# MATLAB® Compiler SDK™

MATLAB® Code Deployment Guide

MATLAB®

**R**2022**a**

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) |

# Contents

# 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 1-4.
2. Validates MEX-files. In particular, `mexFunction` entry points are verified.

   For more details about MEX-file processing, see “MEX-Files, DLLs, or Shared Libraries” on page 1-3.
3. Creates a deployable archive from the input files and their dependencies.

   For more details about deployable archives, see “Deployable Archive” on page 1-6.
4. Generates target-specific wrapper code.
5. Generates target-specific binary package.

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

## MEX-Files, DLLs, or Shared Libraries

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

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

# Dependency Analysis Using MATLAB Compiler

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

`mcc` | `applicationCompiler` | `compiler.build.standaloneApplication`

## More About

- Application Compiler

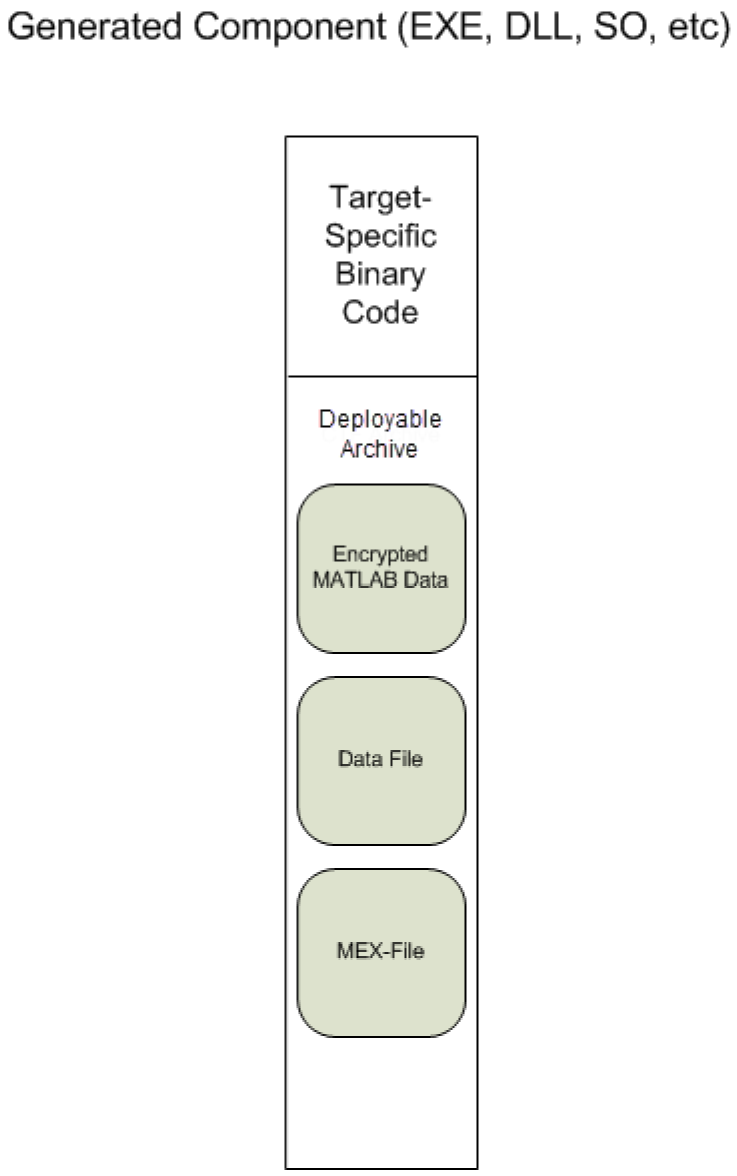
## Deployable Archive

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

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

**Information on Deployable Archive Embedding/Extraction and Component Cache**

| Product | Refer to |
|---|---|
| MATLAB Compiler SDK C/C++ integration | "MATLAB Runtime Component Cache and Deployable Archive Embedding" |
| MATLAB Compiler SDK .NET integration | "MATLAB Runtime Component Cache and Deployable Archive Embedding" |
| MATLAB Compiler SDK Java integration | "Define Embedding and Extraction Options for Deployable Java Archive" |
| MATLAB Compiler Excel® integration | "MATLAB Runtime Component Cache and Deployable Archive Embedding" |


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

## Additional Details

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

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

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

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

# Write Deployable MATLAB Code

## Write Deployable MATLAB Code

**In this section...**

### Packaged Applications Require Functions

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

### Packaged Applications Do Not Process MATLAB Files at Run Time

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

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

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

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

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

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

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

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

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

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

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

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

## Use isdeployed Functions To Execute Deployment-Specific Code Paths

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

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

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

## Gradually Refactor Applications That Depend on Noncompilable Functions

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

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

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

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

### Do Not Create or Use Nonconstant Static State Variables

Avoid using the following:

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

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

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

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

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

### Get Proper Licenses for Toolbox Functionality You Want to Deploy

You must have a valid MathWorks® license for toolboxes you use to create deployable MATLAB code.

### See Also
`isdeployed` | `ismcc`

### More About
- MATLAB Compiler support for MATLAB and toolboxes

## 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"

## MATLAB Data Files in Compiled Applications

| In this section... |
|---|
| "Explicitly Including MATLAB Data files Using the %#function Pragma" on page 2-8 |
| "Load and Save Functions" on page 2-8 |

### Explicitly Including MATLAB Data files Using the %#function Pragma

The compiler excludes MATLAB data files (MAT-files) from dependency analysis by default. See "Dependency Analysis Using MATLAB Compiler" on page 1-4.

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

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

### Load and Save Functions

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

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

For more information about deployable archives, see "Deployable Archive" on page 1-6.

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

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

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

The following example specifies three MATLAB data files:

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

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

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

**ex_loadsave.m**

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

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

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

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

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

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

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

## Share MATLAB Runtime Instances

**In this section...**

### What Is a Singleton MATLAB Runtime?

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

### Advantages and Disadvantages of Using a Singleton

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

#### When You Should Use a Singleton

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

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

#### When You Might Avoid Using a Singleton

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

# 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 a C Shared Library with MATLAB Code

**Supported platform:** Windows, Linux, Mac

This example shows how to create a C shared library using a MATLAB function. The target system does not require a licensed copy of MATLAB.

