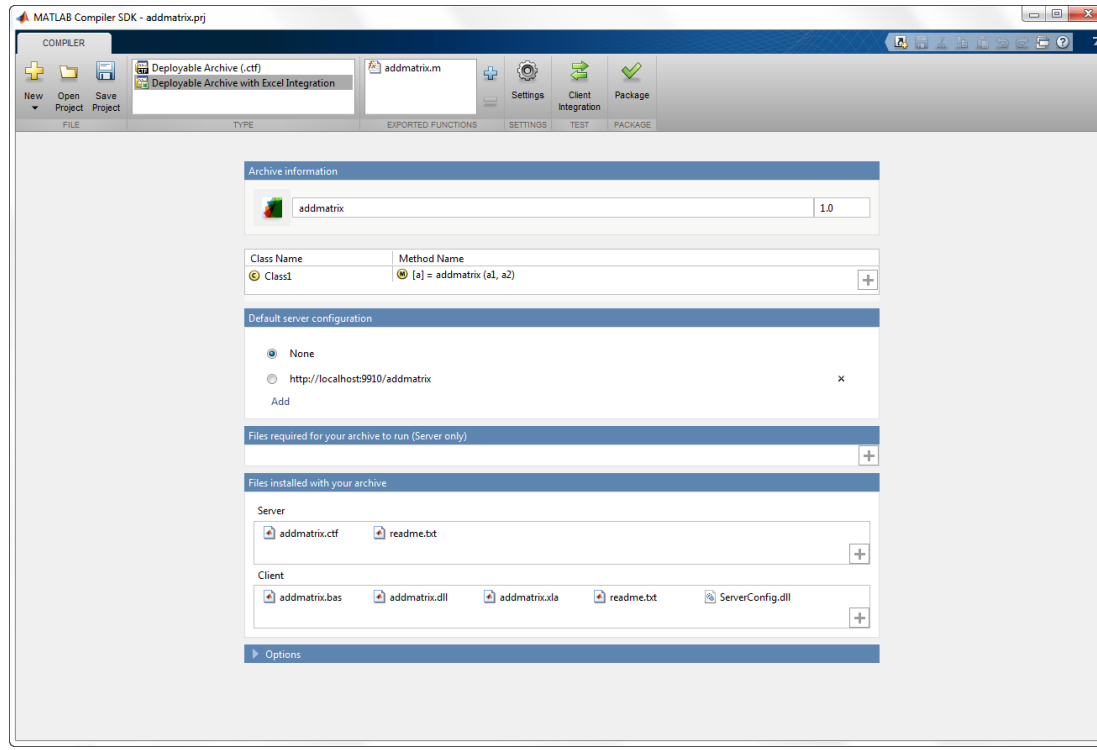
Option	Description
`-a filePath`	Add any files on the path to the generated binary.
`-d outFolder`	Specify the folder into which the results of compilation are written.
`class{className:mfilename...}`	Specify that an additional class is generated that includes methods for the listed MATLAB files.

Build Excel Add-In and Deployable Archive

Note: Excel add-in can be packaged using 64 bit Windows and can be deployed on either 32 or 64 bit Excel.

To create an Excel add-In that integrates with MATLAB Production Server:

1. Ensure that the setting **Trust access to the VBA project object model** is selected in the Excel Trust Center.
2. Open the Production Server Compiler app.
   a. On the toolstrip, select the **Apps** tab.
   b. Click the arrow at the far right of the tab to open the apps gallery.
   c. Click **Production Server Compiler** to open the project window.



3. In the **Application Type** section of the toolstrip, select **Deployable Archive with Excel Integration** from the list.
4. Specify the MATLAB functions you want to deploy.

   a. In the **Exported Functions** section of the toolstrip, click the plus button.
   b. In the file explorer that opens, locate and select the desired files.
   c. Click **Open** to select the files and close the file explorer.

   The selected files are added to the list of files and a minus button appears under the plus button.

   Note: Functions that return a variable number of outputs are not supported by add-ins that use code running on a MATLAB Production Server instance.
5. Inspect the **Archive Information** section of the app.

   The first text field is the name of the archive. The name of the archive determines the names of the generated artifacts and the URL used to connect to the server.
6. Inspect the class mapping table to ensure that all desired functions are being compiled.
7. If you need to change the marshaling rules for a function, select **Data Conversion Properties** from the function name's context menu.

   For more information, see "Data Marshaling Rules".
8. Optionally configure the default server configuration packaged with the installer.

   The server configuration defines the connection to the MATLAB Production Server instance running the MATLAB code.

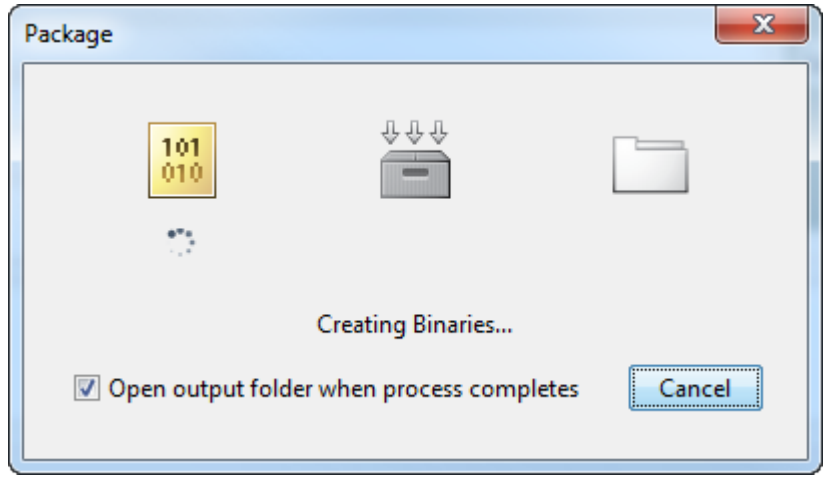
   a. Search the **Default Server Configuration** table for the URL to package with the installer.
   b. If it is in the table, select it.
   c. If not, click **Add** to add it to the table.
9. Inspect the **Files required for your archive to run** and **Files installed with your archive** sections of the app.

   These sections of the app list all of the files that are packaged with the compiled code.

Files required for your archive to run lists the files on which your function is dependent. They are packaged into the deployable archive and are only for the server. See “Manage Required Files in Compiler Project” on page 9-6.

Files installed with your archive includes sections for both the client and the server. The files listed are generated by the compiler and should be delivered to the person installing the application.

10 Click **Package** to generate the add-in and the deployable archive.



11 Select the **Open output folder when process completes** check box to display the generated output.

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

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

13 Verify the contents of the generated output:

- `for_redistribution` — A `client` folder containing the generated installer and a `server` folder containing a `.zip` file
- `for_testing` — A `client` folder containing the raw files generated for the add-in and a `server` folder containing the raw files generated for the deployable archive

- `for_redistribution_files_only` — A `client` folder containing only the files needed to redistribute the add-in and a `server` folder containing only the files needed to redistribute the deployable archive
- `PackagingLog.txt` — A log file generated by the compiler

Compile a COM Component

Compile COM Components with Library Compiler App

See “Create a .NET Assembly” on page 4-2 and select the Generic COM Component as the compilation target.

Compile COM Components from Command Line

A MATLAB class cannot be directly compiled into a COM object. You can, however, use a user-generated class inside a MATLAB file and build a COM object from that file. You can use the MATLAB command-line interface instead of the Library Compiler app to create COM objects. Do this by issuing the `mcc` command with options. If you use `mcc`, you do not create a project.

