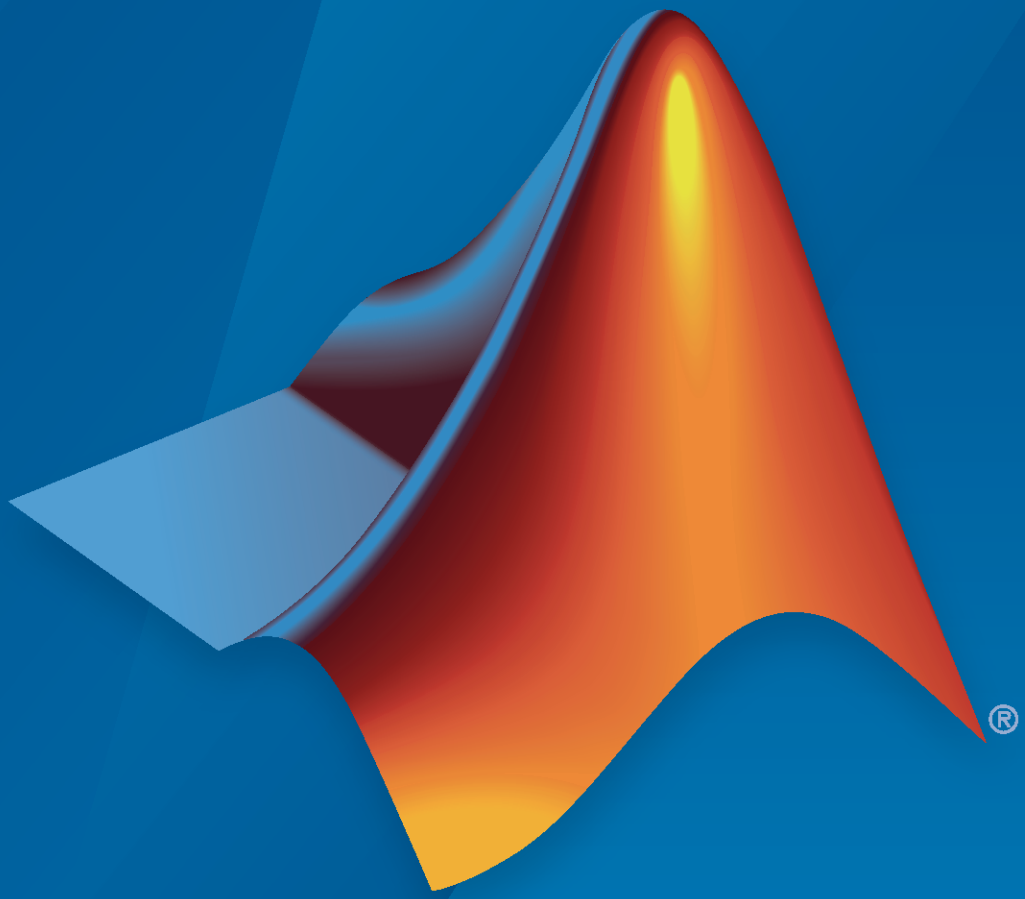
MATLAB® Compiler SDK™

MATLAB® Code Deployment Guide



MATLAB®

R2023a

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

MATLAB® Compiler 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)
September 2017	Online only	Revised for Version 6.4 (Release R2017b)
March 2018	Online only	Revised for Version 6.5 (Release R2018a)
September 2018	Online only	Revised for Version 6.6 (Release R2018b)
March 2019	Online only	Revised for Version 6.6.1 (Release R2019a)
September 2019	Online only	Revised for Version 6.7 (Release R2019b)
March 2020	Online only	Revised for Version 6.8 (Release R2020a)
September 2020	Online only	Revised for Version 6.9 (Release R2020b)
March 2021	Online only	Revised for Version 6.10 (Release R2021a)
September 2021	Online only	Revised for Version 6.11 (Release R2021b)
March 2022	Online only	Revised for Version 7.0 (Release R2022a)
September 2022	Online only	Revised for Version 7.1 (Release R2022b)
March 2023	Online only	Revised for Version 7.2 (Release R2023a)

Contents

1 Overview

2 Write Deployable MATLAB Code

3 Package a C/C++ Shared Library

4 Package a .NET Assembly

5 Package a Java Application

6 Package a Python Application

7 Compile a Deployable Archive for MATLAB Production Server

8 Package 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

13 Functions

14 Apps

Overview

How Does MATLAB Deploy Functions?

To deploy MATLAB functions, the compiler performs these tasks:

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

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

   For more information about dependency analysis, see “Dependency Analysis Using MATLAB Compiler” on page 1-3.
2. Validates MEX-files. In particular, `mexFunction` entry points are verified.

   For more details about including MEX-files in packaged applications, see “Access Files in Packaged Applications” on page 2-9.
3. Creates a deployable archive from the input files and their dependencies.

   For more details about deployable archives, see “About the 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 corresponding third-party compiler.

Dependency Analysis Using MATLAB Compiler

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 package time. Dependency analysis also processes `include`/`exclude` files on each pass.

Tip To improve package time performance and lessen application size, prune the path with the `mcc` command’s `-N` and `-p` flags. You can also specify **Files required for your application to run** in the compiler app or use the `AdditionalFiles` option in a `compiler.build` function.

Function Dependency

The dependency analyzer searches for executable content such as:

- MATLAB files
- P-files

  Note If the MATLAB file corresponding to the p-file is not available, the dependency analysis cannot determine the p-file’s dependencies.
- `.fig` files
- MEX-files

Data File Dependency

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

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

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

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

Exclude Files From Package

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

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

See Also

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

More About

- Application Compiler

Distribute Files to Application Developers

In this section...

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

The `deploytool` apps generate an installer that packages all of the binary artifacts required for distributing a compiled component. The installer is located in the `for_redistribution` folder of the compiler project.

You can also generate an installer using the `compiler.package.installer` function.

If you do not create an installer, distribute the set of files required to integrate the component according to the component type. In order to run the application, the target machine must have access to MATLAB Runtime that matches the version of MATLAB used to compile the component, at the same update level or newer.

Distribute COM Components

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

- Function signatures of the deployed MATLAB functions
- *`packageName`*`.dll` — generated COM component
- `_install.bat` — generated script that registers the component (to register manually, see “Register COM Component”)

Distribute C/C++ Shared Libraries

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

- Function signatures of the deployed MATLAB functions
- *`libraryName`*`.lib/.dylib/.so` — generated library
- *`libraryName`*`.h` — generated header file

Distribute Java Packages

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

- Function signatures of the deployed MATLAB functions
- *`packageName`*`.jar` — generated Java package

Distribute .NET Assemblies

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

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

Distribute Python Packages

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

- Function signatures of the deployed MATLAB functions
- `_init_.py` — initialization script for the Python package
- `setup.py` — generated Python installer

See Also

“Files Generated After Packaging MATLAB Functions”

About the Deployable Archive

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

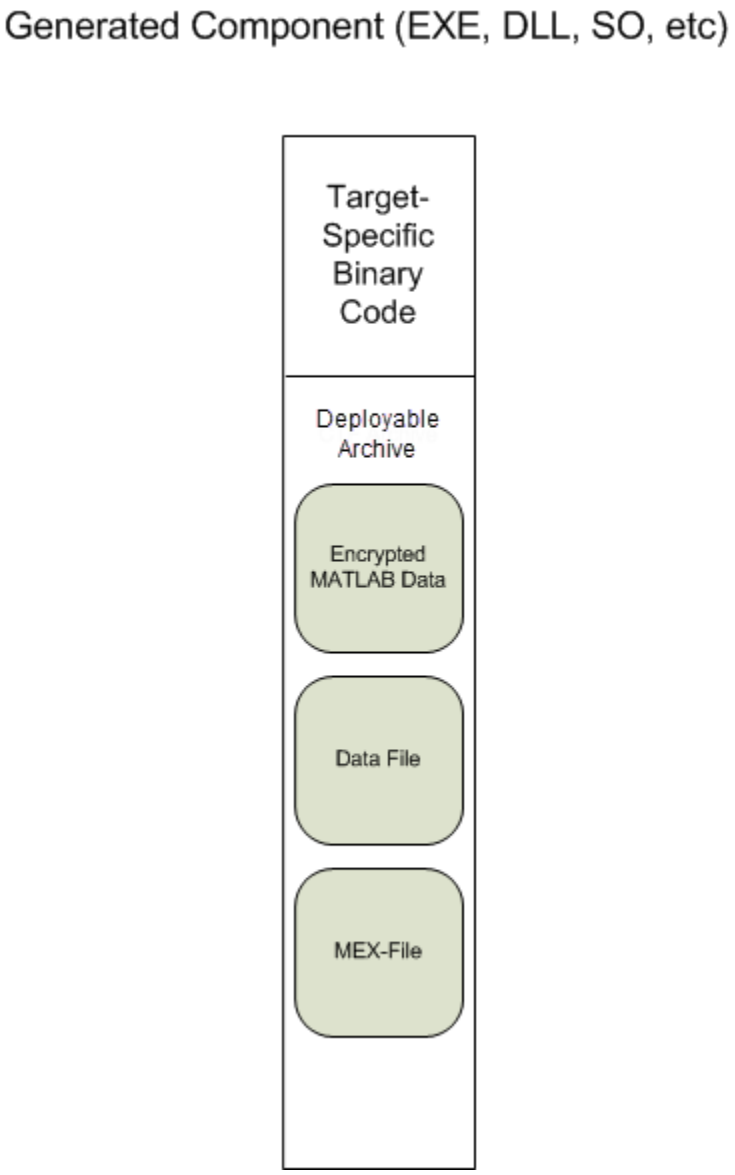
All MATLAB files (`.m` and `.p` files) included in the deployable archive are encrypted using the Advanced Encryption Standard (AES) cryptosystem. By default, the names of files and the directory structure are not obscured and other file types, including MEX-files, MAT-files, FIG-files, Java JAR or class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

Starting in R2022b, you can obscure the names of files and the directory structure, and also encrypt other file types (such as MAT, FIG, MEX, and so on) using the `-s` option for `mcc`. At run time, the encrypted files remain encrypted on the disk but are decrypted in memory to their original form before compiling. You can use this option with the `-j` option to create P-coded files from MATLAB code files before they are compiled.

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	“Define Embedding and Extraction Options for Deployable Java Archive”
MATLAB Compiler Excel® integration	“MATLAB Runtime Component Cache and Deployable Archive Embedding”



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...

In order to package and deploy MATLAB code, your code must follow certain guidelines to avoid errors. You can implement applications to access deployed MATLAB code using APIs generated from MATLAB functions.

Accepted File Types for Packaging

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

Target Application	Valid File Types	Invalid File Types
Standalone Application	MATLAB MEX files, MATLAB scripts, MATLAB functions, and MATLAB class files. These files must have a single entry point.	Protected function files (`.p` files), Java functions, COM or .NET components, and data files.
Library Compiler	MATLAB MEX files, MATLAB functions, and MATLAB class files. These files must have a single entry point.	MATLAB scripts, protected function files (`.p` files), Java functions, COM or .NET components, and data files.
MATLAB Production Server	MATLAB MEX files and MATLAB functions. These files must have a single entry point.	MATLAB scripts, MATLAB class files, protected function files (`.p` files), Java functions, COM or .NET components, and data files. MATLAB class files can be dependent files.

Packaged Applications Do Not Process MATLAB Files at Run Time

The compiler secures your code against unauthorized changes. Deployable MATLAB files are suspended or frozen at the time of compilation. This does not mean that you cannot deploy a flexible application—it means that *you must design your application with flexibility in mind*. If you want the end user to be able to choose between two different methods, for example, both methods must be available in the deployable archive.

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

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

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

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

1. Run the code once in MATLAB to obtain your generated function.
2. Package the MATLAB code and include the generated function.

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

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

Get Proper Licenses for Toolbox Functionality You Want to Deploy

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

Use isdeployed Functions To Execute Deployment-Specific Code Paths

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

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

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

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

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

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

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

Gradually Refactor Applications that Depend on Non-Deployable Functions

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

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

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

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

Do Not Create or Use Nonconstant Static State Variables

Avoid using the following:

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

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

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

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

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

See Also

`isdeployed` | `ismcc`

More About

- MATLAB Compiler support for MATLAB and toolboxes

- “Limitations”

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 also *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 that you call `loadlibrary` with the following syntax.

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

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

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

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

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

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

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

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

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

See Also

`loadlibrary`

Related Examples

- "Call Functions in C Library Loaded with loadlibrary"

Access Files in Packaged Applications

In this section...

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

For more information about deployable archives, see “About the Deployable Archive” on page 1-7.

Include Files in Deployable Archive

MATLAB Compiler uses a dependency analysis function to determine the list of necessary files to include in the generated package. For details, see “Dependency Analysis Using MATLAB Compiler” on page 1-3.

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

Alternatively, you can add files to the **Files installed for your end user** section in a `deploytool` app so that they appear in the same directory as the executable after installation.

Explicitly Include MATLAB Data Files Using %#function Pragma

The compiler excludes MATLAB data files (MAT-files) from dependency analysis by default. If you want the compiler to explicitly inspect data within a MAT-file, specify the `%#function` pragma when writing your MATLAB code.

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

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

Include MEX-Files, DLLs, or Shared Libraries

When you compile MATLAB functions containing MEX-files, ensure that the dependency analyzer can find them. 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.
- 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.