## Create Functions in MATLAB

1. In MATLAB, examine the MATLAB code that you want packaged.

   For this example, Copy the `matrix` folder that ships with MATLAB to your work folder.

   ```
   copyfile(fullfile(matlabroot,'extern','examples','compilersdk','c_cpp','matrix'),'matrix')
   ```

   Navigate to the new `matrix` subfolder in your work folder.
2. Examine and test the functions `addmatrix.m`, `multiplymatrix.m`, and `eigmatrix.m`.

   **`addmatrix.m`**

   ```
   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
   ```

   **`multiplymatrix.m`**

   ```
   function m = multiplymatrix(a1, a2)

   m =  a1*a2;
   ```

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

   The output is:

   ```
    ans =
       30    66   102
       36    81   126
       42    96   150
   ```

   **`eigmatrix.m`**

   ```
   function e = eigmatrix(a1)

       try
           %Tries to calculate the eigenvalues and return them.
           e = eig(a1);
   ```

```
    catch
        %Returns a -1 on error.
        e = -1;
end
```

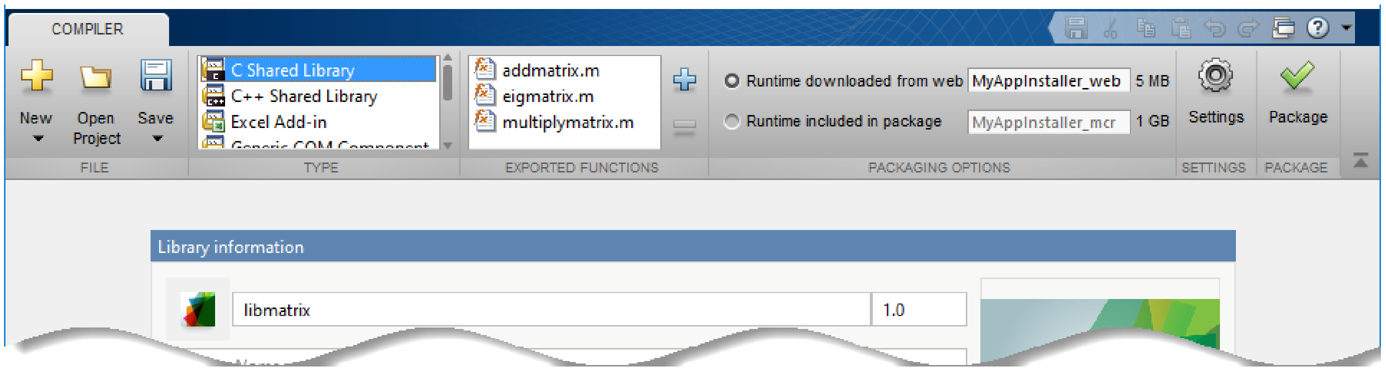
At the MATLAB command prompt, enter `eigmatrix([1 4 7; 2 5 8; 3 6 9])`.

The output is:

```
 ans =
   16.1168
   -1.1168
   -0.0000
```

## Create a C Shared Library Using the Library Compiler App

1. On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Library Compiler**. In the **MATLAB Compiler** project window, click **C Shared Library**.



   Alternately, you can open the **Library Compiler** app by entering `libraryCompiler` at the MATLAB prompt.
2. In the **Library Compiler** app project window, specify the files of the MATLAB application that you want to deploy.

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

   **b** In the **Add Files** window, browse to the example folder, and select the function you want to package. Click **Open**.

   The function is added to the list of exported function files. Repeat this step to package multiple files in the same application.

   Add all three functions to the list of main files.
3. In the **Packaging Options** section of the toolstrip, decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the options:

   - **Runtime downloaded from web** — Generate an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application. You can specify the file name of the installer.
   - **Runtime included in package** — Generate an application that includes the MATLAB Runtime installer. You can specify the file name of the installer.

**Note** The first time you select this option, you are prompted to download the MATLAB Runtime installer.

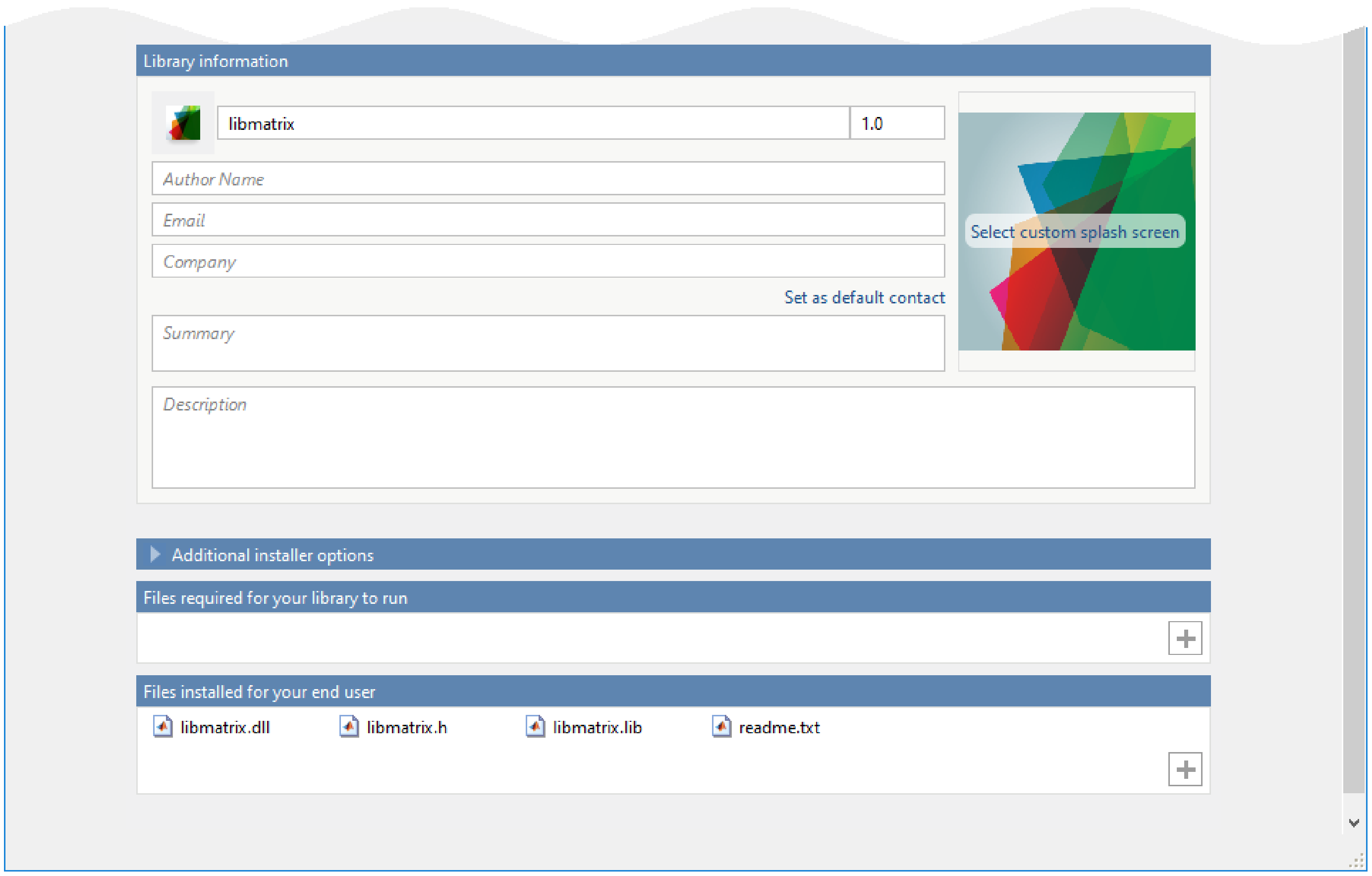
4 In the **Library Name** field, rename the packaged shared library as `libmatrix`. The same name is followed through in the implementation of the shared library.