The following table provides an overview of some `mcc` options related to components, along with syntax and examples of their usage.

Using the Command Line to Create COM Components

Action to Perform	mcc Option to Use	Description
Create component that has one class.	`-W com`	The `W` option with `com` as the type controls the generation of wrapper files, which you can use to support components.
	Syntax `mcc -W 'com:<component_name>[,<class_name>[,<major>.<minor>]]'` An unspecified `<class_name>` defaults to `<component_name>`, and an unspecified version number defaults to the latest version built or 1.0, if there is no previous version.	
	Example `mcc -W 'com:mycomponent,myclass,1.0' -T link:lib foo.m bar.m` The example creates a COM component called `mycomponent`, which contains a single COM class named `myclass` with methods `foo` and `bar`, and a version of 1.0.	
Add additional classes to a COM component.	Not needed	A separate COM named `<class_name>` is created for each class argument that is passed. Following the `<class_name>` parameter is a comma-separated list of source files that are encapsulated as methods for the class.

Action to Perform	mcc Option to Use	Description
	Syntax `class{<class_name>:[file, [file,...]]}`	
	Example `mcc -B 'ccom:mycomponent,myclass,1.0' foo.m bar.m class{myclass2:foo2.m, bar2.m}` The example creates a COM component named `mycomponent` with two classes: `myclass` has methods `foo` and `bar`, and `myclass2` has methods `foo2` and `bar2`. The version is version 1.0.	
Simplify the command-line input for components.	`-B ccom:`	Uses the bundle.
	Syntax `mcc -B '<bundle>'[:<a1>,<a2>,...,<an>]`	
	Example `mcc -B 'ccom:mycomponent,myclass,1.0' foo.m bar.m`	
Control how each COM class uses the MATLAB Runtime.	`-S`	By default, a new MATLAB Runtime instance is created for each instance of each COM class in the component. Use `-S` to change the default. This option tells the compiler to create a single MATLAB Runtime at the time when the first COM class is instantiated. This MATLAB Runtime is reused and shared among all subsequent class instances, resulting in more efficient memory usage and eliminating the MATLAB Runtime startup cost in each subsequent class instantiation. When using `-S`, note that all class instances share a single MATLAB workspace and share global variables in the MATLAB files used to build the component. Therefore, properties of a COM class behave as static properties instead of instance-wise properties.

Action to Perform	mcc Option to Use	Description
		Note: The default behavior dictates that a new MATLAB Runtime be created for each instance of a class, so when the class is destroyed, the MATLAB Runtime is destroyed as well. If you want to retain the state of global variables (such as those allocated for drawing figures, for instance), use the `-S` option.
	Example `mcc -S -B 'ccom:mycomponent,myclass,1.0' foo.m bar.m` The example creates a COM component called `mycomponent` containing a single COM class named `myclass` with methods `foo` and `bar`, and a version of 1.0. When multiple instances of this class are instantiated in an application, only one MATLAB Runtime is initialized, and it is shared by each instance.	
Create subfolders needed for deployment and copy associated files to them.	`-d`	The `\src` and `\distrib` subfolders are needed to package components.
	Syntax `-d` *`foldername`*	

Distribute COM Components to Application Developers

Distribute the following to the application developer integrating the component:

- Function signatures of the deployed MATLAB functions
- Generated COM component
- `mwcomutil.dll`
- MATLAB Runtime installer

The Library Compiler app generates an installer that packages all of the binary artifacts required for distributing a COM component. The installer is located in the `for_redistribution` folder of the compiler project.

9

Customizing a Compiler Project

Customize the Installer

In this section...

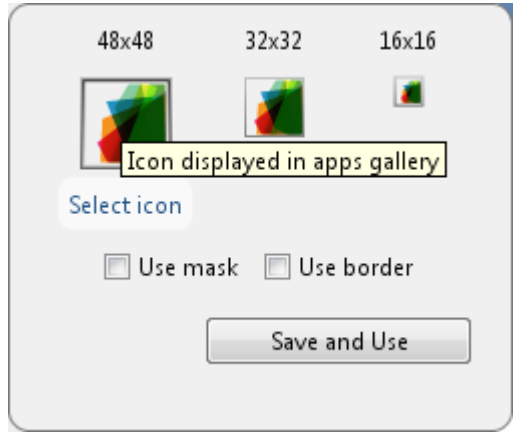
Change the Application Icon

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

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

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

   A window previewing the icon opens.



2. Click **Select icon**.
3. Using the file explorer, locate the graphic file to use as the application icon.
4. Select the graphic file.

5 Click **OK** to return to the icon preview.
6 Select **Use mask** to fill any blank spaces around the icon with white.
7 Select **Use border** to add a border around the icon.
8 Click **Save and Use** to return to the main window.

Add Application Information

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

- Name

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

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

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

  Determines 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
- Description

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

Change the Splash Screen

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

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

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

Change the Installation Path

Default Installation Paths lists the default path the installer will use when installing the compiled binaries onto a target system.

Default Installation Paths

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

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

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

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

  Custom Installation Roots

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

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

Change the Logo

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

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

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

Edit the Installation Notes

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

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

Manage Required Files in Compiler Project

In this section...

Dependency Analysis

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

Using the Compiler Apps

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

To add files:

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

To remove files:

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

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

Using mcc

If you are using `mcc` to compile your MATLAB code, the compiler does not display a list of required files before running. Instead, it compiles all of the required files that are

discovered by the dependency analysis function and adds them to the generated binary file.

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

Specify Files to Install with Application

The compiler apps package files to install along with the ones it generates. By default the installer includes a readme file with instructions on installing the MATLAB Runtine 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:

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

Jar files are added to the application classpath in the same ways as if the user called `javaaddpath`.

To remove files from the list:

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

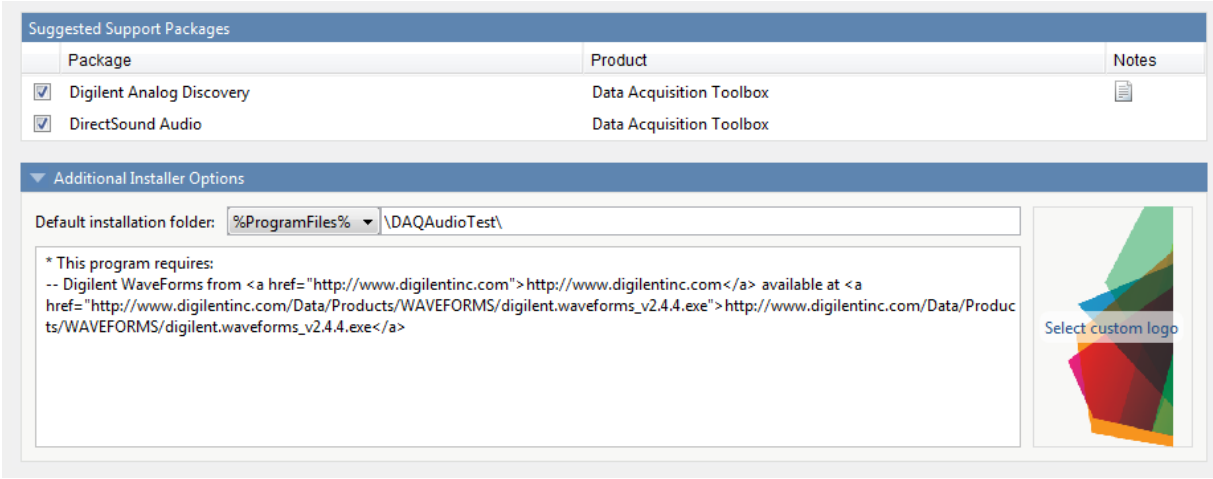
Caution: Removing the binary targets from the list results in an installer that does not install the intended functionality.