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

Access Files from Deployed Functions

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

Use which function

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

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

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

Specify File Location in ctfroot

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

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

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

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

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

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

Example Processing MATLAB Data for Deployed Applications

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

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

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

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

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

3. Examine `ex_loadsave.m`.

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

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

ex_loadsave.m

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

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

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

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

```
disp('A= ');
disp(data1);

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

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

%==== multiply two data matrices together ==============
result = data1*data2;
disp('A * B = ');
disp(result);

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

4 Create a cell array that lists the data files.

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

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

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

6 Run the compiled application.

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

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

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

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

A * B =
   1.0e+03 *

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

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

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

See Also

`ctfroot` | `which`

Related Examples

- “About the Deployable Archive” on page 1-7
- “Dependency Analysis Using MATLAB Compiler” on page 1-3

Share MATLAB Runtime Instances

In a shared MATLAB Runtime instance or Singleton runtime, you create an instance of the MATLAB Runtime that can be shared among all subsequent class instances within a component.

Advantages and Disadvantages of Using a Singleton

In most cases, a singleton MATLAB Runtime will provide many more advantages than disadvantages.

Singleton Advantages

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

Singleton Disadvantages

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

Package 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 https://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.
Installing on `Maci64`	Install X code from installation DVD.

Create C/C++ Shared Libraries from Command Line

In this section...

You can package C/C++ applications at the MATLAB prompt or your system prompt using either of these commands.

- `deploytool` invokes the Application Compiler app to execute a saved compiler project.
- `mcc` invokes the MATLAB Compiler to create a deployable application at the command prompt.

Execute Compiler Projects with deploytool

The `deploytool` command has two flags that invoke one of the compiler apps to package an already existing project without opening a window.

- `-build` *`project_name`* — Invoke the correct compiler app to build the project but 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.

Package a Shared Library with mcc

The `mcc` command invokes MATLAB Compiler to create a deployable application at the command prompt and provides fine-level control while packaging the application. It does not 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/C++ shared library that you can integrate into applications developed in C or C++.

Use the following `mcc` options to package a shared library.

Option	Description
`-W lib:`*`libname`* `-T link:lib`	Generate a C shared library. Equivalent to using `-l`. The `-W lib:<libname>` option tells the compiler to generate a function wrapper for a shared library and call it `libname`. The `-T link:lib` option specifies the target output as a shared library. Note the directory where the product puts the shared library because you will need it later on.

Option	Description
`-W cpplib:`*`libname`* `-T link:lib`	Generate a C++ shared library. The `-W lib:<libname>` option tells the compiler to generate a function wrapper for a shared library and call it `libname`. The `-T link:lib` option specifies the target output as a shared library. Note the directory where the product puts the shared library because you will need it later on.
`-a` *`filePath`*	Add the file or files on the path to the generated binary.
`-d` *`outFolder`*	Specify the folder for the packaged applications.

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

- Function signatures of the deployed MATLAB functions
- *`libraryName`*`.lib`/`.dylib`/`.so` — generated library
- *`libraryName`*`.h` — generated header file

Differences Between Compiler Apps and Command Line

You perform the same functions using the compiler apps, a `compiler.build` function, or the `mcc` command-line interface. The interactive menus and dialog boxes 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 packaging it using `mcc`.

If you know the commands for the type of application you want to deploy and do not require an installer, it is faster to execute either `compiler.build` or `mcc` than go through the compiler app workflow.

Compiler app advantages include:

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

See Also

`mcc` | `deploytool`

More About

- “Create a C Shared Library with MATLAB Code”

4

Package a .NET Assembly

Package .NET Assemblies from Command Line

In this section...

You can package .NET assemblies at the MATLAB prompt or your system prompt using either of these commands.

- `deploytool` invokes the Library Compiler app to create a compiler project.
- `mcc` invokes the MATLAB Compiler to create a deployable application at the command prompt.

Execute Compiler Projects with deploytool

The `deploytool` command has two flags that invoke one of the compiler apps to package an already existing project without opening a window.

- `-build` *`project_name`* — Invoke the correct compiler app to build the project but 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.

Create .NET Assemblies with mcc

The `mcc` command invokes MATLAB Compiler to create a .NET assembly at the command prompt and provides fine-level control while packaging the application. It does not package the results in an installer.

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 package 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 `NET.isNETSupported` to check for a supported version of Microsoft .NET framework.

Differences Between Compiler Apps and Command Line

You perform the same functions using the compiler apps, a `compiler.build` function, or the `mcc` command-line interface. The interactive menus and dialog boxes 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 packaging it using `mcc`.

If you know the commands for the type of application you want to deploy and do not require an installer, it is faster to execute either `compiler.build` or `mcc` than go through the compiler app workflow.

Compiler app advantages include:

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

See Also

`mcc` | `deploytool`

More About

- “Generate .NET Assembly and Build .NET Application”

Package a Java Application

Package Java Applications from Command Line

In this section...

You can package Java applications at the MATLAB prompt or your system prompt using either of these commands.

- `deploytool` invokes the Application Compiler app to execute a saved compiler project.
- `mcc` invokes the MATLAB Compiler to create a deployable application at the command prompt.

Execute Compiler Projects with deploytool

The `deploytool` command has two flags that invoke one of the compiler apps to package an already existing project without opening a window.

- `-build` *`project_name`* — Invoke the correct compiler app to build the project but 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.

Package a Java Application with mcc

The `mcc` command invokes MATLAB Compiler to create a deployable application at the command prompt and provides fine-level control while packaging the application. It does not package the results in an installer.

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

Package Java applications using the following 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 packaging 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.

Differences Between Compiler Apps and Command Line

You perform the same functions using the compiler apps, a `compiler.build` function, or the `mcc` command-line interface. The interactive menus and dialog boxes 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 packaging it using `mcc`.

If you know the commands for the type of application you want to deploy and do not require an installer, it is faster to execute either `compiler.build` or `mcc` than go through the compiler app workflow.

Compiler app advantages include:

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

See Also

`mcc` | `deploytool`

More About

- “Generate Java Package and Build Java Application”

Map Functions to Java Classes

In this section...

Map Functions to Java Classes using Library Compiler App

The Library Compiler app provides 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 application into which the generated classes are placed. By default, the name of the first listed MATLAB file is used as the application name. You can change the application name to fit the naming conventions used by your organization.

The table used to match functions to classes is below the application 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 Application

To add a class to a Java application:

1. Click **Add Class**.
2. Rename the class as described in “Rename a Java Class” on page 5-4.
3. Add one or more methods to the class as described in “Add Method to Java Class” on page 5-5.

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 Class from Java Application

To delete a class from a Java application:

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

Add Method to 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 Method from 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 applications, 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 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 application `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 application `myPackage` that contains two classes: `MyClass` and `MyOtherClass`. `MyClass` has two methods: `fun1` and `fun2`. `MyOtherClass` has one method `fun3`.

6

Package a Python Application

Package Python Applications from Command Line

In this section...

Note MATLAB Compiler SDK cannot package MATLAB code that uses the MATLAB Python interface with in-process execution mode.

You can package Python applications at the MATLAB prompt or your system prompt using either of these commands.

- `deploytool` invokes the Application Compiler app to execute a saved compiler project.
- `mcc` invokes the MATLAB Compiler to create a deployable application at the command prompt.

Execute Compiler Projects with deploytool

The `deploytool` command has two flags that invoke one of the compiler apps to package an already existing project without opening a window.

- `-build` *`project_name`* — Invoke the correct compiler app to build the project but 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.

Package a Python Application with mcc

The `mcc` command invokes MATLAB Compiler to create a deployable application at the command prompt and provides fine-level control while packaging the application. It does not package the results in an installer.

To invoke the compiler to generate a Python application, 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 packaging Python applications, you can also use the following 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 packaging are written.

Differences Between Compiler Apps and Command Line

You perform the same functions using the compiler apps, a `compiler.build` function, or the `mcc` command-line interface. The interactive menus and dialog boxes 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 packaging it using `mcc`.

If you know the commands for the type of application you want to deploy and do not require an installer, it is faster to execute either `compiler.build` or `mcc` than go through the compiler app workflow.

Compiler app advantages include:

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

See Also

`mcc` | `deploytool`

More About

- “Generate a Python Package and Build a Python Application”

Compile a Deployable Archive for MATLAB Production Server

Package Deployable Archives with Production Server Compiler App

Supported platform: Windows, Linux, Mac

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

Create Function In MATLAB

In MATLAB, examine the MATLAB program that you want packaged.

For this example, write a function `addmatrix.m` as follows.

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

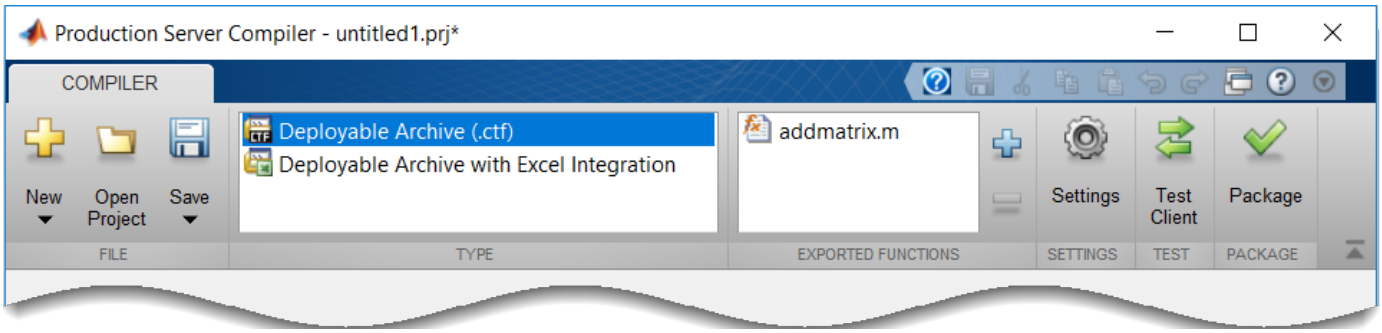
At the MATLAB command prompt, enter `addmatrix([1 4 7; 2 5 8; 3 6 9], [1 4 7; 2 5 8; 3 6 9])`.

The output is:

```
ans =
     2     8    14
     4    10    16
     6    12    18
```

Create Deployable Archive with Production Server Compiler App

1. On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Production Server Compiler**. In the **Production Server Compiler** project window, click **Deployable Archive (.ctf)**.



   Alternately, you can open the **Production Server Compiler** app by entering `productionServerCompiler` at the MATLAB prompt.
2. In the **MATLAB Compiler SDK** project window, specify the main file of the MATLAB application that you want to deploy.
   1. In the **Exported Functions** section of the toolstrip, click  .
   2. In the **Add Files** window, browse to the example folder, and select the function you want to package. Click **Open**.

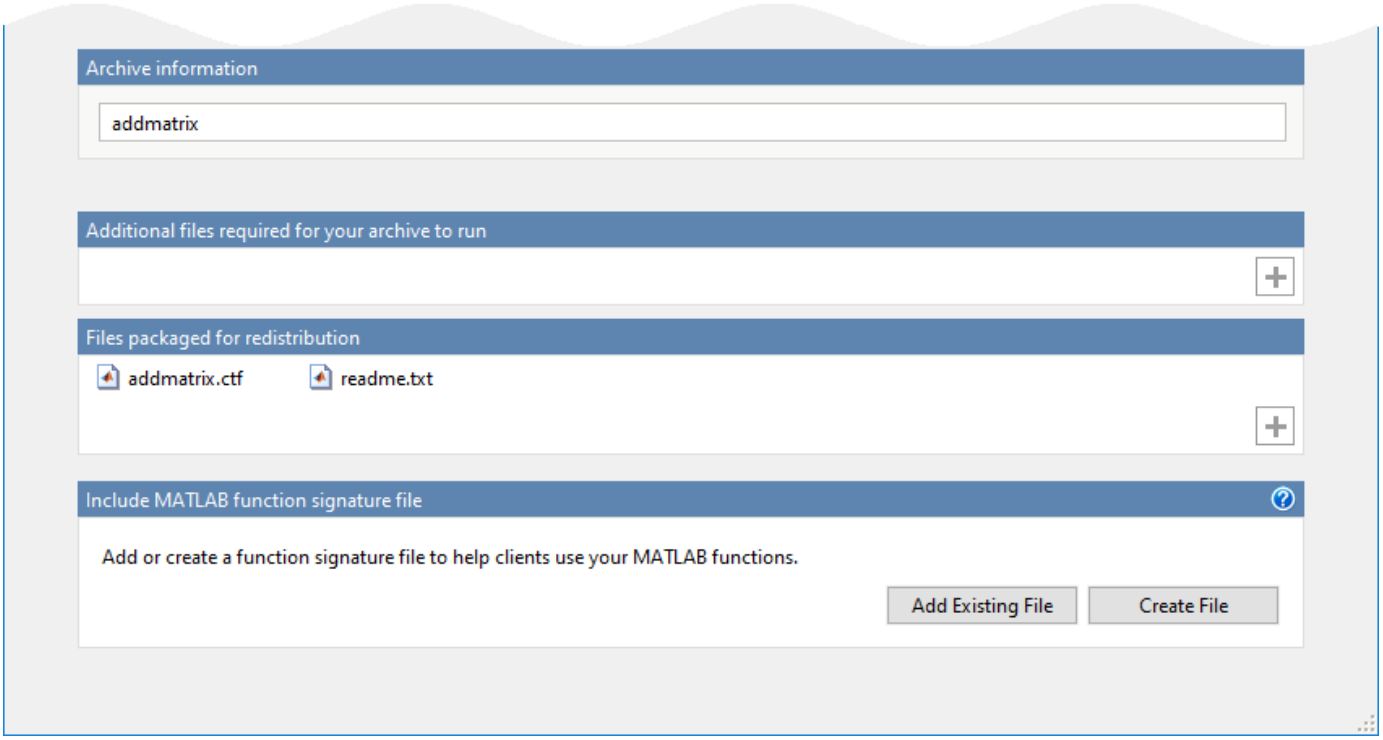
The function `addmatrix.m` is added to the list of main files.

Customize the Application and Its Appearance

You can customize your deployable archive, and add more information about the application as follows:

- **Archive information** — Editable information about the deployed archive.
- **Additional files required for your archive to run** — Additional files required by the generated archive to run. These files are included in the generated archive installer. See “Manage Required Files in Compiler Project”.
- **Files packaged for redistribution** — Files that are installed with your application. These files include:
  - Generated deployable archive
  - Generated `readme.txt`

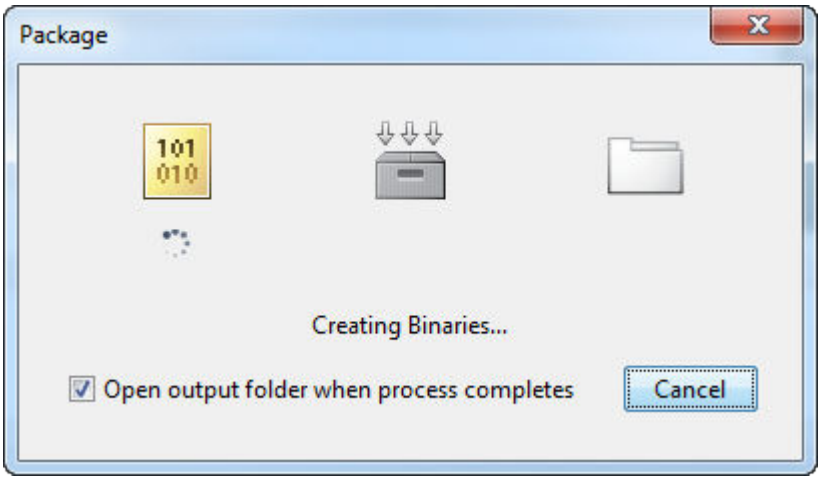
  See “Specify Files to Install with Application”
- **Include MATLAB function signature file** — Add or create a function signature file to help clients use your MATLAB functions.



Package the Application

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

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



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

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

- `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` — Log file generated by the packaging tool.

See Also

`productionServerCompiler` | `mcc` | `deploytool`

More About

- Production Server Compiler (MATLAB Production Server)

Package Deployable Archives from Command Line

In this section...

You can package deployable archives at the MATLAB prompt or your system prompt using either of these commands.

- `deploytool` invokes the Application Compiler app to execute a saved compiler project.
- `mcc` invokes the MATLAB Compiler to create a deployable application at the command prompt.

Execute Compiler Projects with deploytool

The `deploytool` command has two flags that invoke one of the compiler apps to package an already existing project without opening a window.

- `-build` *`project_name`* — Invoke the correct compiler app to build the project but 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.

Package a Deployable Archive with mcc

The `mcc` command invokes the MATLAB Compiler and provides fine-level control over the packaging 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 packaging deployable archives, you can also use the following 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 packaging are written.
`class{`*`className`*`:`*`mfilename`*`...}`	Specify that an additional class is generated that includes methods for the listed MATLAB files.

Differences Between Compiler Apps and Command Line

You perform the same functions using the compiler apps, a `compiler.build` function, or the `mcc` command-line interface. The interactive menus and dialog boxes 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 packaging it using `mcc`.

If you know the commands for the type of application you want to deploy and do not require an installer, it is faster to execute either `compiler.build` or `mcc` than go through the compiler app workflow.

Compiler app advantages include:

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

See Also

`mcc` | `deploytool`

More About

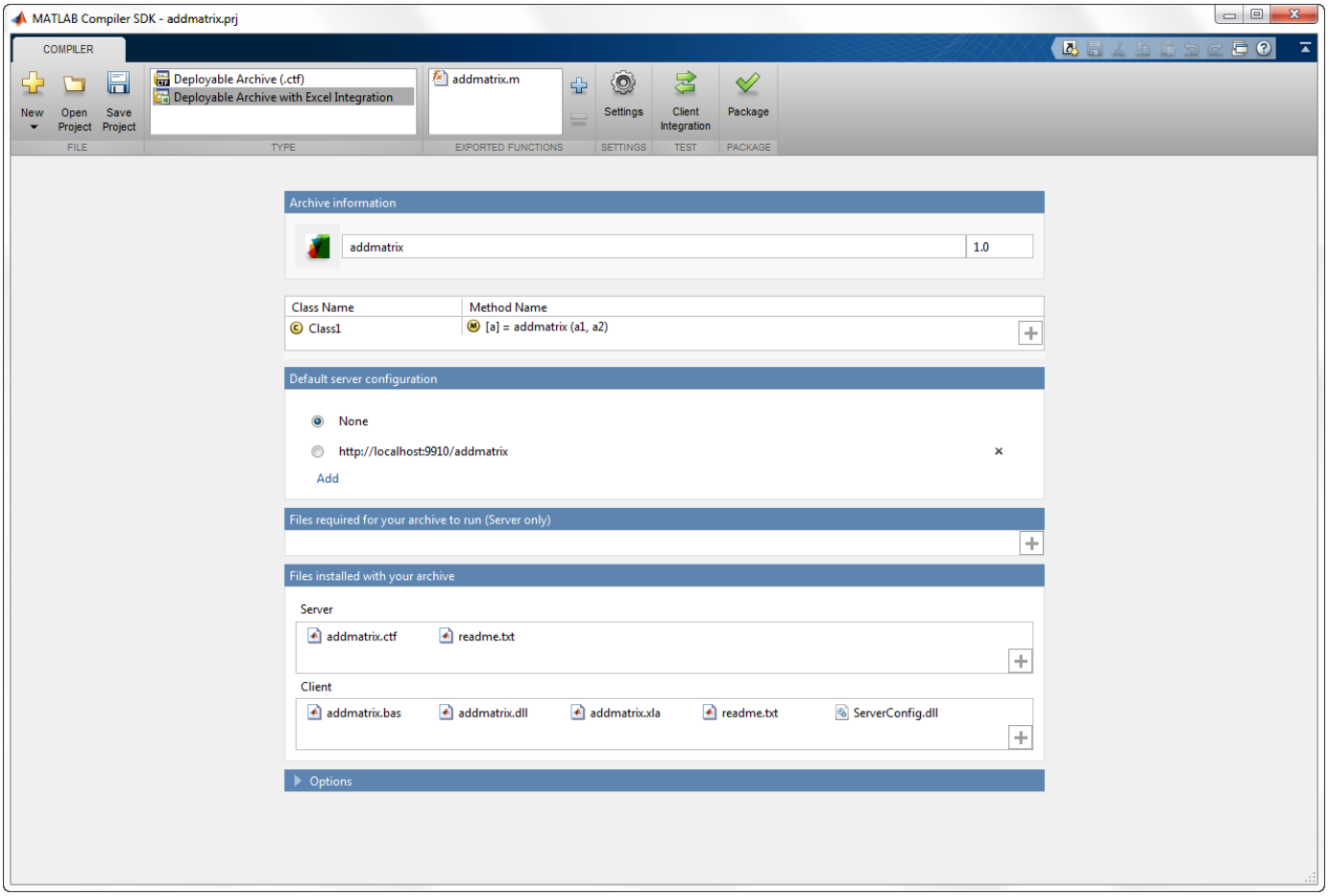
- “Package Deployable Archives with Production Server Compiler App” on page 7-2

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.

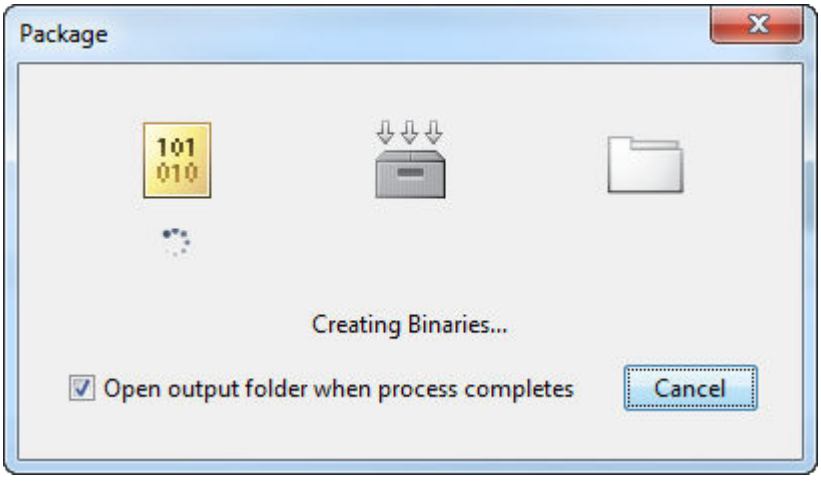
9 Inspect the **Files required for your archive to run** and **Files installed with your archive** sections of the app.

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" (MATLAB Production Server).

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

Package a COM Component

Package COM Components from Command Line

You can package COM components at the MATLAB prompt or your system prompt using either of these commands.

- `deploytool` invokes the Application Compiler app to execute a saved compiler project.
- `mcc` invokes the MATLAB Compiler to create a deployable application at the command prompt.

Execute Compiler Projects with deploytool

The `deploytool` command has two flags that invoke one of the compiler apps to package an already existing project without opening a window.

- `-build` *`project_name`* — Invoke the correct compiler app to build the project but 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.

Create COM Component with mcc

The `mcc` command invokes MATLAB Compiler to create a COM component at the command prompt and provides fine-level control while packaging the component. It does not package the results in an installer.

A MATLAB class cannot be directly packaged 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.

Action to Perform	Description
Create component that has one class.	**`mcc` option to use:** `-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.

Action to Perform	Description
	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.	**`mcc` option to use:** 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.
	Syntax `class{<class_name>:[file, [file,...]]}`
	Example `mcc -B 'com: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.	**`mcc` option to use:** `-B com:` Uses the bundle.
	Syntax `mcc -B '<bundle>'[:<a1>,<a2>,...,<an>]`
	Example `mcc -B 'com:mycomponent,myclass,1.0' foo.m bar.m`

Action to Perform	Description
Control how each COM class uses the MATLAB Runtime.	**`mcc` option to use:** `-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. **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 'com: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.	**`mcc` option to use:** `-d` The `\src` and `\distrib` subfolders are used to package components.
	Syntax `-d` *`foldername`*

Differences Between Compiler Apps and Command Line

You perform the same functions using the compiler apps, a `compiler.build` function, or the `mcc` command-line interface. The interactive menus and dialog boxes 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 packaging it using `mcc`.

If you know the commands for the type of application you want to deploy and do not require an installer, it is faster to execute either `compiler.build` or `mcc` than go through the compiler app workflow.

Compiler app advantages include:

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

See Also

`mcc` | `deploytool`

More About

- “Create a Generic COM Component with MATLAB Code”

Customizing a Compiler Project

Customize an Application

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

Customize the Installer

Change Application Icon

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

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

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

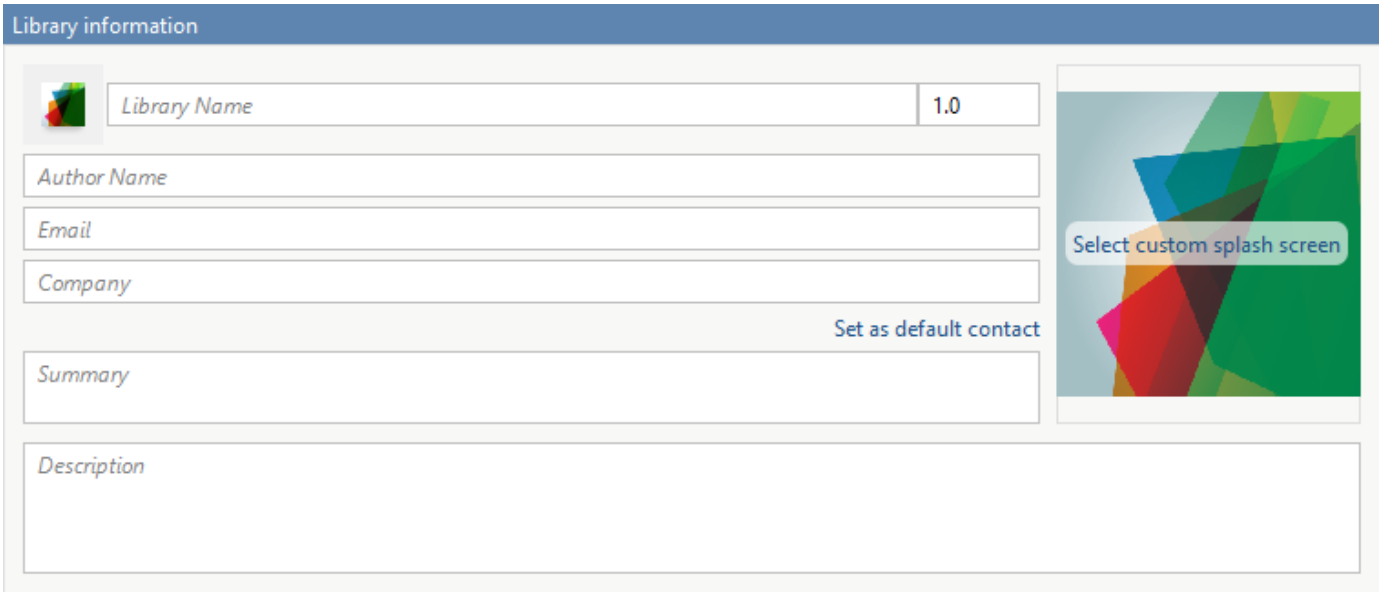
Add Library or Application Information

You can provide further information about your application as follows:

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

  The default value is the name of the first function listed in the **Main File(s)** field of the app.
- Version: The default value is 1.0.
- Author name: Name of the developer.
- Support email address: Email address to use for contact information.
- Company name: The full installation path for the installed MATLAB artifacts. For example, if the company name is `bar`, the full installation path would be *`InstallRoot`*`/bar/`*`ApplicationName`*.
- Summary: Brief summary describing the application.
- Description: Detailed explanation about the application.

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



Change the Splash Screen

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

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

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

Change the Installation Path

This table lists the default path the installer uses when installing the packaged binaries onto a target system.

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

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

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

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

Change the Logo

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

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

Edit the Installation Notes

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

Manage Required Files in Compiler Project

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

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

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

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

Using mcc

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

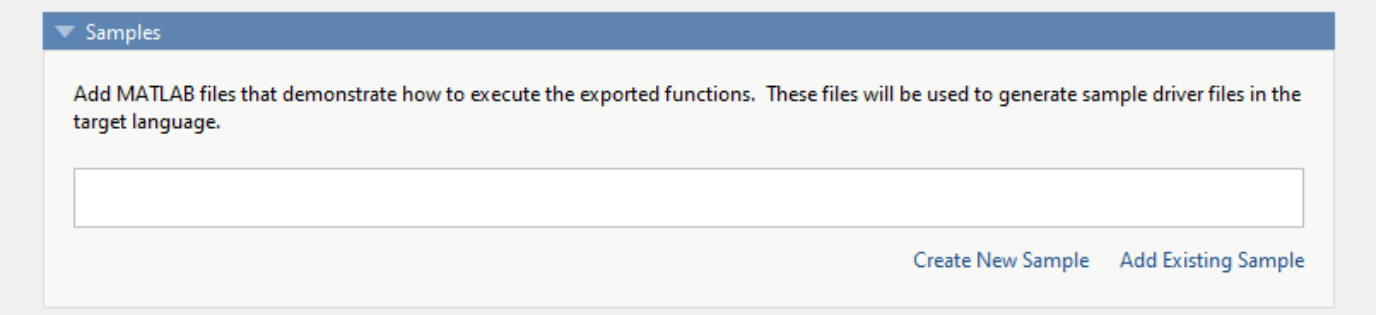
You can add files to the list by passing one or more `-a` arguments to `mcc`. The `-a` arguments add the specified files to the list of files to be added into the generated binary. For example, `-a hello.m` adds the file `hello.m` to the list of required files and `-a ./foo` adds all the files in `foo`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

Sample Driver File Creation

Sample driver files are used to implement the generated component into an application in the target language.

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

- C++ shared library
- Java package
- .NET assembly
- Python package



The sample file creation feature in **Library Compiler** uses MATLAB code to generate sample files in the target language. In the app, click **Create New Sample** to automatically generate a new MATLAB script, or click **Add Existing Sample** to upload a MATLAB script that you have already written. After you package your functions, a sample file in the target language is generated from your MATLAB script and is saved in a folder named `samples`. Sample files are also included in the installer.

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

The sample MATLAB files must follow these guidelines:

- The sample file must be a MATLAB script, not a function.
- The sample file code must use only exported functions. Any user-defined function called in the script must be a top-level exported function.
- Each exported function must be in a separate sample file.
- Each call to the same exported function must be a separate sample file.
- The input parameters of the top-level function are analyzed during the process. An input parameter cannot be a field in a struct.
- The output of the exported function must be an n-dimensional numeric, char, logical, struct, or cell array.
- Data must be saved as a local variable and then passed to the exported function in the sample file code.
- Sample file code should not require user interaction.
- The sample script is executed as part of the process of generating the target language sample code. Any errors in execution (for instance, undefined variables) will prevent a sample from being generated, although the build target will still be generated.

Additional considerations specific to the target language are as follows:

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

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

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

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

If you have already created a MATLAB sample file, you can include it in a `compiler.build` function for the supported targets using the `'SampleGenerationFiles'` option.

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

Specify Files to Install with Application

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

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

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

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

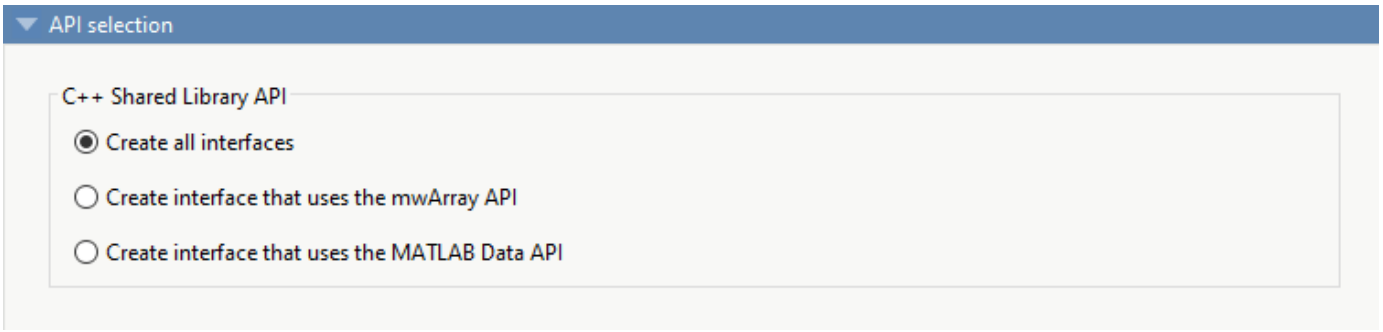
Caution Removing the binary targets from the list results in an installer that does not install the intended functionality.

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

Additional Runtime Settings

Type of Packaged Application	Additional Runtime Settings Options
Generic COM Components	• **Register the component for the current user (Recommended for non-admin users)** —This option enables registering the component for the current user account. It is provided for users without admin rights.
.NET Assembly	• **Create Shared Assembly** — Enables sharing MATLAB Runtime installer instances for multiple .NET assemblies. • **Enable .NET Remoting** — Enables you to remotely access MATLAB functionality, as a part of a distributed system. For more information, see “Create Remotable .NET Assembly”. • **Enable Type Safe API** — Enables the type safe API for the packaged .NET assembly.

API Selection for C++ Shared Library



- **Create all interfaces** — Create interfaces for shared libraries using both the `mwArray` API and the MATLAB Data API.
- **Create interface that uses the mwArray API** — Create an interface for a shared library using the `mwArray` API. The interface uses C-style functions to initialize the MATLAB Runtime, load the compiled MATLAB functions into the MATLAB Runtime, and manage data that is passed between the C++ code and the MATLAB Runtime. The interface supports only C++03 functionality. For an example, see “Generate a C++ mwArray API Shared Library and Build a C++ Application”.
- **Create interface that uses the MATLAB Data API** — Create an interface for a shared library using MATLAB Data API. It uses a generic interface that has modern C++ semantics. The interface supports C++11 functionality. For more information, see “Generate a C++ MATLAB Data API Shared Library and Build a C++ Application”.

See Also

Application Compiler | Library Compiler

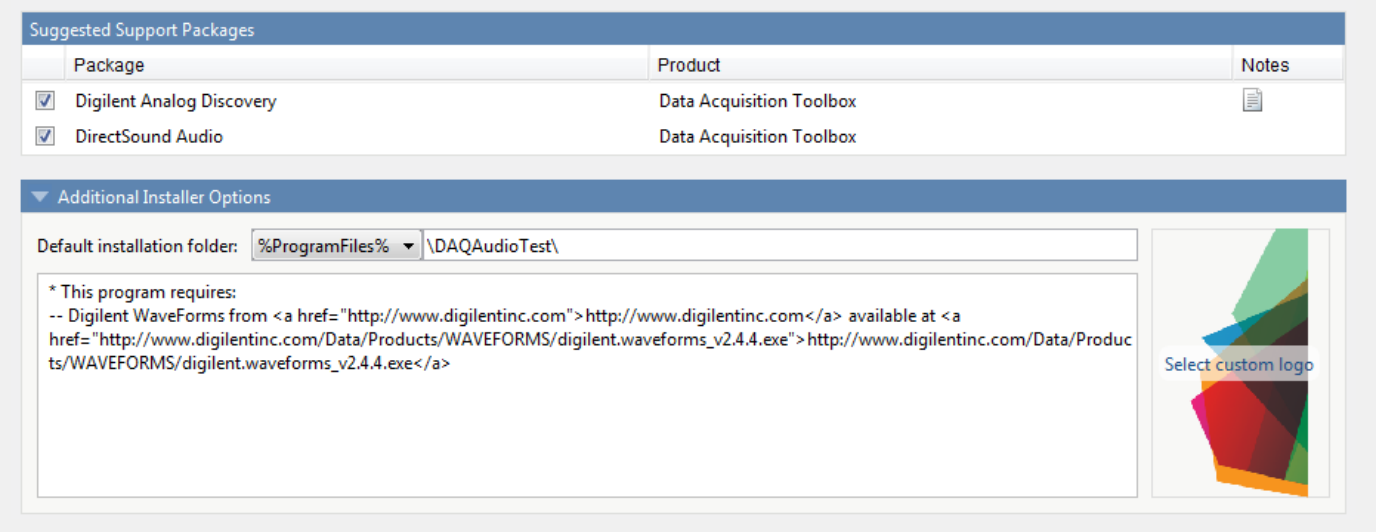
More About

- “Create a C Shared Library with MATLAB Code”
- “Generate a C++ mwArray API Shared Library and Build a C++ Application”
- “Generate a C++ MATLAB Data API Shared Library and Build a C++ Application”
- “Generate .NET Assembly and Build .NET Application”
- “Create a Generic COM Component with MATLAB Code”
- “Generate Java Package and Build Java Application”
- “Generate a Python Package and Build a Python Application”

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\R2023a\toolbox\daq\supportpackages\daqaudio ...
-a 'C:\MATLAB\SupportPackages\R2023a\resources\daqaudio'
```