## Customize the Application and Its Appearance

In the **Library Compiler** app, you can customize the installer, customize your application, and add more information about the application.

- **Library information** — Information about the deployed application. You can also customize the appearance of the application by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata. See “Customize the Installer”.
- **Additional installer options** — Default installation path for the generated installer and custom logo selection. See “Change the Installation Path”.
- **Files required for your library to run** — Additional files required by the generated application to run. These files are included in the generated application installer. See “Manage Required Files in Compiler Project”.
- **Files installed for your end user** — Files that are installed with your application.

  See “Specify Files to Install with Application”.


Library information

libmatrix 1.0

Author Name

Email

Company

Set as default contact

Summary

Description

Select custom splash screen

Additional installer options

Files required for your library to run

Files installed for your end user

libmatrix.dll libmatrix.h libmatrix.lib readme.txt

## Package the Application

When you are finished selecting your packaging options, save your **Library Compiler** project and generate the packaged application.

1. Click **Package**.

   In the Save Project dialog box, specify the location to save the project.
2. In the **Package** dialog box, verify that **Open output folder when process completes** is selected.

   When the packaging process is complete, examine the generated output in the target folder.

   - Three folders are generated: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`.

     For more information about the files generated in these folders, see “Files Generated After Packaging MATLAB Functions”.
   - The log file `PackagingLog.html` contains packaging results.

## Create C Shared Library Using compiler.build.cSharedLibrary

As an alternative to the Library Compiler app, you can create a C shared library using a programmatic approach. If you have already created a library using the Library Compiler, see “Implement C Shared Library in C Application”.

- Build the C shared library using the `compiler.build.cSharedLibrary` function. Use name-value arguments to specify the library name and enable verbose output.

  ```
  buildResults = compiler.build.cSharedLibrary(["addmatrix.m", ...
  "eigmatrix.m","multiplymatrix.m"], ...
  'LibraryName','libmatrix', ...
  'Verbose','on');
  ```

  You can specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.build.cSharedLibrary`.

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

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

  - `GettingStarted.html` — HTML file that contains information on integrating your shared library.
  - `includedSupportPackages.txt` — Text file that lists all support files included in the library.
  - `libmatrix.c` — C source code file.
  - `libmatrix.def` — Module-definition file that provides the linker with module information.
  - `libmatrix.dll` — Dynamic-link library file.
  - `libmatrix.exports` — Exports file that contains all nonstatic function names.
  - `libmatrix.h` — C header file.

- `libmatrix.lib` — Import library file. The file extension is `.dylib` on Mac and `.so` on UNIX.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations.
- `readme.txt` — Text file that contains packaging information.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

**Note** The generated library does not include MATLAB Runtime or an installer. To create an installer using the `buildResults` object, see `compiler.package.installer`.

## Implement C Shared Library in C Application

To implement your shared library using the provided C application code, see “Implement a C Shared Library with a Driver Application”.

## See Also

`libraryCompiler` | `compiler.build.cSharedLibrary` | `deploytool`

## More About

- “Implement a C Shared Library with a Driver Application”
- “Call a C Shared Library”
- “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”

# 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`.<br><br>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.<br><br>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. |

## 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”
- “Implement a C Shared Library with a Driver Application”

## Distribute C/C++ Shared Libraries to Application Developers

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

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

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

# Package a .NET Assembly

## Generate .NET Assembly and Build .NET Application

**Supported platform:** Windows

This example shows how to create a .NET assembly from a MATLAB function and integrate the generated assembly into a .NET application.

### Prerequisites

- Verify that you have met all of the MATLAB Compiler SDK .NET target requirements. For details, see “MATLAB Compiler SDK .NET Target Requirements”.
- Verify that you have Microsoft Visual Studio® installed.
- End users must have an installation of MATLAB Runtime to run the application. For details, see “Install and Configure MATLAB Runtime”.

  For testing purposes, you can use an installation of MATLAB instead of MATLAB Runtime.

### Files

| | |
|---|---|
| **MATLAB Function** | `*matlabroot*\toolbox\dotnetbuilder\Examples\VS*Version*\NET \MagicSquareExample\MagicSquareComp\makesquare.m` |
| **MWArray API Reference** | `*matlabroot*\help\dotnetbuilder\MWArrayAPI` |

### Create Function in MATLAB

In MATLAB, examine the MATLAB code that you want to package. For this example, open `makesquare.m`.

```
function y = makesquare(x)
y = magic(x);
```

At the MATLAB command prompt, enter `makesquare(5)`.

The output is a 5-by-5 matrix.