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

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\daqauc
```

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.

10

Advanced Uses of the Command Line Compiler

Simplify Compilation Using Macros

In this section...

Macros

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

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

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

Working With Macros

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

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

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

-m Macro

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

Changing Macros

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

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

Specifying Default Macros

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

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

```
mcc foo.m
```

to execute as though it were:

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

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

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

to behave as though the command were:

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

Invoke MATLAB Build Options

In this section...

Specify Full Path Names to Build MATLAB Code

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

1. Breaks the full name into the corresponding path name and file names (`<path>` and `<file>`).
2. Replaces the full path name in the argument list with "`-I <path> <file>`".

Specifying Full Path Names

For example:

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

would be treated as

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

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

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

would be equivalent to

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

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

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

Using Bundles to Build MATLAB Code

Bundles provide a convenient way to group sets of compiler options and recall them as needed. The syntax of the bundle option is:

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

where bundle is either a predefined string such as `cpplib` or `csharedlib` or the name of a file that contains a set of `mcc` command-line options, arguments, filenames, and/or other `-B` options.

A bundle can include replacement parameters for compiler options that accept names and version numbers. For example, the bundle for C shared libraries, `csharedlib`, consists of:

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

To invoke the compiler to produce the C shared library `mysharedlib` use:

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

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

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

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

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

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

Available Bundle Files

Bundle File	Creates	Contents
`cpplib`	C++ library	`-W cpplib:`*`library_name`*`-T link:lib`
`csharedlib`	C library	`-W lib:`*`library_name`*`-T link:lib`
`ccom`	COM component	`-W com:`*`component_name`*`,`*`className`*`,`*`version`*` -T link:lib`
`cexcel`	Excel Add-in	`-W excel:`*`addin_name`*`,`*`className`*`,`*`version`*` -T link:lib`
`cjava`	Java package	`-W java:`*`packageName`*`,`*`className`*
`dotnet`	.NET assembly	`-W dotnet:`*`assembly_name`*`,`*`className`*`,`*`framework_version`*`,`*`securit`*`T link:lib`

MATLAB Runtime Component Cache and Deployable Archive Embedding

In this section...

Deployable archive data is automatically embedded directly in compiled components and extracted to a temporary folder.

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

These variables allow you to specify the following:

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

Use the following environment variables to change these settings.

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

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

You can override this automatic embedding and extraction behavior by compiling with the "Overriding Default Behavior" on page 10-8 option.

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

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

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

Overriding Default Behavior

To extract the deployable archive in a manner prior to R2008b, alongside the compiled .NET assembly, compile using the `mcc`'s `-C` option.

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

For More Information

For more information about the deployable archive, see "Deployable Archive" (MATLAB Compiler).

Work with the MATLAB Runtime

MATLAB Runtime Startup Options

Retrieve MATLAB Runtime Startup Options

Use these functions to return data about the MATLAB Runtime state when working with shared libraries.

Function and Signature	When to Use	Return Value
`bool mclIsMCRInitialized()`	Use `mclIsMCRInitialized()` to determine whether or not the MATLAB Runtime has been properly initialized.	Boolean (`true` or `false`). Returns `true` if MATLAB Runtime is already initialized, else returns `false`.
`bool mclIsJVMEnabled()`	Use `mclIsJVMEnabled()` to determine if the MATLAB Runtime was launched with an instance of a Java Virtual Machine (JVM™).	Boolean (`true` or `false`). Returns `true` if MATLAB Runtime is launched with a JVM instance, else returns `false`.
`const char* mclGetLogFileName()`	Use `mclGetLogFileName()` to retrieve the name of the log file used by the MATLAB Runtime.	Character string representing log file name used by the MATLAB Runtime.
`bool mclIsNoDisplaySet()`	Use `mclIsNoDisplaySet()` to determine if `-nodisplay` option is enabled.	Boolean (`true` or `false`). Returns `true` if `-nodisplay` is enabled, else returns `false`. **Note:** `false` is always returned on Windows systems since the `-nodisplay` option is not supported on Windows systems. When running on Mac, if `-nodisplay` is used as one of the options included in `mclInitializeApplication`, then the call to `mclInitializeApplication` must occur before calling `mclRunMain`.

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

Retrieve Information About MATLAB Runtime Startup Options

```
const char* options[4];
   options[0] = "-logfile";
   options[1] = "logfile.txt";
   options[2] = "-nojvm";
   options[3] = "-nodisplay";
   if( !mclInitializeApplication(options,4) )
   {
       fprintf(stderr,
               "Could not initialize the application.\n");
       return -1;
   }
   printf("MCR initialized : %d\n", mclIsMCRInitialized());
   printf("JVM initialized : %d\n", mclIsJVMEnabled());
   printf("Logfile name : %s\n", mclGetLogFileName());
   printf("nodisplay set : %d\n", mclIsNoDisplaySet());
   fflush(stdout);
```

Using the 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 between 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 `mxArray`s, consisting of a mapping from string keys to `mxArray` values. Reasons for doing this include, but are not limited to the following:

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

The API consists of:

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

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.

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

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`.

Display the 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.

12

Limitations and Restrictions

MATLAB Compiler SDK Limitations

In this section...

Compiling MATLAB and Toolboxes

MATLAB Compiler SDK supports the full MATLAB language and almost all toolboxes based on MATLAB. However, some limited MATLAB and toolbox functionality is not licensed for compilation.

- Most of the prebuilt graphical user interfaces included in MATLAB and its companion toolboxes will not compile.
- Functionality that cannot be called directly from the command line will not compile.
- Some toolboxes, such as Symbolic Math Toolbox™, will not compile.

Compiled applications can only run on operating systems that run MATLAB. Also, since the MATLAB Runtime is approximately the same size as MATLAB, applications built with MATLAB Compiler SDK need specific storage memory and RAM to operate. For the most up-to-date information about system requirements, go to the MathWorks Web site.

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

Note: See "MATLAB Functions That Cannot Be Compiled" on page 12-10 for a list of functions that cannot be compiled.

Fixing Callback Problems: Missing Functions

When MATLAB Compiler SDK creates a standalone application, it compiles the MATLAB file(s) you specify on the command line and, in addition, it compiles any other MATLAB files that your MATLAB file(s) calls. MATLAB Compiler SDK uses a dependency analysis, which determines all the functions on which the supplied MATLAB files, MEX-files, and P-files depend.

Note: If the MATLAB file associated with a p-file is unavailable, the dependency analysis will not be able to discover the p-file's dependencies.

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

- In a callback string
- In a 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 get loaded by compiled applications. Use the `mcc -a` argument or the `%#function` pragma to identify `.mat` file classes or functions that should be supported by the `load` command.

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

Symptom

Your application runs, but an interactive user interface element, such as a push button, does not work. The compiled application issues this error message:

```
An error occurred in the callback: change_colormap
```

```
The error message caught was    : Reference to unknown function
                change_colormap from FEVAL in stand-alone mode.
```

Workaround

There are several ways to eliminate this error:

- Using the `%#function pragma` and specifying callbacks as 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's dependency analysis and instructs it to explicitly include the functions listed in the `%#function` pragmas.

For example, the call to the `change_colormap` function in the sample application, `my_test`, illustrates this problem. To make sure MATLAB Compiler SDK processes the `change_colormap` MATLAB file, list the function name in the `%#function` pragma.