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.

Advanced Uses of the Command Line Compiler

Simplify Compilation Using Macros

In this section...

Macros

The `mcc` function, through its exhaustive set of options, allows you to customize the behavior of a compiled component. If you want a simplified approach to compilation, you can use a *macro* 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`*`,`*`security`*`,`*`remote_type`* `-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. This is true for embedded `.ctf` files only.	On macOS, this variable is ignored in MATLAB R2020a and later. The app bundle contains the files necessary for runtime.
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 10-7 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 “About the Deployable Archive”.

mcc Command Arguments Listed Alphabetically

Option	Description	Comment
`-?`	Display `mcc` help message.	Cannot be used in a `deploytool` app.
`-a filepath`	Add `filepath` to the deployable archive.	If you specify a folder name, all files in the folder are added. If you use a wildcard (*), all files matching the wildcard are added.
`-A arch`	Append supported platforms to those detected automatically by the compiler.	Valid only for Python, C/C++ using the MATLAB data array API, and Java targets. *arch* = `win64`, `maci64`, `glnxa64`, or `all`
`-b`	Generate Excel compatible formula function.	Requires MATLAB Compiler for Excel add-ins. Cannot be used in a `deploytool` app.
`-B bundle[:parameters]`	Replace `-B bundle` on the `mcc` command line with the contents of *`bundle`*.	The file should contain only `mcc` command-line options. MathWorks included bundle files are located in *`matlabroot`*`\toolbox\compiler\bundles`.
`-c`	Suppress compiling and linking of the generated C wrapper code.	Must be used in conjunction with the `-l` option.
`-C`	Direct `mcc` to not embed the deployable archive in generated binaries.	
`-d outputfolder`	Place output in folder specified by *`outputfolder`*.	The specified folder must already exist. Cannot be used in a `deploytool` app.
`-e`	Suppresses appearance of the MS-DOS Command Window when generating a standalone application.	Use *`-e`* in place of the *`-m`* option. Available for Windows only. Equivalent to `-W WinMain -T link:exe`. Cannot be used in a `deploytool` app. The standalone app compiler suppresses the MS-DOS command window by default. To enable it, deselect **Do not display the Windows Command Shell (console) for execution** in the **Additional Runtime Settings** area.
`-f filename`	Use the specified options file, *`filename`*, when calling `mbuild`.	`mbuild -setup` is recommended. Valid for C/C++ shared libraries, COM, and Excel targets.
`-G`	Include debugging symbol information for generated C/C++ code.	
`-h helpfile`	Specify a custom help text file.	Display help file contents at runtime using `-?` or `/?`. Valid for standalone applications, C/C++ shared libraries, COM, and Excel targets.
`-I folder`	Add folder to search path for MATLAB files.	

Option	Description	Comment
`-j`	Automatically convert all `.m` files to P-files before packaging.	
`-k 'file=<keyfile>;loader=<mexfile>'`	Specify AES encryption key *`keyfile`* and MEX-file loader interface *`mexfile`* to retrieve decryption key at runtime.	If you do not specify any arguments after `-k`, `mcc` generates a 256-bit AES key and a loader MEX-file.
`-K`	Directs `mcc` to not delete output files if the compilation ends prematurely, due to error.	Default behavior is to dispose of any partial output if the command fails to execute successfully.
`-l`	Create a C shared library.	Equivalent to `-W lib -T link:lib`. Cannot be used in a `deploytool` app.
`-m`	Generate a standalone application.	Equivalent to `-W main -T link:exe`. Cannot be used in a `deploytool` app. On Windows, the command prompt opens on execution of the application. The standalone app compiler suppresses the MS-DOS command window by default. To enable it, deselect **Do not display the Windows Command Shell (console) for execution** in the **Additional Runtime Settings** area.
`-M options`	Pass compile-time options to `mbuild`.	
`-n`	Automatically treat numeric inputs as MATLAB doubles.	Cannot be used in a `deploytool` app.
`-N`	Clear the path of all but a minimal, required set of folders.	Uses the following folders: • *`matlabroot`*`\toolbox\matlab` • *`matlabroot`*`\toolbox\local` • *`matlabroot`*`\toolbox\compiler` • *`matlabroot`*`\toolbox\shared\bigdata`
`-o executablename`	Specify name of standalone application executable file.	Adds appropriate extension. Cannot be used in a `deploytool` app.
`-p folder`	Add *`folder`* to compilation path in an order-sensitive context.	Requires `-N` option.
`-r icon`	Embed resource *`icon`* in binary.	
`-R option`	Specify run-time options for MATLAB Runtime.	Valid only for standalone applications using MATLAB Compiler. *`option`* = `-nojvm`, `-nodisplay`, `'-logfile` *`filename`*`'`, `-startmsg`, and `-completemsg` *`filename`*

Option	Description	Comment
`-s`	Obfuscate folder structures and file names in the deployable archive (`.ctf` file) from the end user.	
`-S`	Create singleton MATLAB Runtime.	Default for generic COM components. Valid for Microsoft Excel and Java packages.
`-T phase:type`	Specify the output target phase and type.	Cannot be used in a `deploytool` app.
`-u`	Registers COM component for current user only on development machine.	Valid only for generic COM components and Microsoft Excel add-ins.
`-U`	Generate a deployable archive (`.ctf` file) for MATLAB Production Server.	Equivalent to `-W 'CTF'`. Cannot be used in a `deploytool` app.
`-v`	Verbose; display compilation steps.	
`-w option[:warning]`	Control warning messages.	Valid arguments are `list`, `enable[:`*`warning`*`]`, `disable[:`*`warning`*`]`, `error[:`*`warning`*`]`, `on[:`*`warning`*`]`, and `off[:`*`warning`*`]`.
`-W 'target[:options]'`	Specify build target and associated options.	*`target`* = `main`, `WinMain`, `excel`, `hadoop`, `spark`, `lib`, `cpplib`, `com`, or `dotnet`, `java`, `python`, `CTF`, or `mpsxl`. Cannot be used in a `deploytool` app.
`-X`	Ignore data files detected by dependency analysis.	For more information, see “Dependency Analysis Using MATLAB Compiler”.
`-Y licensefile`	Override the default license file with the specified file *`licensefile`*.	Can only be used on the system command line.
`-Z supportpackage`	Specify method of including support packages.	*`supportpackage`* = `'autodetect'` (default), `'none'`, or *`packagename`*.

Packaging Log and Output Folders

By default, the deployment app places the packaging log and the **Testing Files**, **End User Files**, and **Packaged Installers** folders in the target folder location. If you specify a custom location, the app creates any folders that do not exist at compile time.

Work with the MATLAB Runtime

Install and Configure MATLAB Runtime

Supported Platforms: Windows, Linux, macOS

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

Download MATLAB Runtime Installer

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

- Download the MATLAB Runtime installer at the latest update level for the selected release from the website at https://www.mathworks.com/products/compiler/matlab-runtime.html. This option is best for end users who want to run deployed applications.
- Use the MATLAB function `compiler.runtime.download` to download the MATLAB Runtime installer matching the version and update level of MATLAB from where the command is executed. If the installer has already been downloaded to the machine, it returns the path to the MATLAB Runtime installer. If the machine is offline, it returns a URL to the MATLAB Runtime installer. This option is best for developers who want to create application installers that contain MATLAB Runtime.

Note If you want to install MATLAB Runtime to a shared network drive, see “Run Applications Using a Network Installation of MATLAB Runtime”.

Install MATLAB Runtime Interactively

To install MATLAB Runtime:

1 Extract the archive containing the MATLAB Runtime installer. The release part of the installer file name (*_R2023a_*) changes from one release to the next.