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

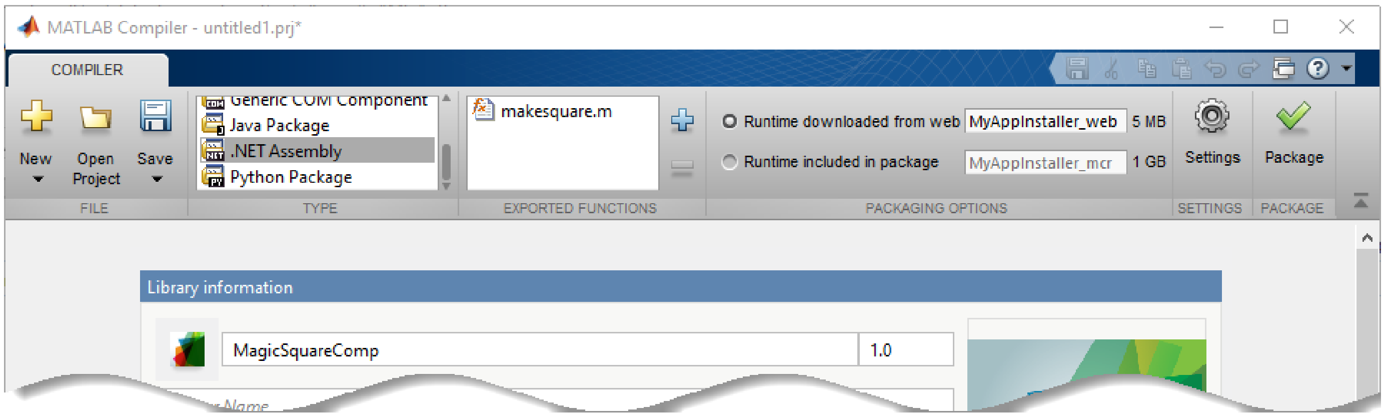
### Create .NET Assembly Using Library Compiler App

Package the function into a .NET assembly using the **Library Compiler** app. Alternatively, if you want to create a .NET assembly from the MATLAB command window using a programmatic approach, see “Create .NET Assembly Using compiler.build.dotNETAssembly”.

1 On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Library Compiler**.

Alternatively, you can open the **Library Compiler** app from the MATLAB command prompt.

```
libraryCompiler
```



2 In the **Type** section of the toolstrip, click **.NET Assembly**.

In the **Library Compiler** app project window, specify the files of the MATLAB application that you want to deploy.

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

**b** In the **Add Files** window, browse to the example folder, and select the function you want to package. Click **Open**.

The function is added to the list of exported function files. Repeat this step to package multiple files in the same application.

For this example, select the file `makesquare.m`.

3 In the **Packaging Options** section of the toolstrip, decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the options:

- **Runtime downloaded from web** — Generate an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application. You can specify the file name of the installer.
- **Runtime included in package** — Generate an application that includes the MATLAB Runtime installer. You can specify the file name of the installer.

**Note** The first time you select this option, you are prompted to download the MATLAB Runtime installer.

## Specify Assembly File Settings

Next, define the name of your assembly and verify the class mapping for the `.m` file that you are building into your application.

1 The **Library Name** field is automatically populated with `makesquare` as the name of the assembly. Rename it as `MagicSquareComp`. The same name is followed through in the implementation of the assembly.

2 Verify that the function defined in `makesquare.m` is mapped into `MagicSquareClass`. Double-click on the class to change the class name.

### Create Sample Driver File

You can use any MATLAB file in the project to generate sample .NET driver files. Although .NET driver files are not necessary to create an assembly, you can use them to implement the generated assembly into a .NET application in the target language, as shown in “Integrate .NET Assembly Into .NET Application”.

In the **Samples** section, select **Create New Sample**, and click `makesquare.m`. A MATLAB file opens for you to edit.

```
% Sample script to demonstrate execution of function y = makesquare(x)
x = 0; % Initialize x here
y = makesquare(x);
```

Change `x = 0` to `x = 5`, save the file, and return to the **Library Compiler** app.

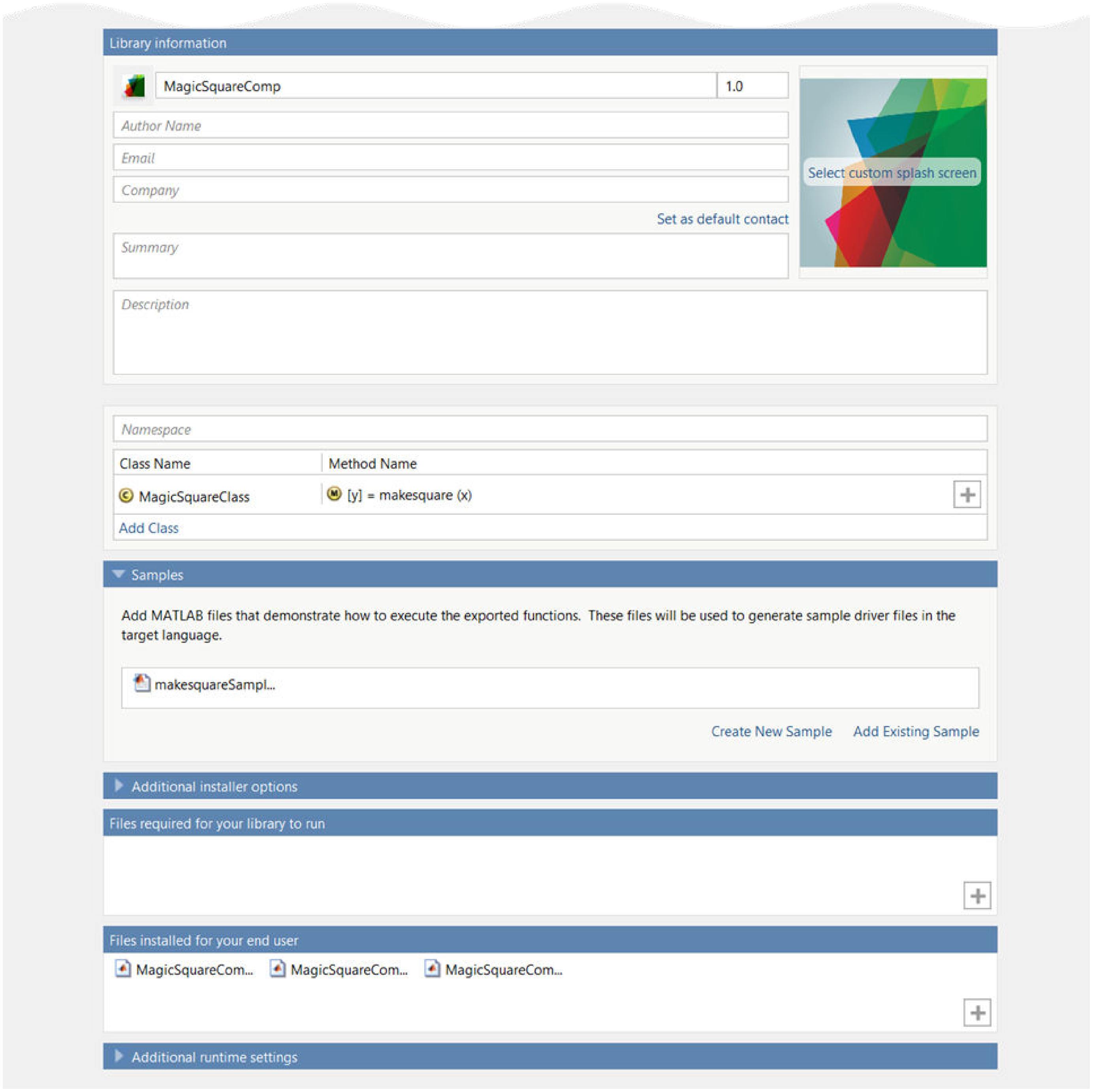
For more information and limitations, see “Sample Driver File Creation”.