```
function my_test()
% Graphics library callback test application

%#function change_colormap

peaks;

p_btn = uicontrol(gcf,...
                  'Style', 'pushbutton',...
                  'Position',[10 10 133 25 ],...
                  'String', 'Make Black & White',...
                  'CallBack','change_colormap');
```

Specifying Callbacks with Function Handles

To specify the callbacks with function handles, use the same code as in the example above and replace the last line with

```
'CallBack',@change_colormap);
```

For more information on specifying the value of a callback, see the MATLAB Programming Fundamentals documentation.

Using the -a Option

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

Finding Missing Functions in a MATLAB File

To find functions in your application that may need to be listed in a `%#function` pragma, search your MATLAB file source code for text 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 will find all the `Callback` properties defined by graphics objects, such as `uicontrol` and `uimenu`. In addition, this will find the properties of figures and axes that end in `Fcn`, such as `CloseRequestFcn`, that also support callbacks.

Suppressing Warnings on the UNIX System

Several warnings may appear when you run a standalone application on the UNIX system. This section describes how to suppress these warnings.

To suppress the `libjvm.so` warning, make sure you set the dynamic library path properly for your platform. See “MATLAB Runtime Path Settings for Run-Time Deployment”.

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 will get a run-time error.

Cannot Create the Output File

If you receive the error

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

there are several possible causes to consider:

- Lack of write permission for the folder where MATLAB Compiler SDK is attempting to write the file (most likely the current working folder).
- Lack of free disk space in the folder where MATLAB Compiler SDK 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 SDK from overwriting it with a new version.

No MATLAB File Help for Compiled Functions

If you create a MATLAB file with self-documenting online help by entering text on one or more contiguous comment lines beginning with the second line of the file and then compile it, the results of the command

```
help filename
```

will be unintelligible.

Note: Due to performance reasons, MATLAB file comments are stripped out before MATLAB Runtime encryption.

No MATLAB Runtime Versioning on Mac OS X

The feature that allows you to install multiple versions of the MATLAB Runtime on the same machine is currently not supported on Mac OS X. When you receive a new version of MATLAB , you must recompile and redeploy all of your applications and components. Also, when you install a new MATLAB Runtime onto a target machine, you must delete the old version of the MATLAB Runtime and install the new one. You can only have one version of the MATLAB Runtime on the target machine.

Older Neural Networks Not Deployable with MATLAB Compiler

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

For example, deploying with Neural Network Toolbox Version 5.0.1 (2006b) and MATLAB Compiler Version 4.5 (R2006b) yields the following errors at run time:

```
??? Error using ==> network.subsasgn
"layers{1}.initFcn" cannot be set to non-existing
 function "initwb".
Error in ==> updatenet at 40
Error in ==> network.loadobj at 10

??? Undefined function or method 'sim' for input
arguments of type 'struct'.
Error in ==> mynetworkapp at 30
```

Restrictions on Calling PRINTDLG with Multiple Arguments in Compiled Mode

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

You will not receive an error when making at call to `printdlg` with multiple arguments. However, when an application containing the multiple-argument call is compiled, the compile will fail with the following error message:

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

Compiling a Function with WHICH Does Not Search Current Working Directory

Using `which`, as in this example:

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

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

Use one of the following solutions as alternatives to using `which`:

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

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

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

  ```
  load myFile.mat
  ```

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

Restrictions on Using C++ SETDATA to Dynamically Resize an MWArray

You cannot use the C++ `SETDATA` function to dynamically resize `MWArray`s.

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

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

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

Licensing Terms and Restrictions on Compiled Applications

Applications you build with a trial MATLAB Compiler SDK license are valid for thirty (30) days only.

Applications you build with a purchased license of MATLAB Compiler SDK have no expiration date.

MATLAB Functions That Cannot Be Compiled

Note: Due to the number of active and ever-changing list of MathWorks products and functions, this is not a complete list of functions that can not be compiled. If you have a question as to whether a specific MathWorks product's function is able to be compiled or not, the definitive source is that product's documentation. 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, for example, the MATLAB `help` function or debug functions, will not work.
- Simulink® functions, in general, will not work.
- Functions that require a command line, for example, the MATLAB `lookfor` function, will not work.
- `clc`, `home`, and `savepath` will not do anything in deployed mode.
- Tools that allow run-time manipulation of figures

Returned values from standalone applications will be 0 for successful completion or a nonzero value otherwise.

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

`mccExcludedFiles.log` lists all the functions and files excluded by `mcc` if they can not be compiled. It is created after each attempted build if there are functions or files that cannot be compiled.

List of Unsupported Functions and Programs

```
add_block
add_line
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`

`mislocked`

`mlock`

```
more
munlock
new_system
open_system
pack
pcode
plotbrowser
plotedit
plottools
profile
profsave
propedit
propertyeditor
publish
rehash
restoredefaultpath
run
segment
set_param
sim
sldebug
type
```

13

Functions

%#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" (MATLAB Compiler).

Introduced before R2006a

componentinfo

Query system registry about COM component created with MATLAB Compiler SDK

Syntax

```
info = componentinfo
info = componentinfo(component_name)
info = componentinfo(component_name, major_revision_number)
info = componentinfo(component_name, major_revision_number,
minor_revision_number)
```

Arguments

component_name	MATLAB character array naming the COM component created by MATLAB Compiler SDK. Names are case sensitive. If the argument is not supplied, information is returned on all installed components.
major_revision_number	Component major revision number. If the argument is not supplied, information is returned on all major revisions.
minor_revision_number	Component minor revision number. Default value is 0.

Description

`info = componentinfo` returns information for all components installed on the system.

`info = componentinfo(`*component_name*`)` returns information for all revisions of *component_name*.

`info = componentinfo(`*component_name*`, `*major_revision_number*`)` returns information for the most recent minor revision corresponding to *major_revision_number* of *component_name*.

`info = componentinfo(component_name, major_revision_number, minor_revision_number)` returns information for the specific major and minor version of `component_name`.

The return value is an array of structures representing all the registry and type information needed to load and use the component.

When you supply a component name, `major_revision_number` and `minor_revision_number` are interpreted as shown next.

Value	Information Returned
> 0	Information on a specific major and minor revision.
0	Information on the most recent revision. When omitted, `minor_revision_number` is assumed to be 0.
< 0	Information on all versions.

This table describes the fields in `componentinfo`.

Registry Information Returned by componentinfo

Field	Description
`Name`	Component name.
`TypeLib`	Component type library.
`LIBID`	Component type library GUID.
`MajorRev`	Major version number .
`MinorRev`	Minor version number.
`FileName`	Type library file name and path. Since all the compiler components have the type library bound into the DLL, this file name is the same as the DLL name and path.
`Interfaces`	An array of structures defining all interface definitions in the type library. Each structure contains two fields: • `Name` - Interface name. • `IID` - Interface GUID.
`CoClasses`	An array of structures defining all COM classes in the component. Each structure contains these fields:

Field	Description

	• Name - Class name.
	• CLSID - GUID of the class.
	• ProgID - Version-dependent program ID.
	• VerIndProgID - Version-independent program ID.
	• InprocServer32 - Full name and path to component DLL.
	• Methods - A structure containing function prototypes of all class methods defined for this interface. This structure contains four fields:
	  • IDL - An array of Interface Description Language function prototypes.
	  • M - An array of MATLAB function prototypes.
	  • C - An array of C-language function prototypes.
	  • VB - An array of VBA function prototypes.
	• Properties - A cell array containing the names of all class properties.
	• Events - A structure containing function prototypes of all events defined for this class. This structure contains four fields:
	  • IDL - An array of Interface Description Language function prototypes.
	  • M - An array of MATLAB function prototypes.
	  • C - An array of C-language function prototypes.
	  • VB - An array of VBA function prototypes.