Platform	Steps
Windows	Unzip the MATLAB Runtime installer. Right-click the ZIP file `MATLAB_Runtime_R2023a_win64.zip` and select **Extract All**.
Linux	Unzip the MATLAB Runtime installer at the terminal using the `unzip` command. For example, if you are unzipping the R2023a MATLAB Runtime installer, at the terminal, type: `unzip MATLAB_Runtime_R2023a_glnxa64.zip`
macOS	Unzip the MATLAB Runtime installer at the terminal using the `unzip` command. For example, if you are unzipping the R2023a MATLAB Runtime installer, at the terminal, type: `unzip MATLAB_Runtime_R2023a_maci64.zip`

2 Start the MATLAB Runtime installer.

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

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

macOS	At the terminal, type: `./install` **Note** You may need to enter an administrator user name and password after you run `./install`.

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

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

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

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

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

6 On Linux and macOS platforms, after copying files to your disk, the MATLAB Runtime installer displays the **Product Configuration Notes** dialog box. This dialog box contains information necessary for setting your path environment variables. Copy the path information from this dialog box, save it to a text file, and then click **Next**.

For instructions on setting the path environment variables, see “Set MATLAB Runtime Path for Deployment” on page 11-11.

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

Default Install Folder

The default MATLAB Runtime installation folders for R2023a are specified in the following table:

Operating System	MATLAB Runtime Installation Folder
Windows	`C:\Program Files\MATLAB\MATLAB Runtime\R2023a`
Linux	`/usr/local/MATLAB/MATLAB_Runtime/R2023a`
macOS	`/Applications/MATLAB/MATLAB_Runtime/R2023a`

Install MATLAB Runtime Noninteractively

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

When running noninteractively, the installer overwrites the installation location.

Caution On Linux and macOS systems, the installer displays information necessary for setting your environment variables in the **Product Configuration Notes** dialog box. If you use the installer noninteractively, you must locate your MATLAB Runtime installation directory in order to set the library path after installation. For more information, see “Set MATLAB Runtime Path for Deployment” on page 11-11.

Run Installer in Silent Mode

To install MATLAB Runtime in silent mode:

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

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

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

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

4 View a log of the installation.

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

Customize Noninteractive Installation

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

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

```
./install -agreeToLicense yes \
-outputFile myapp_log.txt -destinationFolder ~/MW/MATLAB_Runtime
```

Option	Description
`-agreeToLicense`	Agree to the MATLAB Runtime license.
`-destinationFolder`	Specifies where MATLAB Runtime is installed.
`-outputFile`	Specifies where the installation log file is written.

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

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

Install MATLAB Runtime without Administrator Rights

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

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

1 Install MATLAB Runtime on a Windows machine where you have administrator rights.
2 Copy the folder where MATLAB Runtime was installed to the machine without administrator rights. You can compress the folder into a zip file for distribution.
3 On the machine without administrator rights, add the *<MATLAB_RUNTIME_INSTALL_DIR>*`\runtime\`*`arch`* directory to the user's `PATH` environment variable. For more information, see "Set MATLAB Runtime Path for Deployment" on page 11-11.

Install Multiple MATLAB Runtime Versions on Single Machine

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

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

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

MATLAB and MATLAB Runtime on Same Machine

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

You can install MATLAB Runtime for debugging purposes.

Modify Path

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

To run deployed MATLAB code against MATLAB Runtime rather than MATLAB, ensure that your library path lists the MATLAB Runtime directories before any MATLAB directories. For information on setting environment variables, see “Set MATLAB Runtime Path for Deployment” on page 11-11.

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

Uninstall MATLAB Runtime

Windows

1. Start the uninstaller.

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

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

Linux

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

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

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

macOS

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

See Also

`compiler.runtime.download`

More About

- About MATLAB Runtime
- "MATLAB Runtime Startup Options"
- "Set MATLAB Runtime Path for Deployment" on page 11-11

MATLAB Runtime Startup Options

Set MATLAB Runtime Options

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

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

You can set options in the following ways:

- The **Additional Runtime Settings** area of the compiler apps.
- The `mcc` command using the `-R` flag.

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

Compiler App

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

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

Set Multiple Options Using -R

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

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

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 is started with an instance of a Java Virtual Machine (JVM).	Boolean (`true` or `false`). Returns `true` if MATLAB Runtime has been started 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, preceded by the character.
`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

The following example demonstrates how to pass options to a C or C++ shared library and how to retrieve the corresponding values after they are set.

```
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);
```

See Also

mcc | deploytool

Set MATLAB Runtime Path for Deployment

In this section...

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

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

Note

- Your library path may contain multiple versions of MATLAB Runtime. Applications launched without using the shell script use the first version listed in the path.
- Save the value of your current library path as a backup before modifying it.
- If you are using a network install of MATLAB Runtime, see “Run Applications Using a Network Installation of MATLAB Runtime”.

Library Path Environment Variables and MATLAB Runtime Folders

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

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

macOS	`DYLD_LIBRARY_PATH`	`<`*`MATLAB_RUNTIME_INSTALL_DIR`*`>/runtime/maci64`
		`<`*`MATLAB_RUNTIME_INSTALL_DIR`*`>/bin/maci64`
		`<`*`MATLAB_RUNTIME_INSTALL_DIR`*`>/sys/os/maci64`
		`<`*`MATLAB_RUNTIME_INSTALL_DIR`*`>/extern/bin/maci64`

Windows

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

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

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

   For example, if you are using MATLAB Runtime R2023a located in the default installation folder on 64-bit Windows, add `C:\Program Files\MATLAB\MATLAB Runtime\R2023a\runtime\win64`.
4. Click **OK** to apply the change.

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

Linux

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

Bash Shell

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

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

   ```
   export LD_LIBRARY_PATH="${LD_LIBRARY_PATH:+${LD_LIBRARY_PATH}:}\
   <MATLAB_RUNTIME_INSTALL_DIR>/runtime/glnxa64:\
   <MATLAB_RUNTIME_INSTALL_DIR>/bin/glnxa64:\
   <MATLAB_RUNTIME_INSTALL_DIR>/sys/os/glnxa64:\
   <MATLAB_RUNTIME_INSTALL_DIR>/extern/bin/glnxa64"
   ```

   Note If you require Mesa Software OpenGL® rendering to resolve low level graphics issues, add the folder *<MATLAB_RUNTIME_INSTALL_DIR>*`/sys/opengl/lib/glnxa64` to the path. For details, see “Resolving Low-Level Graphics Issues”.

   For example, if you are using MATLAB Runtime R2023a located in the default installation folder, use the following command:

   ```
   export LD_LIBRARY_PATH="${LD_LIBRARY_PATH:+${LD_LIBRARY_PATH}:}\
   /usr/local/MATLAB/MATLAB_Runtime/R2023a/runtime/glnxa64:\
   ```

```
/usr/local/MATLAB/MATLAB_Runtime/R2023a/bin/glnxa64:\
/usr/local/MATLAB/MATLAB_Runtime/R2023a/sys/os/glnxa64:\
/usr/local/MATLAB/MATLAB_Runtime/R2023a/extern/bin/glnxa64"
```

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

   ```
   echo $LD_LIBRARY_PATH
   ```

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

   ```
   export LD_PRELOAD="${LD_PRELOAD:+${LD_PRELOAD}:}\
   <MATLAB_RUNTIME_INSTALL_DIR>/bin/glnxa64/glibc-2.17_shim.so"
   ```

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

macOS

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

   ```
   echo $DYLD_LIBRARY_PATH
   ```

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

   ```
   export DYLD_LIBRARY_PATH="${DYLD_LIBRARY_PATH:+${DYLD_LIBRARY_PATH}:}\
   <MATLAB_RUNTIME_INSTALL_DIR>/runtime/maci64:\
   <MATLAB_RUNTIME_INSTALL_DIR>/bin/maci64:\
   <MATLAB_RUNTIME_INSTALL_DIR>/sys/os/maci64:\
   <MATLAB_RUNTIME_INSTALL_DIR>/extern/bin/maci64"
   ```

   For example, if you are using MATLAB Runtime R2023a located in the default installation folder, use the following command:

   ```
   export DYLD_LIBRARY_PATH="${DYLD_LIBRARY_PATH:+${DYLD_LIBRARY_PATH}:}\
   /Applications/MATLAB/MATLAB_Runtime/R2023a/runtime/maci64:\
   /Applications/MATLAB/MATLAB_Runtime/R2023a/bin/maci64:\
   /Applications/MATLAB/MATLAB_Runtime/R2023a/sys/os/maci64:\
   /Applications/MATLAB/MATLAB_Runtime/R2023a/extern/bin/maci64"
   ```

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

   ```
   echo $DYLD_LIBRARY_PATH
   ```

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

Set Path Permanently on UNIX

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

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

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

See Also

More About

- “Install and Configure MATLAB Runtime”
- “Run Applications Using a Network Installation of MATLAB Runtime”
- “Change Environment Variable for Shell Command”

Using MATLAB Runtime User Data Interface

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

- You need to supply MATLAB Runtime profile information to a client running an application created with the Parallel Computing Toolbox™. You supply and change profile information on a per-execution basis. For example, two instances of the same application may run simultaneously with different profiles. For more information, see “Use Parallel Computing Toolbox in Deployed Applications”.
- You want to initialize MATLAB Runtime with constant values that can be accessed by all your MATLAB applications.
- You want to set up a global workspace — a global variable or variables that MATLAB and your client can access.
- You want to store the state of any variable or group of variables.

MATLAB Functions

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

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

Set and Retrieve MATLAB Runtime Data for Shared Libraries

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

1. In your code, include the MATLAB Runtime header file and the library header generated by MATLAB Compiler SDK.
2. Properly initialize your application using `mclInitializeApplication`.
3. After creating your input data, write or *set* it to the MATLAB Runtime with `setmcruserdata`.
4. After calling functions or performing other processing, retrieve the new MATLAB Runtime data with `getmcruserdata`.
5. Free up storage memory in work areas by disposing of unneeded arrays with `mxDestroyArray`.
6. Shut down your application properly with `mclTerminateApplication`.

See Also

`setmcruserdata` | `getmcruserdata`

More About

- “Use Parallel Computing Toolbox in Deployed Applications”
- “Specify Parallel Computing Toolbox Profile in .NET Application”
- “Specify Parallel Computing Toolbox Profile in Java Application”

Display MATLAB Runtime Initialization Messages

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

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

You have the following options:

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

Some examples of different ways to invoke this option follow:

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

Best Practices

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

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

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

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

Limitations and Restrictions

Limitations

Packaging MATLAB and Toolboxes

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

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

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

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

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

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

Note For a list of functions not supported by the MATLAB Compiler SDK See “Functions Not Supported for Compilation by MATLAB Compiler and MATLAB Compiler SDK” on page 12-8.

Fixing Callback Problems: Missing Functions

When MATLAB Compiler SDK creates a standalone application, it packages the MATLAB files that you specify on the command line. In addition, it includes any other MATLAB files that your packaged MATLAB files call. MATLAB Compiler 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 cannot discover the p-file dependencies.

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

- Callback string

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

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

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

Symptom

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

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

Workaround

There are several ways to eliminate this error:

- Using the `%#function pragma` and specifying callbacks as character arrays
- Specifying callbacks with function handles
- Using the `-a` option

Specifying Callbacks as Character Arrays

Create a list of all the functions that are specified only in callback character arrays and pass these functions using separate `%#function` pragma statements. This overrides the product dependency analysis and instructs it to explicitly include the functions listed in the `%#function` pragmas.

For example, the call to the `change_colormap` function in the sample application `my_test` illustrates this problem. To make sure MATLAB Compiler 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 need to be listed in a `%#function` pragma, search your MATLAB file source code for text specified as callback character arrays or as arguments to the `feval`, `fminbnd`, `fminsearch`, `funm`, and `fzero` functions or any ODE solvers.

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

Suppressing Warnings on the UNIX System

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

To suppress the `libjvm.so` warning, set the dynamic library path properly for your platform. See “Set MATLAB Runtime Path for 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 get a run-time error.

Cannot Create the Output File

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

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

Possible causes include:

- Lack of write permission for the folder where MATLAB Compiler 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 Packaged Functions

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

```
help filename
```

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

No MATLAB Runtime Versioning on Mac OS X

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

Older Neural Networks Not Deployable with MATLAB Compiler

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

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

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

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

Restrictions on Calling PRINTDLG with Multiple Arguments in Packaged Mode

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

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

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

Packaging a Function with which Does Not Search Current Working Folder

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

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

Use one of the following solutions as an alternative:

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

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

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

  ```
  load myFile.mat
  ```

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

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

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

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

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

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

Accepted File Types for Packaging

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

Target Application	Valid File Types	Invalid File Types
Standalone Application	MATLAB MEX files, MATLAB scripts, MATLAB functions, and MATLAB class files. These files must have a single entry point.	Protected function files (.p files), Java functions, COM or .NET components, and data files.
Library Compiler	MATLAB MEX files, MATLAB functions, and MATLAB class files. These files must have a single entry point.	MATLAB scripts, protected function files (.p files), Java functions, COM or .NET components, and data files.
MATLAB Production Server	MATLAB MEX files and MATLAB functions. These files must have a single entry point.	MATLAB scripts, MATLAB class files, protected function files (.p files), Java functions, COM or .NET components, and data files. MATLAB class files can be dependent files.

See Also

More About

- “Functions Not Supported for Compilation by MATLAB Compiler and MATLAB Compiler SDK” on page 12-8

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

Note Due to the number of active and ever-changing list of MathWorks products and functions, this is not a complete list of functions that cannot be compiled. If you have a question as to whether a specific MathWorks product's function is able to be compiled or not, the definitive source is that product's documentation. For an updated list of such functions, see Support for MATLAB and Toolboxes.

Functions that cannot be compiled fall into the following categories:

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

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

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

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

List of Unsupported Functions and Programs

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

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

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 “Access Files in Packaged Applications” on page 2-9.

Version History

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,
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*`,` *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.

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:
	• 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`.
`Info = componentinfo('mycomponent',2,3)`	Information for revision 2.3 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.

Version History

Introduced before R2006a

compiler.package.installer

Create an installer for files generated by MATLAB Compiler

Syntax

```
compiler.package.installer(results)
compiler.package.installer(results,Name,Value)
compiler.package.installer(results,'Options',opts)
compiler.package.installer(files,filePath,'ApplicationName',appName)
compiler.package.installer(files,filePath,'ApplicationName',appName,
Name,Value)
compiler.package.installer(files,filePath,'Options',opts)
```

Description

`compiler.package.installer(results)` creates an installer using the `compiler.build.Results` object `results` generated from a `compiler.build` function.