### Customize Application and Its Appearance

In the **Library Compiler** app, you can customize the installer, customize your application, and add more information about the application.

- **Library information** — Information about the deployed application. You can also customize the appearance of the application by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata. See “Customize the Installer”.
- **Additional installer options** — Default installation path for the generated installer and custom logo selection. See “Change the Installation Path”.
- **Files required for your library to run** — Additional files required by the generated application to run. These files are included in the generated application installer. See “Manage Required Files in Compiler Project”.
- **Files installed for your end user** — Files that are installed with your application.

  See “Specify Files to Install with Application”.
- **Additional runtime settings** — Platform-specific options for controlling the generated executable. See “Additional Runtime Settings”.


Library information
MagicSquareComp
1.0
Author Name
Email
Company
Set as default contact
Summary
Select custom splash screen
Description
Namespace
Class Name
Method Name
MagicSquareClass
[y] = makesquare (x)
Add Class
Samples
Add MATLAB files that demonstrate how to execute the exported functions. These files will be used to generate sample driver files in the target language.
makesquareSampl...
Create New Sample
Add Existing Sample
Additional installer options
Files required for your library to run
Files installed for your end user
MagicSquareCom...
MagicSquareCom...
MagicSquareCom...
Additional runtime settings

### Package the Application

When you are finished selecting your packaging options, save your **Library Compiler** project and generate the packaged application.

**1** Click **Package**.

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

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

   When the packaging process is complete, examine the generated output in the target folder.

   - Three folders are generated: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`.

     For more information about the files generated in these folders, see “Files Generated After Packaging MATLAB Functions”.
   - The log file `PackagingLog.html` contains packaging results.

## Create .NET Assembly Using compiler.build.dotNETAssembly

As an alternative to the **Library Compiler** app, you can create a .NET assembly using a programmatic approach. If you have already created an assembly using the **Library Compiler**, see “Integrate .NET Assembly Into .NET Application”.

1. Save the path to the file `makesquare.m` located in `*matlabroot*\toolbox\dotnetbuilder\Examples\VS*Version*\NET\MagicSquareExample\MagicSquareComp`. For example, if you are using Visual Studio version 15, type:

   ```
   appFile = fullfile(matlabroot,'toolbox','dotnetbuilder','Examples', ...
       'VS15','NET','MagicSquareExample','MagicSquareComp','makesquare.m');
   ```
2. Save the following code in a sample file named `makesquareSample1.m`:

   ```
   x = 5;
   y = makesquare(x);
   ```
3. Build the .NET assembly using the `compiler.build.dotNETAssembly` function. Use name-value arguments to specify the assembly name, class name, and sample file.

   ```
   buildResults = compiler.build.dotNETAssembly(appFile, ...
   'AssemblyName','MagicSquareComp', ...
   'ClassName','MagicSquareClass', ...
   'SampleGenerationFiles','makesquareSample1.m');
   ```

   You can specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.build.dotNETAssembly`.

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

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

   - `samples\makesquareSample1.cs` — .NET sample driver file.
   - `GettingStarted.html` — HTML file that contains steps on compiling .NET driver applications from the command line.
   - `includedSupportPackages.txt` — Text file that lists all support files included in the assembly.
   - `MagicSquareComp.dll` — Dynamic-link library file that can be accessed using the `mwArray` API.
   - `MagicSquareComp.xml` — XML file that contains documentation for the `mwArray` assembly.
   - `MagicSquareComp_overview.html` — HTML file that contains requirements for accessing the assembly and for generating arguments using the `mwArray` class hierarchy.

- `MagicSquareCompNative.dll` — Dynamic-link library file that can be accessed using the native API.
- `MagicSquareCompNative.xml` — XML file that contains documentation for the native assembly.
- `MagicSquareCompVersion.cs` — C# file that contains version information.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations.
- `readme.txt` — Text file that contains packaging and interface information.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

---

**Note** The generated assembly does not include MATLAB Runtime or an installer. To create an installer using the `buildResults` object, see `compiler.package.installer`.

---

## Integrate .NET Assembly Into .NET Application

After creating your .NET assembly, you can integrate it into any .NET application. This example uses the sample .NET application code generated during packaging. You can use this sample .NET application code as a guide to write your own application.

1. Open Microsoft Visual Studio and create a C# **Console App (.NET Framework)** called `MainApp`.
2. Remove any source code files that were created within your project, if necessary.
3. Add the generated sample .NET application code `makesquareSample1.cs` in the `for_redistribution_files_only\samples` folder to the project.