Examples

Function Call	Returned Information
`Info = componentinfo`	Information for all installed components.
`Info = componentinfo('mycomponent')`	Information for all revisions of `mycomponent`.

Function Call	Returned Information
`Info = componentinfo('mycomponent',1,0)`	Information for revision 1.0 of `mycomponent`.

Tips

Use the `componentinfo` function to get information (such as class name, program ID) to pass on to users of a component that you create.

The `componentinfo` function also provides a record of changes made to the registry on your development machine. This information might be useful for debugging if you run into problems.

Introduced before R2006a

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

Compile and package functions for external deployment

Syntax

```
deploytool
deploytool project_name
deploytool -build project_name
deploytool -package project_name
```

Description

`deploytool` opens a list of the compiler apps.

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

`deploytool -build project_name` runs the appropriate compiler app to build the specified project. The installer is not generated.

`deploytool -package project_name` runs the appropriate compiler app to build and package the specified project. The installer is generated.

Examples

Create a New Compiler Project

Open the compiler to create a new project.

```
deploytool
```

Package an Application using an Existing Project

Open the compiler to build a new application using an existing project.

```
deploytool -package my_magic
```

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.

Introduced in R2006b

figToImStream

Stream figure as byte array encoded in specified format

Syntax

```
output = figToImStream
output = figToImStream (Name,Value)
```

Description

`output = figToImStream` creates a signed byte array with the PNG data for the current figure. The size and position of the printed output depends on the figure's `PaperPosition[mode]` properties.

`output = figToImStream (Name,Value)` creates a byte array with the image data for the specified figure. You can specify the encoding format for the image and if the byte array is signed or unsigned. The size and position of the printed output depends on the figure's `PaperPosition[mode]` properties.

Examples

Convert current figure to a signed PNG formatted byte array

```
surf(peaks)
bytes = figToImStream
```

Convert a specific figure to a bitmap stored in an unsigned byte array

```
f = figure;
surf(peaks);
bytes = figToImStream('figHandle',f,...
                      'imageFormat','bmp',...
                      'outputType','uint8');
```

Input Arguments

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 single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'figHandle', f, 'imageFormat', 'bmp', 'outputType', 'uint8'` specifies the figure `f` is streamed into an unsigned byte array as a bitmap.

`'figHandle'` — Figure to stream

empty character array or string (default) | figure handle

Figure to stream, specified as the comma-separated pair consisting of `'figHandle'` and a figure handle.

`'imageFormat'` — Encoding format

png (default) | jpg | bmp | gif

Encoding format, specified as the comma-separated pair consisting of `'imageFormat'` and one of these values:

- `png` — encode the image using the Portable Network Graphics (PNG) format
- `jpg` — encode the image using the JPEG format
- `bmp` — encode the image as a bitmap
- `gif` — encode the image using the Graphics Interchange Format (GIF)

`'outputType'` — Type of bytes to store the image stream

int8 (default) | uint8

Type of bytes to store the image stream, specified as the comma-separated pair consisting of `'outputType'` and one of these values:

- `int8` — use a signed byte array
- `uint8` — use an unsigned byte array

Output Arguments

`output` — Encoded figure data

byte array

Encoded figure data returned as a byte array.

Introduced in R2009b

getmcruserdata

Retrieve MATLAB array value associated with a given key

Syntax

```
function_value = getmcruserdata(key)
```

Description

The `function_value = getmcruserdata(key)` command is part of the MATLAB Runtime User Data interface API. It returns an empty matrix if no such key exists.

Examples

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

See Also

`setmcruserdata`

Introduced in R2008b

isdeployed

Determine whether code is running in deployed or MATLAB mode

Syntax

```
x = isdeployed
```

Description

`x = isdeployed` returns true (1) when the function is running in deployed mode and false (0) if it is running in a MATLAB session.

If you include this function in an application and compile the application, the function will return true when the application is run in deployed mode. If you run the application containing this function in a MATLAB session, the function will return false.

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`, it will return true. This function will return false when executed within MATLAB as well as in deployed mode. To test for deployed mode execution, use `isdeployed`. This function should be used to guard code in `matlabrc`, or `hgrc` (or any function called within them, for example `startup.m` in the example on this page), from being executed by MATLAB Compiler (`mcc`) or 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 as shown in the example on this page.

Examples

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

See Also

`isdeployed` | `mcc`

libraryCompiler

Build and package functions for use in external applications

Syntax

```
libraryCompiler
libraryCompiler project_name
libraryCompiler -build project_name
libraryCompiler -package 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.

`libraryCompiler -build project_name` runs the Library Compiler app to build the specified project. The installer is not generated.

`libraryCompiler -package project_name` runs the Library Compiler app to build and package the specified project. The installer is generated.

Examples

Create a New Project

Open the Library Compiler app to create a new project.

```
libraryCompiler
```

Package a Function using an Existing Project

Open the Library Compiler app using an existing project.

```
libraryCompiler -package my_magic
```

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.

Introduced in R2013b

mbuild

Compile and link source files against MATLAB generated shared libraries

Syntax