`compiler.package.installer(results,Name,Value)` creates an installer using the `compiler.build.Results` object `results` with additional options specified using one or more name-value arguments.

`compiler.package.installer(results,'Options',opts)` creates an installer using the `compiler.build.Results` object `results` with installer options specified by an `InstallerOptions` object `opts`. If you use an `InstallerOptions` object, you cannot specify any other options using name-value arguments.

`compiler.package.installer(files,filePath,'ApplicationName',appName)` creates an installer for files generated by the `mcc` command. The installed application name is specified by `appName`. The installer file extension is determined by the operating system in which you run the function.

`compiler.package.installer(files,filePath,'ApplicationName',appName, Name,Value)` creates an installer for files generated by the `mcc` command. The installed application name is specified by `appName`. The installer can be customized using optional name-value arguments.

`compiler.package.installer(files,filePath,'Options',opts)` creates an installer for files generated by the `mcc` command with installer options specified by an `InstallerOptions` object `opts`. If you use an `InstallerOptions` object, you cannot specify any other options using name-value arguments.

Examples

Create Installer Using Results Object

Create an installer for a standalone application using the results from the `compiler.build.standaloneApplication` function.

In MATLAB, locate the MATLAB code that you want to deploy as a standalone application. For this example, compile using the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

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

Build a standalone application using the `compiler.build.standaloneApplication` command.

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

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

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

The function generates an installer named `MyAppInstaller` within a folder named `magicsquareinstaller`.

Customize Installer Using Results Object

Create an installer for a standalone application using the results from the `compiler.build.standaloneApplication` function and customize it using name-value arguments.

Save the path to the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

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

Build a standalone application using the `compiler.build.standaloneApplication` command.

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

Create an installer for the standalone application using the `compiler.package.installer` function using the `Results` object. Use name-value arguments to specify the installer name and include MATLAB Runtime within the installer.

```
compiler.package.installer(results,...
    'InstallerName','MyMagicInstaller',...
    'RuntimeDelivery','installer');
```

The function generates an installer named `MyMagicInstaller` within a folder named `magicsquareinstaller`.

Customize Installer Using Results Object and Options Object

Create an installer for a standalone application on a Windows system using the results from the `compiler.build.standaloneApplication` function. Customize the installer using an `InstallerOptions` object.

Save the path to the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

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

Build a standalone application using the `compiler.build.standaloneApplication` command.

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

Create an `InstallerOptions` object. Use name-value arguments to specify the application name, author company, author name, installer name, and summary.

```
opts = compiler.package.InstallerOptions(results,...
    'ApplicationName','MagicSquare_Generator',...
    'AuthorCompany','Boston Common',...
    'AuthorName','Frog',...
    'InstallerName','MagicSquare_Installer',...
    'Summary','Generates a magic square.')
```

```
opts =

  InstallerOptions with properties:

           RuntimeDelivery: 'web'
           InstallerSplash: 'C:\Program Files\MATLAB\R2023a\toolbox\compiler\packagingResources\c
             InstallerIcon: 'C:\Program Files\MATLAB\R2023a\toolbox\compiler\packagingResources\c
             InstallerLogo: 'C:\Program Files\MATLAB\R2023a\toolbox\compiler\packagingResources\c
           AdditionalFiles: {}
     AddRemoveProgramsIcon: 'C:\Program Files\MATLAB\R2023a\toolbox\compiler\packagingResources\c
                AuthorName: 'Frog'
               AuthorEmail: ''
             AuthorCompany: 'Boston Common'
                   Summary: 'Generates a magic square.'
               Description: ''
         InstallationNotes: ''
                  Shortcut: 'D:\Work\magicsquarestandaloneApplication\magicsquare.exe'
                   Version: '1.0'
             InstallerName: 'MagicSquare_Installer'
           ApplicationName: 'MagicSquare_Generator'
                 OutputDir: '.\MagicSquare_Generatorinstaller'
    DefaultInstallationDir: 'C:\Program Files\MagicSquare_Generator'
```

Create an installer for the standalone application using the `Results` and `InstallerOptions` objects as inputs to the `compiler.package.installer` function.

```
compiler.package.installer(results,'Options',opts);
```

The function generates an installer named `MagicSquare_Installer` within a folder named `MagicSquare_Generatorinstaller`.

Create Installer Using Files

Create an installer for a standalone application on a Windows system.

Write a MATLAB function that generates a magic square. Save the function in a file named `mymagic.m`.

```
function out = mymagic(in)
out = magic(in)
```

Build a standalone application using the `mcc` command.

```
mcc -m mymagic.m
```

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

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

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

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

Customize Installer Using Files

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

Build a standalone application using the `compiler.build.standaloneApplication` command.

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

Save the path to the generated `requiredMCRProducts.txt` file.

```
runtimeProducts = fullfile(buildResults.Options.OutputDir,'requiredMCRProducts.txt')
```

Save the list of files from the standalone application build results.

```
fileList = buildResults.Files
```

Optionally, you can add additional files to the installer by modifying `fileList`. Additional files are installed in the installation directory along with the application executable.

```
fileList = [fileList; {'UsageNotes.txt'}];
```

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

```
compiler.package.installer(fileList, runtimeProducts,...
    'ApplicationName','CustomMagicSquare',...
    'InstallerName','Installer_With_Addl_Files',...
    'Summary','See UsageNotes.txt for info.')
```

Customize Installer Using Files and Installer Options Object

Customize an installer for a standalone application on a Windows system using an `InstallerOptions` object.

Create an `InstallerOptions` object.

```
opts = compiler.package.InstallerOptions('ApplicationName','MagicSquare_Generator',...
    'AuthorCompany','Boston Common',...
    'AuthorName','Frog',...
    'InstallerName','MagicSquare_Installer',...
    'Summary','Generates a magic square.')
```

```
opts =

  InstallerOptions with properties:

          RuntimeDelivery: 'web'
          InstallerSplash: 'C:\Program Files\MATLAB\R2023a\toolbox\compiler\packagingResources\c
            InstallerIcon: 'C:\Program Files\MATLAB\R2023a\toolbox\compiler\packagingResources\c
            InstallerLogo: 'C:\Program Files\MATLAB\R2023a\toolbox\compiler\packagingResources\c
          AdditionalFiles: {}
     AddRemoveProgramsIcon: ''
                AuthorName: 'Frog'
```

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

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

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

Input Arguments

`results` — Build results object

`Results` object

Build results, specified as a `compiler.build.Results` object. Create the `Results` object by saving the output from a `compiler.build` function.

`files` — List of files and folders for installation

character vector | string scalar | cell array of character vectors | string array

List of files and folders for installation, specified as a character vector, a string scalar, a cell array of character vectors, or a string array. These files are typically generated by the `mcc` command or a `compiler.build` function and can also include any additional files and folders required by the installed application to run. Additional files are installed in the installation directory along with the application executable.

- Files generated in a particular release can be packaged using the `compiler.package.installer` function of the same release.
- Files of type `.ctf` on one operating system can be packaged using the `compiler.package.installer` function on a different operating system, as long as the build command and the `compiler.package.installer` function are from the same release.

Example: `{'mymagic.exe','UsageNotes.txt'}`

Data Types: `char` | `string`

`filePath` — Path to `requiredMCRProducts.txt` file

character vector | string scalar

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

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

Data Types: `char` | `string`

`appName` — Name of the installed application

character vector | string scalar

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

Example: `'MagicSquare_Generator'`

Data Types: `char` | `string`

`opts` — Installer options object
`InstallerOptions` object

Installer options, specified as an `InstallerOptions` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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

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

Additional files and folders to be installed with the application, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute.

Example: `'AdditionalFiles',["myimage.png","data.mat"]`

Data Types: `char` | `string` | `cell`

`AddRemoveProgramsIcon` — Add or remove programs icon
character vector | string scalar | string array

Add or remove programs icon, specified as a character vector or string scalar. The image is used as the icon in the list of Windows applications within settings. Paths can be relative to the current working directory or absolute.

The default path is:

`'`*`matlabroot`*`\toolbox\compiler\packagingResources\default_icon_48.png'`

Example: `'AddRemoveProgramsIcon',"win_icon.png"`

Data Types: `char` | `string`

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

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

Example: `'ApplicationName','MagicSquare_Generator'`

Data Types: `char` | `string`

`AuthorCompany` — Company name
`''` (default) | character vector | string scalar

Name of company that created the application, specified as a character vector or a string scalar.

Example: `'AuthorCompany','Boston Common'`

Data Types: `char` | `string`

`AuthorEmail` — Email address
`''` (default) | character vector | string scalar

Email address of the application author, specified as a character vector or a string scalar.

Example: `'AuthorEmail','frog@example.com'`

Data Types: `char` | `string`

`AuthorName` — Name
`''` (default) | character vector | string scalar

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

Example: `'AuthorName','Frog'`

Data Types: `char` | `string`

`DefaultInstallationDir` — Default installation path
character vector | string scalar

Default directory where you want the installer to install the application, specified as a character vector or a string scalar.

If no path is specified, the default path for each operating system is:

Operating System	Default Installation Directory
Windows	`C:\Program Files\`*`appName`*
Linux	`/usr/`*`appName`*
macOS	`/Applications/`*`appName`*

Example: `'DefaultInstallationDir','C:\Users\MW_Programs\MagicSquare_Generator'`

Data Types: `char` | `string`

`Description` — Detailed application description
`''` (default) | character vector | string scalar

Detailed description of the application, specified as a character vector or a string scalar.

Example: `'Description','The MagicSquare_Generator application generates an n-by-n matrix constructed from the integers 1 through n2 with equal row and column sums.'`

Data Types: `char` | `string`

`InstallationNotes` — Notes
`''` (default) | character vector | string scalar

Notes about additional requirements for using application, specified as a character vector or a string scalar.

Example: `'InstallationNotes','This is a Linux installer.'`

Data Types: `char` | `string`

`InstallerIcon` — Path to icon image
character vector | string scalar

Path to an image file used as the icon for the installed application, specified as a character vector or a string scalar.

The default path is:

`'`*`matlabroot`*`\toolbox\compiler\packagingResources\default_icon_48.png'`

Example: `'InstallerIcon','D:\Documents\MATLAB\work\images\myIcon.png'`

`InstallerLogo` — Path to installer image
character vector | string scalar

Path to an image file used as the installer's logo, specified as a character vector or a string scalar. The logo will be resized to 112 pixels by 290 pixels.

The default path is:

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

Example: `'InstallerLogo','D:\Documents\MATLAB\work\images\myLogo.png'`

`InstallerName` — Name of installer file
`MyAppInstaller` (default) | character vector | string scalar

Name of the installer file, specified as a character vector or a string scalar. The extension is determined by the operating system in which the function is executed.

Example: `'InstallerName','MagicSquare_Installer'`

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

Path to an image file used as the installer's splash screen, specified as a character vector or a string scalar. The splash screen icon will be resized to 400 pixels by 400 pixels.

The default path is:

`'`*`matlabroot`*`\toolbox\compiler\packagingResources\default_splash.png'`

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

`OutputDir` — Path to folder where the installer will be saved
character vector | string scalar

Path to folder where the installer is saved, specified as a character vector or a string scalar.

If no path is specified, the default path for each operating system is:

Operating System	Default Installation Directory
Windows	`.\`*`appName`*`installer`

Operating System	Default Installation Directory
Linux	`./`*`appName`*`installer`
macOS	`./`*`appName`*`installer`

The `.` in the directories listed above represents the present working directory.

Example: `'OutputDir','D:\Documents\MATLAB\work\MagicSquare'`

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

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

- `'web'`—Option for installer to download MATLAB Runtime from MathWorks website during application installation. This is the default option.
- `'installer'`—Option to include MATLAB Runtime within the installer so that it can be installed during application installation without connecting to the MathWorks website. Use this option if you think your end-user may not have access to the Internet.

Example: `'RuntimeDelivery','installer'`

Data Types: `char` | `string`

`Shortcut` — Path to shortcut
`''` (default) | character vector | string scalar

Path to a file or folder that the installer will create a shortcut to at install time, specified as a character vector or a string scalar.

Example: `'Shortcut','.\mymagic.exe'`

Data Types: `char` | `string`

`Summary` — Summary description of application
`''` (default) | character vector | string scalar

Summary description of the application, specified as a character vector or a string scalar.

Example: `'Summary','Generates a magic square.'`

Data Types: `char` | `string`

`Version` — Version of installed application
`'1.0'` (default) | character vector | string scalar

Version number of the installed application, specified as a character vector or a string scalar.

Example: `'Version','2.0'`

Data Types: `char` | `string`

Version History

Introduced in R2020a

See Also

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

compiler.package.InstallerOptions

Options for creating MATLAB Compiler package installers

Syntax

```
opts = compiler.package.InstallerOptions(results)
opts = compiler.package.InstallerOptions(results,Name,Value)
opts = compiler.package.InstallerOptions('ApplicationName',appName)
opts = compiler.package.InstallerOptions('ApplicationName',appName,
Name,Value)
```

Description

`opts = compiler.package.InstallerOptions(results)` creates a default `InstallerOptions` object `opts` using the `compiler.build.Results` object `results` generated from a `compiler.build` function. The `InstallerOptions` object is passed as an input to the `compiler.package.installer` function.

`opts = compiler.package.InstallerOptions(results,Name,Value)` creates an `InstallerOptions` object `opts` using the `compiler.build.Results` object `results` with additional options specified using one or more name-value arguments. The `InstallerOptions` object is passed as an input to the `compiler.package.installer` function.

`opts = compiler.package.InstallerOptions('ApplicationName',appName)` creates a default `InstallerOptions` object `opts` with application name specified by `appName`. The `InstallerOptions` object is passed as an input to the `compiler.package.installer` function.

`opts = compiler.package.InstallerOptions('ApplicationName',appName, Name,Value)` creates an `InstallerOptions` object `opts` with application name specified by `appName` and additional customizations specified by name-value arguments. The `InstallerOptions` object is passed as an input to the `compiler.package.installer` function.

Examples

Create an Installer Options Object Using Results

Create an `InstallerOptions` object using the results from the `compiler.build.standaloneApplication` function and additional options specified as name-value arguments.