   The program listing is shown below.

```
using System;
using System.Collections.Generic;
using System.Text;
using MathWorks.MATLAB.NET.Arrays;
using MathWorks.MATLAB.NET.Utility;
using MagicSquareComp;

/// <summary>
/// Sample driver code that integrates a compiled MATLAB function
/// generated by MATLAB Compiler SDK
///
/// Refer to the MATLAB Compiler SDK documentation for more
/// information.
/// </summary>
class makesquareSample1 {
    static MagicSquareClass MagicSquareClassInstance;

    static void Setup() {
        MagicSquareClassInstance = new MagicSquareClass();
    }

    /// <summary>
    /// Example of using the var function.
    /// </summary>
    public static void makesquareSample() {
        double xInData = 5.0;
        MWNumericArray yOut = null;
        Object[] results = null;
        try {
            MWNumericArray xIn = new MWNumericArray(xInData);
            results = MagicSquareClassInstance.makesquare(1, xIn);
```

```
                if (results[0] is MWNumericArray) {
                    yOut = (MWNumericArray) results[0];
                }
                Console.WriteLine(yOut);
            } catch (Exception e) {
                Console.WriteLine(e);
            }
        }

        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args) {
            try {
                Setup();
            } catch (Exception e) {
                Console.WriteLine(e);
                Environment.Exit(1);
            }
            try {
                makesquareSample();
            } catch (Exception e) {
                Console.WriteLine(e);
                Environment.Exit(1);
            }
        }
    }
```

4. In Visual Studio, add a reference to your assembly file `MagicSquareComp.dll` located in the folder where you generated or installed the assembly.
5. Add a reference to the `MWArray` API.

| | |
|---|---|
| If MATLAB is installed on your system | `*matlabroot*\toolbox\dotnetbuilder\bin\<*arch*>\<*framework_version*>\MWArray.dll` |
| If MATLAB Runtime is installed on your system | `<*MATLAB_RUNTIME_INSTALL_DIR*>\toolbox\dotnetbuilder\bin\<*arch*>\<*framework_version*>\MWArray.dll` |

6. Go to **Build**, then **Configuration Manager**, and change the platform from **Any CPU** to **x64**.
7. After you finish adding your code and references, build the application with Visual Studio.

   The build process generates an executable named `makesquareSample1.exe`.
8. Run the application from Visual Studio, in a command window, or by double-clicking the generated executable.

   The application returns the same output as the sample MATLAB code.

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

**Note** This example shows how to call the .NET assembly from a sample C# application. To call the assembly from a Visual Basic® application, use the Microsoft Visual Studio project file `MagicSquareVBApp.vbproj` and Visual Basic file `MagicSquareApp.vb` located in

`*matlabroot*\toolbox\dotnetbuilder\Examples\VS*Version*\NET\MagicSquareExample\MagicSquareVBApp\`

## See Also

`libraryCompiler` | `compiler.build.dotNETAssembly` | `mcc` | `deploytool`

## Related Examples

- “Build .NET Core Application That Runs on Linux and macOS”
- “Integrate Simple MATLAB Function Into .NET Application”

## Package .NET Assemblies from Command Line

| **In this section...** |
|---|
| "Execute Compiler Projects with deploytool" on page 4-10 |
| "Create .NET Assemblies with mcc" on page 4-10 |
| "Differences Between Compiler Apps and Command Line" on page 4-11 |

You can package .NET assemblies 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 .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”

## Distribute .NET Assemblies to Application Developers

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

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

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

# Package a Java Application

## Configure Your Java Environment

| In this section... |
|---|
| “Install the Required JDK” on page 5-2 |
| “Set JAVA_HOME” on page 5-2 |
| “Set the CLASSPATH” on page 5-3 |
| “Configure the Native Library Path Variables” on page 5-3 |

Before you can package MATLAB functions into Java applications or use the generated Java application in a Java development environment, you must ensure that your Java environment is properly configured. You should verify that:

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

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

### Install the Required JDK

To install the proper version of the JDK:

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

   ```
   version -java
   ```
2. Download a Java Developer's Kit (JDK) with the same major version from https://adoptopenjdk.net/.
3. Install the JDK.

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

### Set JAVA_HOME

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

## Set the CLASSPATH

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

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

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

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

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

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

## Configure the Native Library Path Variables

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

| | |
|---|---|
| Windows | `PATH` |
| Linux | `LD_LIBRARY_PATH` |
| Macintosh | `DYLD_LIBRARY_PATH` |

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

## Generate Java Package and Build Java Application

**Supported platforms:** Windows, Linux, Mac

This example shows how to create a Java package from a MATLAB function and generate sample Java code.

### Prerequisites

- Verify that you have a version of Java installed that is compatible with MATLAB Compiler SDK. For information on supported Java versions, see MATLAB Interfaces to Other Languages.

  For information on configuring your development environment after installation, see “Configure Your Java Environment for Generating Packages”.
- End users must have an installation of MATLAB Runtime to run the application. For details, see “Install and Configure MATLAB Runtime”.

  For testing purposes, you can use an installation of MATLAB instead of MATLAB Runtime.

### Create Function in MATLAB

In MATLAB, examine the MATLAB code that you want to package. For this example, open `makesqr.m` located in *`matlabroot`*`\toolbox\javabuilder\Examples\MagicSquareExample\MagicDemoComp`.

```
function y = makesqr(x)
y = magic(x);
```

At the MATLAB command prompt, enter `makesqr(5)`.

The output is a 5-by-5 matrix.

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

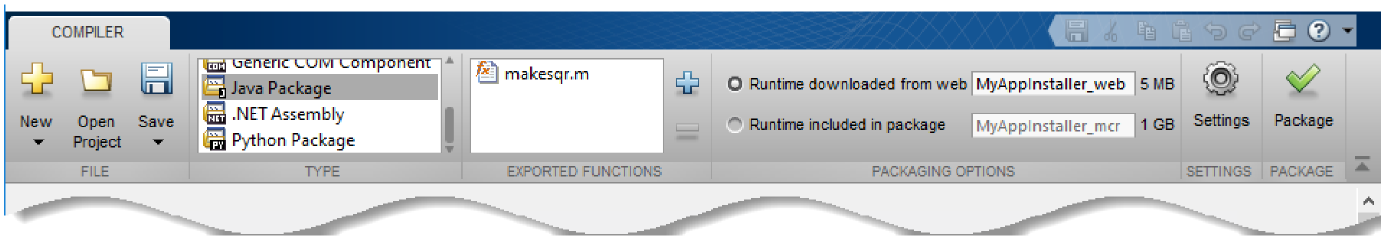
### Create Java Package Using Library Compiler App

Compile the function into a Java package using the **Library Compiler** app. Alternatively, if you want to create a Java package from the MATLAB command window using a programmatic approach, see “Create Java Package Using compiler.build.javaPackage”.