```
mbuild [option1 ... optionN] sourcefile1 [... sourcefileN]
   [objectfile1 ... objectfileN] [libraryfile1 ... libraryfileN]
```

Description

`mbuild` compiles and links customer written C or C++ code against MATLAB generated shared libraries.

Some of these options (`-f`, `-g`, and `-v`) are available on the `mcc` command line and are passed along to `mbuild`. Others can be passed along using the `-M` option to `mcc`. For details on the `-M` option, see the `mcc` reference page.

Supported Source File Types

Supported types of source files are:

- `.c`
- `.cpp`

Arguments to `mbuild` that are not options and do not belong to one of the supported source file types are assumed to be library names, and are passed to the linker.

Options

This table lists the set of `mbuild` options. If no platform is listed, the option is available on both UNIX and Windows.

Option	Description
`@<rspfile>`	(Windows only) Include the contents of the text file `<rspfile>` as command line arguments to `mbuild`.

Option	Description
`-<arch>`	Build an output file for architecture `-<arch>`. To determine the value for `-<arch>`, type `computer('arch')` at the MATLAB command prompt on the target machine. Note: Valid values for `-<arch>` depend on the architecture of the build platform.
`-c`	Compile only. Creates an object file only.
`-D<name>`	Define a symbol name to the C preprocessor. Equivalent to a `#define <name>` directive in the source.
`-D<name>=<value>`	Define a symbol name and value to the C preprocessor. Equivalent to a `#define <name> <value>` directive in the source.
`-f <optionsfile>`	Specify location and name of options file to use. Overrides the `mbuild` default options file search mechanism.
`-g`	Create an executable containing additional symbolic information for use in debugging. This option disables the `mbuild` default behavior of optimizing built object code (see the `-O` option).
`-h[elp]`	Print help for `mbuild`.
`-I<pathname>`	Add `<pathname>` to the list of folders to search for `#include` files.
`-l<name>`	Link with object library. On Windows systems, `<name>` expands to `<name>.lib` or `lib<name>.lib` and on UNIX systems, to `lib<name>.so` or `lib<name>.dylib`. Do not add a space after this switch. **Note:** When linking with a library, it is essential that you first specify the path (with `-I<pathname>`, for example).
`-L<folder>`	Add `<folder>` to the list of folders to search for libraries specified with the `-l` option. On UNIX systems, you must also set the run-time library paths. Do not add a space after this switch.

Option	Description
`-n`	No execute mode. Print out any commands that `mbuild` would otherwise have executed, but do not actually execute any of them.
`-O`	Optimize the object code. Optimization is enabled by default and by including this option on the command line. If the `-g` option appears without the `-O` option, optimization is disabled.
`-outdir <dirname>`	Place all output files in folder `<dirname>`.
`-output <resultname>`	Create an executable named `<resultname>`. An appropriate executable extension is automatically appended. Overrides the `mbuild` default executable naming mechanism.
`-setup`	Interactively specify the C/C++ compiler options file to use as the default for future invocations of `mbuild` by placing it in the user profile folder (returned by the `prefdir` command). When this option is specified, no other command line input is accepted.
`-setup -client mbuild_com`	Interactively specify the COM compiler options file to use as the default for future invocations of `mbuild` by placing it in the user profile folder (returned by the `prefdir` command). When this option is specified, no other command line input is accepted.
`-U<name>`	Remove any initial definition of the C preprocessor symbol `<name>`. (Inverse of the `-D` option.)
`-v`	Verbose mode. Print the values for important internal variables after the options file is processed and all command line arguments are considered. Prints each compile step and final link step fully evaluated.

Option	Description
`<name>=<value>`	Supplement or override an options file variable for variable `<name>`. This option is processed after the options file is processed and all command line arguments are considered. You may need to use the shell's quoting syntax to protect characters such as spaces that have a meaning in the shell syntax. On Windows double quotes are used (e.g., `COMPFLAGS="opt1 opt2"`), and on UNIX single quotes are used (e.g., `CFLAGS='opt1 opt2'`). It is common to use this option to supplement a variable already defined. To do this, refer to the variable by prepending a `$` (e.g., `COMPFLAGS="$COMPFLAGS opt2"` on Windows or `CFLAGS='$CFLAGS opt2'` on UNIX shell).

Examples

To change the default C/C++ compiler for use with MATLAB Compiler SDK, use

```
mbuild -setup
```

To compile and link an external C program `foo.c` against `libfoo`, use

```
mbuild foo.c -L. -lfoo (on UNIX)
mbuild foo.c libfoo.lib (on Windows)
```

This assumes both `foo.c` and the library generated above are in the current working folder.

Introduced before R2006a

mcc

Compile MATLAB functions for deployment

Syntax

```
mcc options mfilename1,...,mfilenameN

mcc -l options mfilename1,...,mfilenameN
mcc -c options mfilename1,...,mfilenameN
mcc -W cpplib:library_name -T link:lib options
mfilename1,...,mfilenameN

mcc -W
dotnet:assembly_name,className,framework_version,security,remote_type
-T link:lib options mfilename1,...,mfilenameN
mcc -W
dotnet:assembly_name,className,framework_version,security,remote_type
-T link:lib options class{className:mfilename1,...,mfilenameN}

mcc -W java:packageName,className options mfilename1,...,mfilenameN
mcc -W java:packageName,className options class{className:
mfilename1,...,mfilenameN}

mcc -W python:namespace.packageName -T link:lib options
mfilename1,...,mfilenameN

mcc -W CTF:archive_name options mfilename1,...,mfilenameN

mcc -W mpsxl:addin_name,className,version input_marshaling_flags
output_marshaling_flags -T link:lib options
mfilename1,...,mfilenameN
```

Description

`mcc options mfilename1,...,mfilenameN` compiles the functions as specified by the options.

The options used depend on the intended results of the compilation. For information on compiling:

- standalone applications, Excel add-ins, or Hadoop® jobs see `mcc` for MATLAB Compiler

`mcc -l options mfilename1,...,mfilenameN` compiles the listed functions into a C shared library and generates C wrapper code for integration with other applications.

This syntax is equivalent to `-W lib:`*`libname`* `-T link:lib`.

`mcc -c options mfilename1,...,mfilenameN` generates C wrapper code for the listed functions.

This is equivalent to `-W lib:`*`libname`* `-T codegen`.

`mcc -W cpplib:`*`library_name`* `-T link:lib options mfilename1,...,mfilenameN` compiles the listed functions into a C++ shared library and generates C++ wrapper code for integration with other applications.

- *`library_name`* — Specifies the name of the shared library.

`mcc -W dotnet:`*`assembly_name`*`,`*`className`*`,`*`framework_version`*`,`*`security`*`,`*`remote_type`* `-T link:lib options mfilename1,...,mfilenameN` creates a .NET assembly with a single class from the specified files.

- *`assembly_name`* — Specifies the name of the assembly preceded by its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *`className`* — Specifies the name of the .NET class to be created.
- *`framework_version`* — Specifies the version of the Microsoft .NET Framework you want to use to compile the assembly. Specify either:
  - `0.0` — Use the latest supported version on the target machine.
  - *`version_major.version_minor`* — Use a specific version of the framework.

  Features are often version-specific. Consult the documentation for the feature you are implementing to get the Microsoft .NET Framework version requirements.
- *`security`* — Specifies whether the assembly to be created is a private assembly or a shared assembly.

  - To create a private assembly, specify `Private`.
  - To create a shared assembly, specify the full path to the encryption key file used to sign the assembly.
- *`remote_type`* — Specifies the remoting type of the assembly. Values are `remote` and `local`.

`mcc -W dotnet:`*`assembly_name`*`,`*`className`*`,`*`framework_version`*`,`*`security`*`,`*`remote_type`*` -T link:lib options class{`*`className`*`:mfilename1,...,mfilenameN}` creates a .NET assembly with multiple classes from the specified files.

- *`assembly_name`* — Specifies the name of the assembly and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *`className`* — Specifies the name of the .NET class to be created.

  Note: You can include multiple class specifiers.
- *`framework_version`* — Specifies the version of the Microsoft .NET Framework you want to use to compile the assembly. Specify either:
  - `0.0` — Use the latest supported version on the target machine.
  - *`version_major.version_minor`* — Use a specific version of the framework.

  Features are often version-specific. Consult the documentation for the feature you are implementing to get the Microsoft .NET Framework version requirements.
- *`security`* — Specifies whether the assembly to be created is a private assembly or a shared assembly.
  - To create a private assembly, specify `Private`.
  - To create a shared assembly, specify the full path to the encryption key file used to sign the assembly.
- *`remote_type`* — Specifies the remoting type of the assembly. Values are `remote` and `local`.

`mcc -W java:`*`packageName`*`,`*`className`*` options mfilename1,...,mfilenameN` creates a Java package from the specified files.

- *`packageName`* — Specifies the name of the Java package and its namespace, which is a period-separated list, such as `companyname.groupname.component`.

- *`className`* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the last item in *`packageName`*.

`mcc -W java:`*`packageName`*`,`*`className`* `options class{`*`className`*`:mfilename1,...,mfilenameN}` creates a Java package with multiple classes from the specified files.

- *`packageName`* — Specifies the name of the Java package and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *`className`* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the last item in *`packageName`*.

Note: You can include multiple class specifiers.

`mcc -W python:`*`namespace`*`.`*`packageName`* `-T link:lib options mfilename1,...,mfilenameN` creates a Python package from the specified files.

- *`namespace`* — Specifies the optional namespace for the package, which is a period-separated list, such as `companyname.groupname.component`
- *`packageName`* — Specifies the name of the Python package.

`mcc -W CTF:`*`archive_name`* `options mfilename1,...,mfilenameN` instructs the compiler to create a deployable archive for use with a MATLAB Production Server instance.

`mcc -W mpsxl:`*`addin_name`*`,`*`className`*`,`*`version input_marshaling_flags output_marshaling_flags`* `-T link:lib options mfilename1,...,mfilenameN` creates a Microsoft Excel add-in that is integrated with MATLAB Production Server from the specified files.

- *`addin_name`* — Specifies the name of the add-in and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *`className`* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the *`addin_name`* as the default.
- *`version`* — Specifies the version of the add-in specified as *`major`*`.`*`minor`*.
  - *`major`* — Specifies the major version number. If you do not specify a version number, `mcc` uses the latest version.
  - *`minor`* — Specifies the minor version number. If you do not specify a version number, `mcc` uses the latest version.

- *`input_marshaling_flags`* — Specifies options for how data is marshaled between Microsoft Excel and MATLAB.
  - `-replaceBlankWithNaN` — Specifies that a blank in Microsoft Excel is mashaled into NaN in MATLAB. If you do not specify this flag, blanks are marshaled into 0.
  - `-convertDateToString` — Specifies that dates in Microsoft Excel are marshaled into MATLAB character vectors. If you do not specify this flag, dates are marshaled into MATLAB doubles.
- *`output_marshaling_flags`* — Specifies options for how data is marshaled between MATLAB and Microsoft Excel.
  - `-replaceNaNWithZero` — Specifies that NaN in MATLAB is marshaled into a 0 in Microsoft Excel. If you do not specify this flag, NaN is marshaled into `#QNAN` in Visual Basic®.
  - `-convertNumericToDate` — Specifies that MATLAB numeric values are marshaled into Microsoft Excel dates. If you do not specify this flag, Microsoft Excel does not receive dates as output.

Examples

Compile a C++ shared library