For this example, build a standalone application using the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

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

Create an `InstallerOptions` object. Use name-value arguments to specify the application name, author company, author name, installer name, and summary.

```
opts = compiler.package.InstallerOptions(results,...
    'ApplicationName','MagicSquare_Generator',...
```

```
    'AuthorCompany','Boston Common',...
    'AuthorName','Frog',...
    'InstallerName','MagicSquare_Installer',...
    'Summary','Generates a magic square.')

opts =

  InstallerOptions with properties:

          RuntimeDelivery: 'web'
          InstallerSplash: 'C:\Program Files\MATLAB\R2023a\toolbox\compiler\packagingResources\default_splash.png'
            InstallerIcon: 'C:\Program Files\MATLAB\R2023a\toolbox\compiler\packagingResources\default_icon_48.png'
            InstallerLogo: 'C:\Program Files\MATLAB\R2023a\toolbox\compiler\packagingResources\default_logo.png'
          AdditionalFiles: {}
    AddRemoveProgramsIcon: 'C:\Program Files\MATLAB\R2023a\toolbox\compiler\packagingResources\default_icon_48.png'
               AuthorName: 'Frog'
              AuthorEmail: ''
            AuthorCompany: 'Boston Common'
                  Summary: 'Generates a magic square.'
              Description: ''
        InstallationNotes: ''
                 Shortcut: 'D:\Work\magicsquarestandaloneApplication\magicsquare.exe'
                  Version: '1.0'
            InstallerName: 'MagicSquare_Installer'
          ApplicationName: 'MagicSquare_Generator'
                OutputDir: '.\MagicSquare_Generatorinstaller'
   DefaultInstallationDir: 'C:\Program Files\MagicSquare_Generator'
```

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

```
opts.InstallerName = 'MyMagicInstaller'
```

To create an installer for the standalone application, use the `Results` and `InstallerOptions` objects as inputs to the `compiler.package.installer` function.

```
compiler.package.installer(results,'Options',opts);
```

The function generates an installer named `MagicSquare_Installer` within a folder named `MagicSquare_Generatorinstaller`.

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

```
opts.InstallationNotes = 'Windows installer for MagicSquare'
```

Create an Installer Options Object Using Application Name

Create an `InstallerOptions` object with an application name and additional options specified as name-value arguments.

```
opts = compiler.package.InstallerOptions('ApplicationName','MagicSquare_Generator',...
    'AuthorCompany','Boston Common',...
    'AuthorName','Frog',...
    'InstallerName','MagicSquare_Installer',...
    'Summary','Generates a magic square.')

opts =

  InstallerOptions with properties:

          RuntimeDelivery: 'web'
          InstallerSplash: 'C:\Program Files\MATLAB\R2023a\toolbox\compiler\packagingResources\default_splash.png'
            InstallerIcon: 'C:\Program Files\MATLAB\R2023a\toolbox\compiler\packagingResources\default_icon_48.png'
            InstallerLogo: 'C:\Program Files\MATLAB\R2023a\toolbox\compiler\packagingResources\default_logo.png'
          AdditionalFiles: {}
    AddRemoveProgramsIcon: ''
               AuthorName: 'Frog'
              AuthorEmail: ''
            AuthorCompany: 'Boston Common'
```

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

Input Arguments

`results` — Build results object

`Results` object

Build results, specified as a `compiler.build.Results` object. Create the `Results` object by saving the output from a `compiler.build` function.

`appName` — Name of the installed application

character vector | string scalar

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

Example: `'MagicSquare_Generator'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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

`AdditionalFiles` — Additional files

character vector | string scalar | cell array of character vectors | string array

Additional files and folders to be installed with the application, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute.

Example: `'AdditionalFiles',["myimage.png","data.mat"]`

Data Types: `char` | `string` | `cell`

`AddRemoveProgramsIcon` — Add or remove programs icon

character vector | string scalar | string array

Add or remove programs icon, specified as a character vector or string scalar. The image is used as the icon in the list of Windows applications within settings. Paths can be relative to the current working directory or absolute.

The default path is:

`'`*`matlabroot`*`\toolbox\compiler\packagingResources\default_icon_48.png'`

Example: `'AddRemoveProgramsIcon',"win_icon.png"`

Data Types: `char` | `string`

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

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

Example: `'ApplicationName','MagicSquare_Generator'`

Data Types: `char` | `string`

`AuthorCompany` — Company name
`''` (default) | character vector | string scalar

Name of company that created the application, specified as a character vector or a string scalar.

Example: `'AuthorCompany','Boston Common'`

Data Types: `char` | `string`

`AuthorEmail` — Email address
`''` (default) | character vector | string scalar

Email address of the application author, specified as a character vector or a string scalar.

Example: `'AuthorEmail','frog@example.com'`

Data Types: `char` | `string`

`AuthorName` — Name
`''` (default) | character vector | string scalar

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

Example: `'AuthorName','Frog'`

Data Types: `char` | `string`

`DefaultInstallationDir` — Default installation path
character vector | string scalar

Default directory where you want the installer to install the application, specified as a character vector or a string scalar.

If no path is specified, the default path for each operating system is:

Operating System	Default Installation Directory
Windows	`C:\Program Files\`*`appName`*
Linux	`/usr/`*`appName`*
macOS	`/Applications/`*`appName`*

Example: `'DefaultInstallationDir','C:\Users\MW_Programs\MagicSquare_Generator'`

Data Types: `char` | `string`

`Description` — Detailed application description
`''` (default) | character vector | string scalar

Detailed description of the application, specified as a character vector or a string scalar.

Example: `'Description','The MagicSquare_Generator application generates an n-by-n matrix constructed from the integers 1 through n2 with equal row and column sums.'`

Data Types: `char` | `string`

`InstallationNotes` — Notes
`''` (default) | character vector | string scalar

Notes about additional requirements for using application, specified as a character vector or a string scalar.

Example: `'InstallationNotes','This is a Linux installer.'`

Data Types: `char` | `string`

`InstallerIcon` — Path to icon image
character vector | string scalar

Path to an image file used as the icon for the installed application, specified as a character vector or a string scalar.

The default path is:

`'`*`matlabroot`*`\toolbox\compiler\packagingResources\default_icon_48.png'`

Example: `'InstallerIcon','D:\Documents\MATLAB\work\images\myIcon.png'`

`InstallerLogo` — Path to installer image
character vector | string scalar

Path to an image file used as the installer's logo, specified as a character vector or a string scalar. The logo will be resized to 112 pixels by 290 pixels.

The default path is:

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

Example: `'InstallerLogo','D:\Documents\MATLAB\work\images\myLogo.png'`

`InstallerName` — Name of installer file
`MyAppInstaller` (default) | character vector | string scalar

Name of the installer file, specified as a character vector or a string scalar. The extension is determined by the operating system in which the function is executed.

Example: `'InstallerName','MagicSquare_Installer'`

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

Path to an image file used as the installer's splash screen, specified as a character vector or a string scalar. The splash screen icon will be resized to 400 pixels by 400 pixels.

The default path is:

`'`*`matlabroot`*`\toolbox\compiler\packagingResources\default_splash.png'`

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

`OutputDir` — Path to folder where the installer will be saved
character vector | string scalar

Path to folder where the installer is saved, specified as a character vector or a string scalar.

If no path is specified, the default path for each operating system is:

Operating System	Default Installation Directory
Windows	`.\`*`appName`*`installer`
Linux	`./`*`appName`*`installer`
macOS	`./`*`appName`*`installer`

The `.` in the directories listed above represents the present working directory.

Example: `'OutputDir','D:\Documents\MATLAB\work\MagicSquare'`

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

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

- `'web'`—Option for installer to download MATLAB Runtime from MathWorks website during application installation. This is the default option.
- `'installer'`—Option to include MATLAB Runtime within the installer so that it can be installed during application installation without connecting to the MathWorks website. Use this option if you think your end-user may not have access to the Internet.

Example: `'RuntimeDelivery','installer'`

Data Types: `char` | `string`

`Shortcut` — Path to shortcut
`''` (default) | character vector | string scalar

Path to a file or folder that the installer will create a shortcut to at install time, specified as a character vector or a string scalar.

Example: `'Shortcut','.\mymagic.exe'`

Data Types: `char` | `string`

`Summary` — Summary description of application
`''` (default) | character vector | string scalar

Summary description of the application, specified as a character vector or a string scalar.

Example: `'Summary','Generates a magic square.'`

Data Types: `char` | `string`

`Version` — Version of installed application
`'1.0'` (default) | character vector | string scalar

Version number of the installed application, specified as a character vector or a string scalar.

Example: `'Version','2.0'`

Data Types: `char` | `string`

Output Arguments

`opts` — Installer options object
`InstallerOptions` object

Installer options, returned as an `InstallerOptions` object.

Version History

Introduced in R2020a

See Also

`compiler.package.installer` | `mcc`

createDeploymentScript

Create a deployment script from a MATLAB Compiler PRJ file

Syntax

```
createDeploymentScript(prjfile,outputfile)
```

Description

`createDeploymentScript(prjfile,outputfile)` creates a deployment script file by obtaining the information from a MATLAB Compiler project file `prjfile` and generating a MATLAB script file at the location defined by `outputfile`.

Examples

Create Deployment Script

Create a deployment script named `deployMyMagic` using the project file `mymagic.prj`.

Create a MATLAB Compiler project using a `deploytool` app, such as the **Library Compiler**. Add a main file and save the project as `mymagic.prj`.

Save the compiler settings from `mymagic.prj` in a deployment script named `deployMyMagic` using `createDeploymentScript`.

```
createDeploymentScript("mymagic.prj", "deployMyMagic.m");
```

Input Arguments

`prjfile` — Path to MATLAB Compiler or MATLAB Compiler SDK project file

Path to the MATLAB Compiler or MATLAB Compiler SDK project file, specified as a character vector or string scalar.

Example: `"mymagic.prj"`

Data Types: `char` | `string`

`outputfile` — Path to file to write as deployment script

Path to the file to write as the deployment script, specified as a character vector or string scalar. The file must not exist and must include either the extension `.m` or no extension. If you do not specify an extension, `.m` is appended. If you do not specify a full path, `outputfile` is saved in the current folder.

Example: `"deployMyMagic.m"`

Data Types: `char` | `string`

Version History

Introduced in R2022b

See Also

`deploytool` | `mcc` | `compiler.build.Results`

ctfroot

Location of files related to deployed application

Syntax

```
root = ctfroot
```

Description

`root = ctfroot` returns the name of the folder where the deployable archive for the application is expanded.

Use this function to access any file that the user would have included in their project (excluding the ones in the packaging folder).

Examples

Determine location of deployable archive

```
appRoot = ctfroot;
```

Output Arguments

`root` — Path to expanded deployable archive

character vector

Path to expanded deployable archive returned as a character vector in the form: *`application_name`*`_mcr`. .

Version History

Introduced in R2006a

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 PNG 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'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 | 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
- `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.

Version History

Introduced in R2009b

getmcruserdata

Retrieve MATLAB array value associated with a given key

Syntax

```
value = getmcruserdata(key)
```

Description

`value = getmcruserdata(key)` returns MATLAB data associated with the string `key` in the current MATLAB Runtime instance. If there is no data associated with the key, it returns an empty matrix.

This function is part of the MATLAB Runtime User Data interface API. It is available both in MATLAB and in deployed applications created with MATLAB Compiler and MATLAB Compiler SDK.

Examples

Get the magic square data associated with the string `'magic'` in the current instance of the MATLAB Runtime.

```
value = magic(3);
setmcruserdata('magic', value);
getmcruserdata('magic')
```

```
ans =
     8     1     6
     3     5     7
     4     9     2
```

Input Arguments

key — Key associated with MATLAB data

string

`key` is the MATLAB string with which MATLAB data `value` is associated within the current instance of the MATLAB Runtime.

Output Arguments

value — Value of MATLAB data

any MATLAB data type including matrices, cell arrays, and Java objects

`value` is the MATLAB data associated with input string `key` for the current instance of the MATLAB Runtime.

Version History

Introduced in R2008b

See Also

`setmcruserdata`

isdeployed

Determine whether code is running in deployed or MATLAB mode

Syntax

```
x = isdeployed
```

Description

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

An application running in deployed mode consists of a collection of MATLAB functions and data packaged using MATLAB Compiler SDK into software components that run outside a MATLAB session using MATLAB Runtime libraries.

Examples

Access Files from Deployed Functions

You can access files included in a packaged application by using the `which` function, or by specifying the file location relative to `ctfroot`.

Add a file such as `extern_app.exe` to your MATLAB Compiler SDK project.

Check if the code is running in deployed mode by using `isdeployed`. Then, obtain the path to the file by using the `which` function.

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

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

For more information, see “Access Files in Packaged Applications” on page 2-9.

Protect Use of `ADDPATH`

The path of a deployed application is fixed at compile time and cannot change. Use `isdeployed` to ensure that the application does not attempt to use path modifying functions, such as `addpath`, after deployment.

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

Display Documentation

You cannot use the `doc` function to open the Help browser from a deployed application. Instead, redirect a help query to the MathWorks website.

```
if ~isdeployed
    doc(mfile);
else
   web('https://www.mathworks.com/support.html');
end
```

Suppress Warnings for Non-Deployable Function

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

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

The pragma excludes the function from dependency analysis during compilation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Returns true and false as appropriate for MEX targets.
- Returns false for SIM targets, which you should query using `coder.target`.
- Returns false for other targets.

See Also

`ismcc` | `mcc` | `%#exclude` | `deploytool`

Topics

"Write Deployable MATLAB Code"
"Access Files in Packaged Applications" on page 2-9

ismcc

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

Syntax

```
x = ismcc
```

Description

`x = ismcc` returns true when the function is being executed by `mcc` dependency checker and false otherwise.

When this function is executed by the compilation process started by `mcc` that runs outside of MATLAB in a system command prompt, it will return true. This function will return false when executed within MATLAB as well as in deployed mode. To test for deployed mode execution, use `isdeployed`. This function must be used in `matlabrc` or `hgrc` (or any function called within them, for example `startup.m`) to guard code from being executed by MATLAB Compiler (`mcc`) or MATLAB Compiler SDK.

In a typical example, a user has `ADDPATH` calls in their MATLAB code. These can be guarded from executing using `ismcc` during the compilation process and `isdeployed` for the deployed application in `startup.m`, as shown in the example on this page.

Examples

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

See Also

`isdeployed` | `mcc`

libraryCompiler

Open the Library Compiler app

Syntax

```
libraryCompiler
libraryCompiler project_name
```

Description

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

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

Examples

Create a New Project

Open the Library Compiler app to create a new project.

```
libraryCompiler
```

Input Arguments

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

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

Version History

Introduced in R2013b

R2020a: `-build` and `-package` options will be removed
Warns starting in R2020a

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

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`.
`-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).

Option	Description
`-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.
`-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). For the MinGW-w64 compiler, which is based on gcc/g++, use single quotes (`'`).

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.