1. On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Library Compiler**.

   Alternatively, you can open the **Library Compiler** app from the MATLAB command prompt by entering:

   ```
   libraryCompiler
   ```



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

In the **Library Compiler** app project window, specify the files of the MATLAB application that you want to deploy.

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

**b** In the **Add Files** window, browse to the example folder, and select the function you want to package. Click **Open**.

The function is added to the list of exported function files. Repeat this step to package multiple files in the same application.

For this example, select the file `makesqr.m`.

**3** In the **Packaging Options** section of the toolstrip, decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the options:

- **Runtime downloaded from web** — Generate an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application. You can specify the file name of the installer.
- **Runtime included in package** — Generate an application that includes the MATLAB Runtime installer. You can specify the file name of the installer.

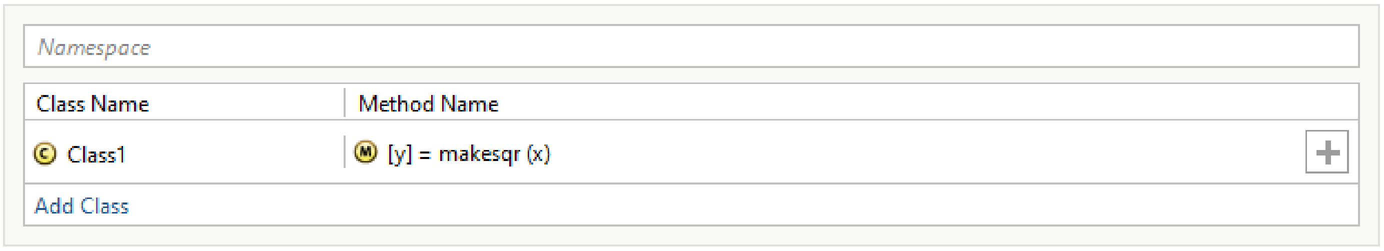
**Note** The first time you select this option, you are prompted to download the MATLAB Runtime installer.

## Specify Package Settings

Next, define the name of your Java package and verify the class mapping for the `.m` file that you are building into your application.

**1** Choose a name for your package. The **Library Name** field is automatically populated with `makesqr` as the name of the package. The same name is followed through in the package implementation steps below.

**2** Verify that the function defined in `makesqr.m` is mapped into `Class1`.



| Class Name | Method Name |
|---|---|
| Class1 | [y] = makesqr (x) |

### Create Sample Driver File

You can use any MATLAB file in the project to generate sample Java driver files. Although Java driver files are not necessary to create a package, you can use them to implement a Java application, as shown in “Compile and Run MATLAB Generated Java Application”.

In the **Samples** section, select **Create New Sample**, and click `makesqr.m`. A MATLAB file opens for you to edit.

```
% Sample script to demonstrate execution of function y = makesqr(x)
x = 0; % Initialize x here
y = makesqr(x);
```

Change `x = 0` to `x = 5`, save the file, and return to the **Library Compiler** app. The compiler converts this MATLAB code to Java code during packaging.

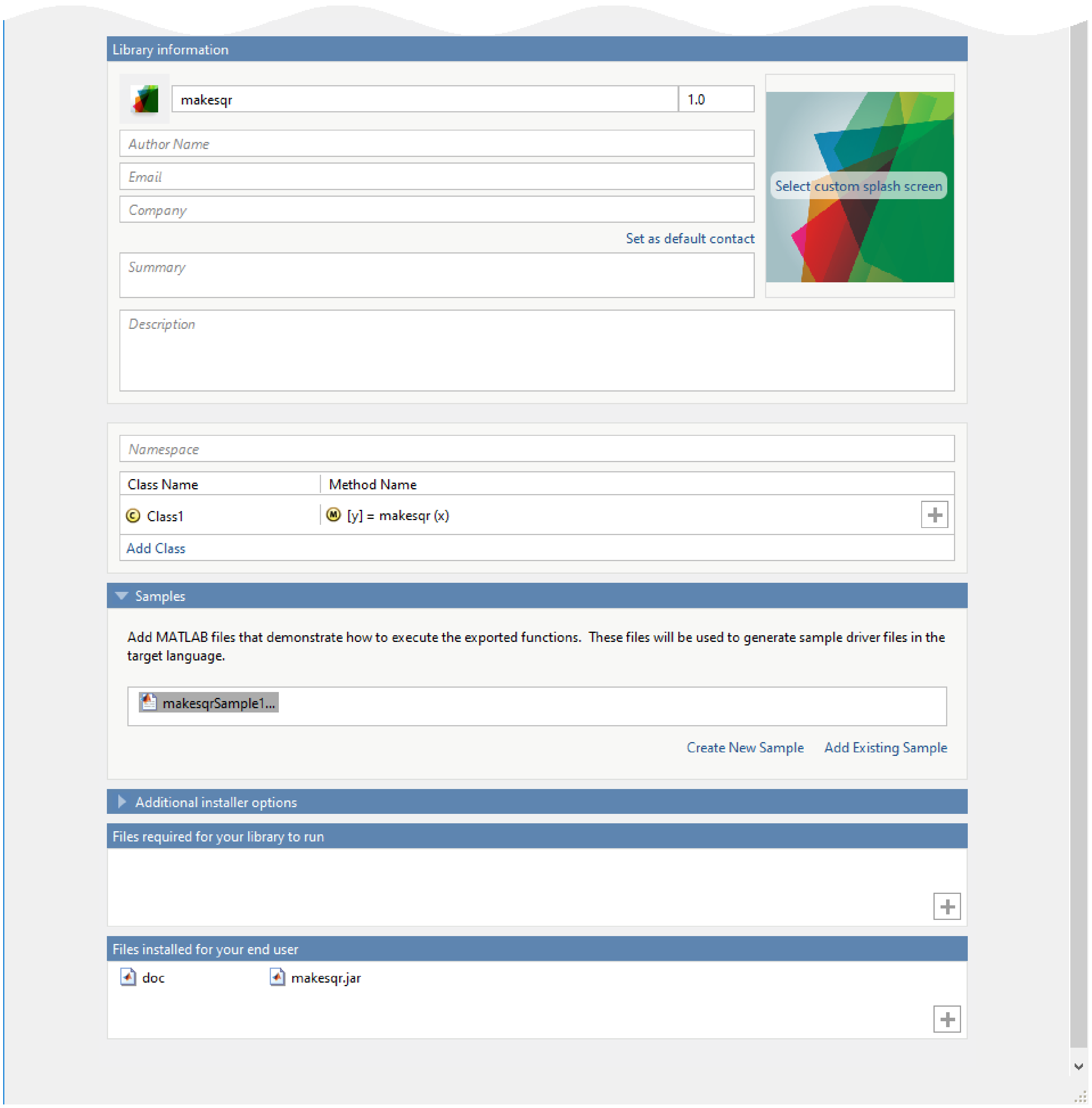
For more information and limitations, see “Sample Driver File Creation”.