```
cpplib:myMagic -T link:lib magic.m
```

Compile a Java package containing multiple classes

```
mcc -W java:myMatrix,add class{add:add.m} class{sub:minus.m}
```

Compile a Python package

```
mcc -W python:myMagic -T link:lib magic.m
```

Input Arguments

`mfilename1,...,mfilenameN` — Files to be compiled

list of filenames

One, or more, files to be compiled, specified as a comma-separated list of filenames.

`options` — Options for customizing the output

-a | -b | -B | -C | -d | -f | -g | -G | -I | -K | -m | -M | -N | -o | -p | -R | -S | -T | -u | -v | -w | -W | -Y

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

- `-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 include files, the absolute path for the DLL and header files changes. The files are placed in the `.\exe_mcr\` folder when the archive is expanded. The file is not placed in the local folder. This folder is created from the deployable archive the first time the application is executed. The `isdeployed` function is provided to help you accommodate this difference in deployed mode.

The `-a` switch also creates a `.auth` file for authorization purposes. It ensures that the executable looks for the DLL- and H-files in the `exe_mcr\exe` folder.

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

Note: 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” on page 10-5.

- `-C`

  Do not embed the deployable archive in binaries.

- `-d`

  Place output in a specified folder. Use

  ```
  -d outFolder
  ```

  to direct the generated files to *`outFolder`*.

- `-f`

  Override the default options file with the specified options file. Use

  ```
  -f filename
  ```

  to specify `filename` as the options file when calling `mbuild`. This option lets you use different ANSI compilers for different invocations of 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 adds a folder to the beginning of the list of paths to search. For example,

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

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

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` not to 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 provides a useful mechanism for defining compile-time options, e.g., `-M "-Dmacro=value"`.

  Note: Multiple `-M` options do not accumulate; only the rightmost `-M` option is used.

- `-N`

  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`

  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 (e.g., `.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`

  Provides MATLAB Runtime options. This option is only relevant when building standalone applications using MATLAB Compiler. The syntax is as follows:

  ```
  -R option
  ```

Option	Description	Target
`-logfile,`	Specify a log file name.	MATLAB Compiler

Option	Description	Target
`-nodispla`	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
`-complete`	Customizable user message displayed when initialization is complete.	MATLAB Compiler Standalone Applications

Caution: When running on Mac OS X, 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` will still compile without errors and generate the results. But the `-R` option doesn't apply to these libraries and won't 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. This 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. • Using `mcc`, pass the `-S` flag.
Java package	

- `-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 links object files into a standalone application.
`link:lib`	Same as `compile:lib`, and also links 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.

- `-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 all of the possible warning that `mcc` can generate.
`-w enable`	Enable complete warnings.
`-w disable[:<string>]`	Disable specific warnings associated with `<string>`. See "Warning Messages" (MATLAB Compiler) for a list of the `<string>` values. Omit the optional `<string>` to apply the `disable` action to all warnings.
`-w enable[:<string>]`	Enable specific warnings associated with `<string>`. See "Warning Messages" (MATLAB Compiler) for a list of the `<string>` values. Omit the optional `<string>` to apply the `enable` action to all warnings.
`-w error[:<string>]`	Treat specific warnings associated with `<string>` as an error. Omit the optional `<string>` to apply the `error` action to all warnings.

Syntax	Description
`-w off[:<string>] [<filename>]`	Turn warnings off for specific error messages defined by <*string*>. You can also narrow scope by specifying warnings be turned off when generated by specific <*filename*>s.
`-w on[:<string>] [<filename>]`	Turn warnings on for specific error messages defined by <*string*>. You can also narrow scope by specifying warnings be turned on when generated by specific <*filename*>s.

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

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

```
if isdeployed
    warning off
end
```

To turn warnings on for deployed applications, you write:

```
if isdeployed
    warning on
end
```

- `-W`

  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.

- `-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
```

See Also

See Also

`mbuild`

Introduced before R2006a

mcrinstaller

Display version and location information for MATLAB Runtime installer corresponding to current platform

Syntax

```
[INSTALLER_PATH, MAJOR, MINOR, PLATFORM, LIST] = mcrinstaller;
```

Description

Displays information about available MATLAB Runtime installers using the format: `[INSTALLER_PATH, MAJOR, MINOR, PLATFORM, LIST] = mcrinstaller;` where:

- *`INSTALLER_PATH`* is the full path to the installer for the current platform.
- *`MAJOR`* is the major version number of the installer.
- *`MINOR`* is the minor version number of the installer.
- *`PLATFORM`* is the name of the current platform (returned by `COMPUTER(`*`arch`*`)`).
- *`LIST`* is a cell array of character vectors containing the full paths to MATLAB Runtime installers for other platforms. This list is non-empty only in a multi-platform MATLAB installation.

Note: 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 the MATLAB Runtime”for more information about the MATLAB Runtime installer.