Version History

Introduced before R2006a

mcrinstaller

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

Syntax

```
[installer_path, major, minor, platform] = mcrinstaller
```

Description

`[installer_path, major, minor, platform] = mcrinstaller` displays information about available MATLAB Runtime installers.

If no MATLAB Runtime installer is found, you are prompted to download an installer using the command `compiler.runtime.download`.

You must distribute the MATLAB Runtime library to your end users to enable them to run applications developed with MATLAB Compiler or MATLAB Compiler SDK.

For more information about the MATLAB Runtime installer, see “Install and Configure MATLAB Runtime” on page 11-2.

Examples

Find MATLAB Runtime Installer Location

Display the location of MATLAB Runtime installers for a particular platform. This example shows output for a `win64` system. The release number is called `R20xxx` indicating the release for which the MATLAB Runtime installer has been downloaded.

```
mcrinstaller
```

```
C:\Program Files\MATLAB\R20xxx\toolbox\compiler\deploy\win64\MCR_R20xxx_win64_installer.exe
```

For example, for R2018b, the path would be:

```
C:\Program Files\MATLAB\R2018b\toolbox\compiler\deploy\win64\MCR_R2018b_win64_installer.exe
```

Output Arguments

`installer_path` — Full path to the installer
character vector

The `installer_path` is the full path to the installer for the current platform.

`major` — Major version number
positive integer scalar

The `major` is the major version number of the installer.

`minor` — Minor version number
positive integer scalar

The `minor` is the minor version number of the installer.

`platform` — Name of the current platform
character vector

The `platform` is the name of the current platform (returned by `COMPUTER(arch)`).

See Also

`mcrversion` | `compiler.runtime.download`

Topics
“Install and Configure MATLAB Runtime” on page 11-2

mcrversion

Return MATLAB Runtime version number that matches MATLAB version

Syntax

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

Description

`[major,minor] = mcrversion` returns the MATLAB Runtime version number matching the version of MATLAB from where the command is executed. The MATLAB Runtime version number consists of two digits, separated by a decimal point. This function returns each digit as a separate output variable: `major`, `minor`.

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

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

At this time, all outputs past “minor” are returned as zeros.

Examples

Return the MATLAB Runtime Version

Return the MATLAB Runtime Version Number Matching the Version of MATLAB.

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

```
major =
     9
minor =
     9
```

Output Arguments

`major` — Major version number
positive integer scalar

Major version number returned as a positive integer scalar.

Data Types: `double`

`minor` — Minor version number
positive integer scalar

Minor version number returned as a positive integer scalar.

Data Types: `double`

See Also

`compiler.runtime.download` | `mcrinstaller`

Topics

“Install and Configure MATLAB Runtime” on page 11-2

productionServerCompiler

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

Syntax

```
productionServerCompiler
productionServerCompiler project_name
```

Description

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

`productionServerCompiler project_name` opens the Production Server Compiler app with the project preloaded.

Examples

Create a New Production Server Project

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

```
productionServerCompiler
```

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.

Version History

Introduced in R2014a

R2020a: `-build` and `-package` options will be removed
Warns starting in R2020a

The `-build` and `-package` options will be removed. To generate deployable archives, use the `compiler.build.productionServerArchive` function, or the `mcc` command, or the **Production Server Compiler** app.

setmcruserdata

Associate MATLAB data value with a key

Syntax

```
void setmcruserdata(key, value)
```

Description

`void setmcruserdata(key, value)` associates the MATLAB data `value` with the string `key` in the current MATLAB Runtime instance. If there is already a `value` associated with the `key`, it is overwritten.

This function is part of the MATLAB Runtime User Data interface API. It is available both in MATLAB and in deployed applications created with MATLAB Compiler and MATLAB Compiler SDK.

Examples

Store a cell array and associate it with the string `'PI_Data'` in the current instance of the MATLAB Runtime.

```
value = {3.14159, 'March 14th is PI day'};
setmcruserdata('PI_Data', value);
```

Input Arguments

`value` — Value of MATLAB data

any MATLAB data type including matrices, cell arrays, and Java objects

`Value` is the MATLAB data associated with input string `key` for the current instance of the MATLAB Runtime.

`key` — Key associated with MATLAB data

string

`key` is a MATLAB string with which MATLAB data `value` is associated within the current instance of the MATLAB Runtime.

Version History

Introduced in R2008a

See Also

`getmcruserdata`

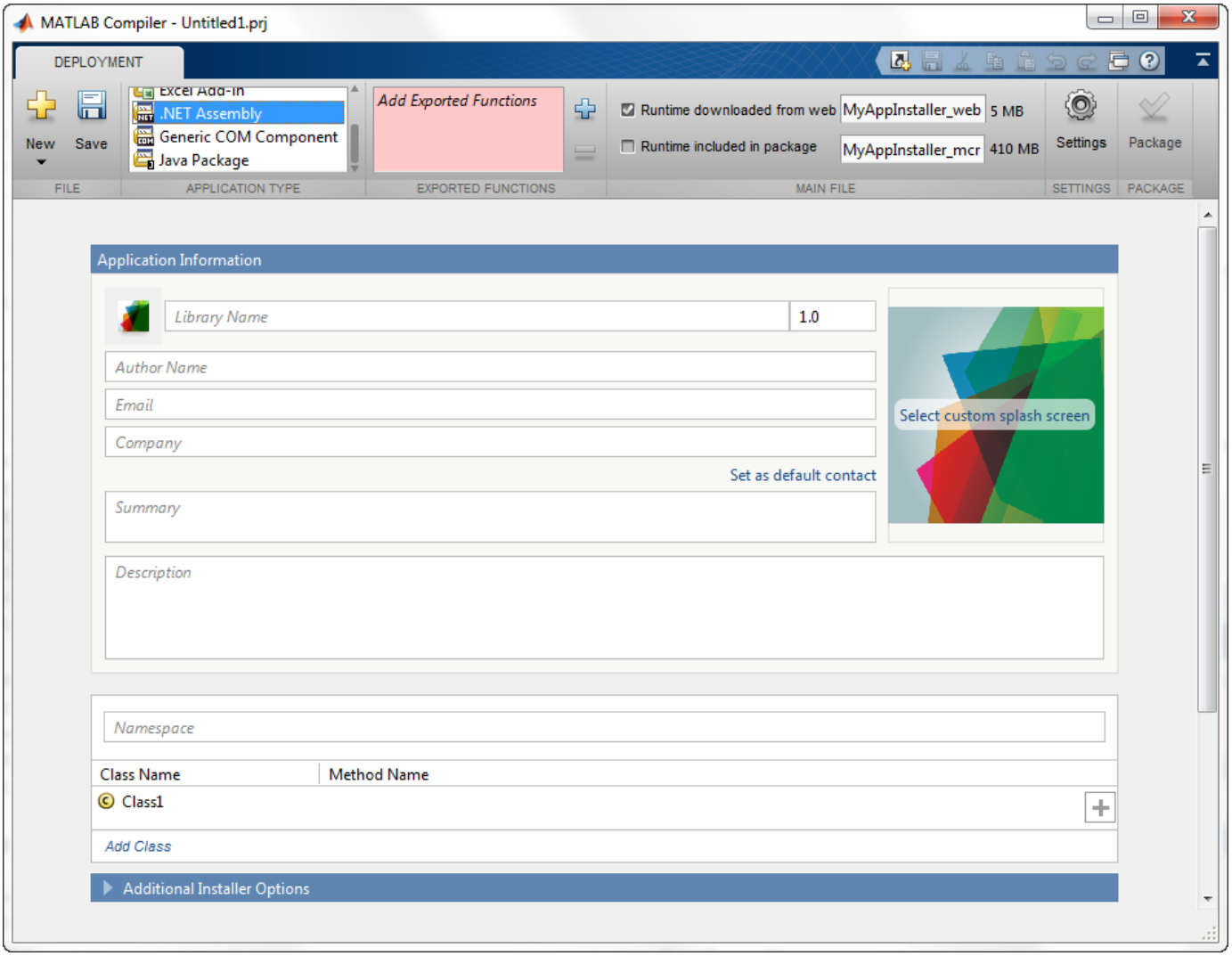
Apps

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 Excel Add-In from MATLAB”
- “Create a C Shared Library with MATLAB Code”
- “Generate a C++ mwArray API Shared Library and Build a C++ Application”
- “Generate a C++ MATLAB Data API Shared Library and Build a C++ Application”
- “Generate .NET Assembly and Build .NET Application”
- “Create a Generic COM Component with MATLAB Code”
- “Generate Java Package and Build Java Application”
- “Generate a Python Package and Build a Python Application”

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

You can decide whether or not to include the MATLAB Runtime fallback for MATLAB Runtime installer in the generated application by selecting one of the two options in the **Packaging Options** section. Including the MATLAB Runtime installer in the package significantly increases the size of the package.

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

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

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

`files required for your 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

Enter `libraryCompiler`.

Alternatively, enter `deploytool` and click **Library Compiler**.

See Also

`deploytool` | `mcc`

Topics

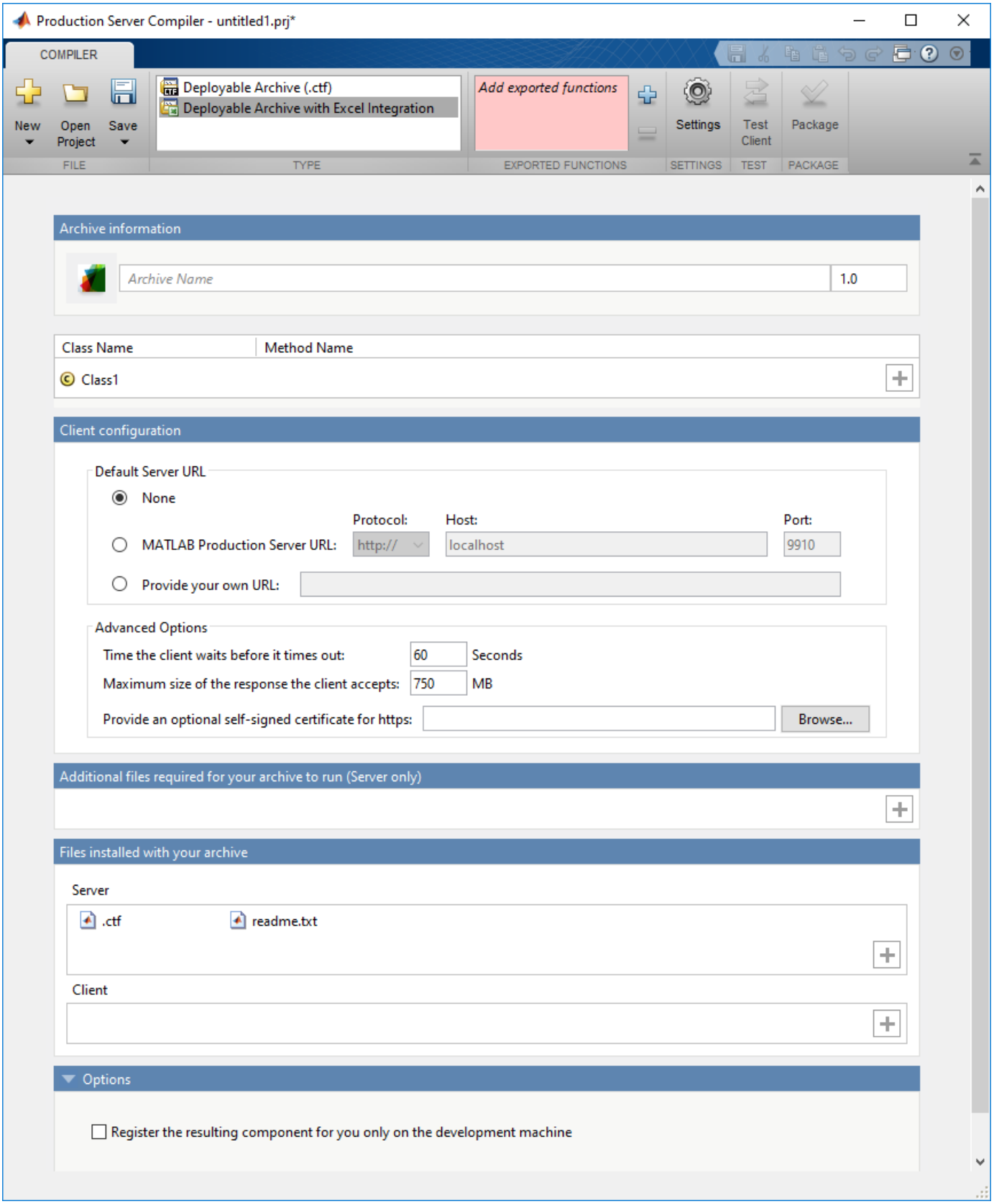
“Create Excel Add-In from MATLAB”
“Create a C Shared Library with MATLAB Code”
“Generate a C++ mwArray API Shared Library and Build a C++ Application”
“Generate a C++ MATLAB Data API Shared Library and Build a C++ Application”
“Generate .NET Assembly and Build .NET Application”
“Create a Generic COM Component with MATLAB Code”
“Generate Java Package and Build Java Application”
“Generate a Python Package and Build a Python Application”

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.


Production Server Compiler - untitled1.prj*
COMPILER
New
Open Project
Save
FILE
Deployable Archive (.ctf)
Deployable Archive with Excel Integration
TYPE
Add exported functions
EXPORTED FUNCTIONS
Settings
SETTINGS
Test Client
TEST
Package
PACKAGE
Archive information
Archive Name
1.0
Class Name
Method Name
Class1
Client configuration
Default Server URL
None
Protocol:
Host:
Port:
MATLAB Production Server URL:
http://
localhost
9910
Provide your own URL:
Advanced Options
Time the client waits before it times out:
60
Seconds
Maximum size of the response the client accepts:
750
MB
Provide an optional self-signed certificate for https:
Browse...
Additional files required for your archive to run (Server only)
Files installed with your archive
Server
.ctf
readme.txt
Client
Options
Register the resulting component for you only on the development machine

Open the Production Server Compiler App

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

Examples

- “Create Deployable Archive for MATLAB Production Server”
- “Create and Install a Deployable Archive with Excel Integration for MATLAB Production Server”
- “Test Client Data Integration Against MATLAB”

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

Enter `productionServerCompiler`.

Alternatively, enter `deploytool` and click **Production Server Compiler**.

Version History

Introduced in R2013b

See Also

`deploytool` | `compiler.build.productionServerArchive` | `mcc`

Topics

“Create Deployable Archive for MATLAB Production Server”
“Create and Install a Deployable Archive with Excel Integration for MATLAB Production Server”
“Test Client Data Integration Against MATLAB”