Examples

Find MATLAB Runtime Installer Locations

Display locations of MATLAB Runtime installers for platform. This example shows output for a `win64` system.

```
mcrinstaller

The WIN64 MATLAB Runtime Installer, version 9.0.1, is:
    C:\Program Files\MATLAB\R2016a\toolbox\compiler\deploy\win64\MCRInstaller.exe

MATLAB Runtime installers for other platforms are located in:
    C:\Program Files\MATLAB\R2016a\toolbox\compiler\deploy\<ARCH>
    <ARCH> is the value of COMPUTER('arch') on the target machine.

Full list of available MATLAB Runtime installers:
   C:\Program Files\MATLAB\R2016a\toolbox\compiler\deploy\win64\MCRInstaller.exe

For more information, read your local MATLAB Runtime Installer Help.
Or see the online documentation at MathWorks' web site. (Page
                                                 may load slowly.)

ans =

C:\Program Files\MATLAB\R2016a\toolbox\compiler\deploy\win64\MCRInstaller.exe
```

Introduced in R2009a

mcrversion

Determine version of installed MATLAB Runtime

Syntax

```
[major, minor] = mcrversion;
```

Description

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] = mcrversion;` Major and minor are returned as integers.

If the version number ever increases to three or more digits, call `mcrversion` with more outputs, as follows:

```
[major, minor, point] = mcrversion;
```
At this time, all outputs past "minor" are returned as zeros.

Typing only `mcrversion` will return the major version number only.

Examples

```
mcrversion
ans =
     7
```

Introduced in R2008a

productionServerCompiler

Test, build and package functions for use with MATLAB Production Server

Syntax

```
productionServerCompiler
productionServerCompiler project_name
productionServerCompiler -build project_name
productionServerCompiler -package project_name
```

Description

`productionServerCompiler` opens the Production Server Compiler app for the creation of a new compiler project.

`productionServerCompiler project_name` opens the appropriate compiler app with the project preloaded.

`productionServerCompiler -build project_name` runs the appropriate compiler app to build the specified project. The installer is not generated.

`productionServerCompiler -package project_name` runs the appropriate compiler app to build and package the specified project. The installer is generated.

Examples

Create a New Production Server Project

Open the Production Server Compiler app to create a new project.

```
productionServerCompiler
```

Package a Deployable Archive using an Existing Project

Open the appropriate compiler app to package an existing project file.

```
productionServerCompiler -package my_magic
```

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.

Introduced in R2014a

setmcruserdata

Associate MATLAB data value with a key

Syntax

```
function setmcruserdata(key, value)
```

Description

The `function setmcruserdata(key, value)` command is part of the MATLAB Runtime User Data interface API.

Examples

In C++:

```
mxArray *key = mxCreateString("ParallelProfile");
mxArray *value = mxCreateString("\usr\userdir\config.settings");
if (!setmcruserdata(key, value))
{
    fprintf(stderr,
            “Could not set MCR user data: \n %s ”,
            mclGetLastErrorMessage());
    return -1;
}
```

In C:

```
mxArray *key = mxCreateString("ParallelProfile");
mxArray *value = mxCreateString("\usr\userdir\config.settings");
if (!mlfSetmcruserdata(key, value))
{
    fprintf(stderr,
            “Could not set MCR user data: \n %s ”,
            mclGetLastErrorMessage());
    return -1;
}
```

See Also

`getmcruserdata`

Introduced in R2008a

webfigure

Export a figure to a deployed application

Syntax

```
exportFigure = webfigure(h)
```

Description

`exportFigure = webfigure(h)` exports the figure identified by the handle h to a deployed application for use as a WebFigure. How the deployed application manages the exported figure is specific to the target language. The WebFigure feature is support by Java and .NET.

Examples

Export a Figure

```
function returnFigure = getWebFigure()
        h = figure;
        set(h, 'Visible', 'off');
        surf(peaks);
        set(h, 'Color', [.8,.9,1]);
        returnFigure = webfigure(h);
        close(h);
```

Input Arguments

h — handle of the figure to be exported

figure handle

Specify the figure to be exported as a figure handle.

Output Arguments

`exportFigure` — exported figure

structure

Specifies the exported figure in a language specific structure.

Introduced in R2008a

14

Apps

Library Compiler

Package MATLAB programs for deployment as shared libraries and components

Description

The **Library Compiler** app packages MATLAB functions to include MATLAB functionality in applications written in other languages.

Open the Library Compiler App

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

Examples

- "Create a C/C++ Shared Library with MATLAB Code"
- "Package Excel Add-In with Library Compiler App" (MATLAB Compiler)
- "Create a .NET Application with MATLAB Code"
- "Package a Deployable COM Component"
- "Create a Java Application with MATLAB Code"
- "Compile Python Packages with Library Compiler App" on page 6-2

Parameters

`type` — type of library generated

C Shared Library | C++ Shared Library | Excel Add-in | Generic COM Component | Java Package | .NET Assembly | Python Package

Type of library to generate.

`exported functions` — functions to package

list of character vectors

Functions to package as a list of character vectors.

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

Method for installing the MATLAB Runtime as a radio button. Including MATLAB Runtime in the package significantly increases the size of the package.

`files required for your library to run` — files that must be included with library
list of files

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

`files installed for your end user` — optional files installed with library
list of files

Optional files installed with library 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 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 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.

Library Information

`library name` — name of the installed library
character vector

Name of the installed library as a character vector.

The default value is the name of the first function listed in the **Exported Functions** field of the app.

`version` — version of the generated library
character vector

Version of the generated library as a character vector.

`splash screen` — image displayed on installer
image

Image displayed on installer as an image.

`author name` — name of the library author
character vector

Name of the library author as a character vector.

`e-mail` — e-mail address used to contact library support
character vector

E-mail address used to contact library support as a character vector.

`summary` — brief description of library
character vector

Brief description of library as a character vector.

`description` — detailed description of library
character vector

Detailed description of library as a character vector.

Additional Installer Options

`default installation folder` — folder where artifacts are installed
character vector

Folder where artifacts are installed as a character vector.

`installation notes` — notes about additional requirements for using artifacts
character vector

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

Programmatic Use

```
libraryCompiler
```

See Also

Topics

“Create a C/C++ Shared Library with MATLAB Code”
“Package Excel Add-In with Library Compiler App” (MATLAB Compiler)
“Create a .NET Application with MATLAB Code”
“Package a Deployable COM Component”
“Create a Java Application with MATLAB Code”
“Compile Python Packages with Library Compiler App” on page 6-2

Production Server Compiler

Package MATLAB programs for deployment to MATLAB Production Server

Description

The **Production Server Compiler** app tests the integration of client code with MATLAB functions. It also packages MATLAB functions into archives for deployment to MATLAB Production Server.

Open the Production Server Compiler App

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

Examples

- "Create a Deployable Archive for MATLAB Production Server"
- "Build Excel Add-In and Deployable Archive" on page 7-7

Parameters

`type` — type of archive generated

Deployable Archive | Deployable Archive with Excel Integration

Type of archive to generate as a character array.

`exported functions` — functions to package

list of character arrays

Functions to package as a list of character arrays.

`archive information` — name of the archive

character array

Name of the archive as a character array.

`files required for your archive to run` — files that must be included with archive
list of files

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

`files packaged with the archive` — optional files installed with archive
list of files

Optional files installed with archive as a list of files.

Settings

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

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

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

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

`end user files` — folder where files for building a custom installer are stored
character array

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

`packaged installers` — folder where generated installers are stored
character array

Folder where generated installers are stored as a character array.

Programmatic Use

```
productionServerCompiler
```

See Also

Topics

"Create a Deployable Archive for MATLAB Production Server"
"Build Excel Add-In and Deployable Archive" on page 7-7

Introduced in R2013b