### Customize the Application and Its Appearance

In the **Library Compiler** app, you can customize the installer, customize your application, and add more information about the application.

- **Library information** — Information about the deployed application. You can also customize the appearance of the application by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata. See “Customize the Installer”.
- **Additional installer options** — Default installation path for the generated installer and custom logo selection. See “Change the Installation Path”.
- **Files required for your library to run** — Additional files required by the generated application to run. These files are included in the generated application installer. See “Manage Required Files in Compiler Project”.
- **Files installed for your end user** — Files that are installed with your application.

  See “Specify Files to Install with Application”.



Add MATLAB files that demonstrate how to execute the exported functions. These files will be used to generate sample driver files in the target language.

makesqrSample1...
Create New Sample
Add Existing Sample
Additional installer options
Files required for your library to run
Files installed for your end user
doc
makesqr.jar

### Package the Application

When you are finished selecting your packaging options, save your **Library Compiler** project and generate the packaged application.

**1** Click **Package**.

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

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

   When the packaging process is complete, examine the generated output in the target folder.

   - Three folders are generated: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`.

     For more information about the files generated in these folders, see “Files Generated After Packaging MATLAB Functions”.
   - The log file `PackagingLog.html` contains packaging results.

## Create Java Package Using compiler.build.javaPackage

As an alternative to the **Library Compiler** app, you can create a Java package using a programmatic approach. If you have already created a package using the **Library Compiler**, see “Compile and Run MATLAB Generated Java Application”.

1. Save the path to the `makesqr.m` file located in *`matlabroot`*`\toolbox\javabuilder\Examples\MagicSquareExample\MagicDemoComp`.

```
appFile = fullfile(matlabroot,'toolbox','javabuilder','Examples', ...
    'MagicSquareExample','MagicDemoComp','makesqr.m');
```
2. Save the following code in a sample file named `makesqrSample1.m`:

```
x = 5;
y = makesqr(x);
```
3. Build the Java package using the `compiler.build.javaPackage` function. Use name-value arguments to add a sample file and enable verbose output.

```
buildResults = compiler.build.javaPackage(appFile, ...
'SampleGenerationFiles','makesqrSample1.m', ...
'Verbose','on');
```

   You can specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.build.javaPackage`.

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

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

   - `classes` — Folder that contains the Java class files and the deployable archive CTF file.
   - `doc` — Folder that contains HTML documentation for all classes in the package.
   - `example` — Folder that contains Java source code files.
   - `samples` — Folder that contains the Java sample driver file `makesqrSample1.java`.
   - `GettingStarted.html` — File that contains information on integrating your package.
   - `includedSupportPackages.txt` — Text file that lists all support files included in the package.
   - `makesqr.jar` — Java archive file.
   - `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see Functions Not Supported For Compilation on page 12-7.

- `readme.txt` — Text file that contains information on deployment prerequisites and the list of files to package for deployment.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

**Note** The generated package does not include MATLAB Runtime or an installer. To create an installer using the `buildResults` object, see `compiler.package.installer`.

## Compile and Run MATLAB Generated Java Application

After creating your Java package, you can call it from a Java application. This example uses the sample Java code generated during packaging. You can use this sample Java application code as a guide to write your own application.

1. Copy and paste the generated Java file `makesqrSample1.java` from the `samples` folder into the folder that contains the `makesqr.jar` package. If you used the Library Compiler, `makesqr.jar` is located in the `for_testing` folder.
2. At the system command prompt, navigate to the folder that contains `makesqrSample1.java` and `makesqr.jar`.
3. Compile the application using `javac`. In the classpath argument, you specify the paths to `javabuilder.jar`, which contains the `com.mathworks.toolbox.javabuilder` package, and your generated Java package `makesqr.jar`.

   - On Windows, type:

     ```
     javac -classpath "matlabroot\toolbox\javabuilder\jar\javabuilder.jar";.\makesqr.jar makesqrSample1.java
     ```
   - On UNIX, type:

     ```
     javac -classpath "matlabroot/toolbox/javabuilder/jar/javabuilder.jar":./makesqr.jar makesqrSample1.java
     ```

     Replace *`matlabroot`* with the path to your MATLAB or MATLAB Runtime installation folder. For example, on Windows, the path may be `C:\Program Files\MATLAB\R2022a`.

   **Note** If `makesqr.jar` or `makesqrSample1.java` is not in the current directory, specify the full or relative path in the command. If the path contains spaces, surround it with double quotes.
4. Run the application using `java`.

   - On Windows, type:

     ```
     java -classpath .;"matlabroot\toolbox\javabuilder\jar\javabuilder.jar";.\makesqr.jar makesqrSample1
     ```
   - On UNIX, type:

     ```
     java -classpath .:"matlabroot/toolbox/javabuilder/jar/javabuilder.jar":./makesqr.jar makesqrSample1
     ```

   **Note** The dot (`.`) in the first position of the class path represents the current working directory. If it is not there, you get a message stating that Java cannot load the class.

   The application returns the same output as the sample MATLAB code.

   ```
     17    24     1     8    15
     23     5     7    14    16
   ```

```
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

## See Also

`libraryCompiler` | `compiler.build.javaPackage` | `mcc` | `deploytool`

## Related Examples

- "Integrate Simple MATLAB Function Into Java Application"
- "Display MATLAB Plot in 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

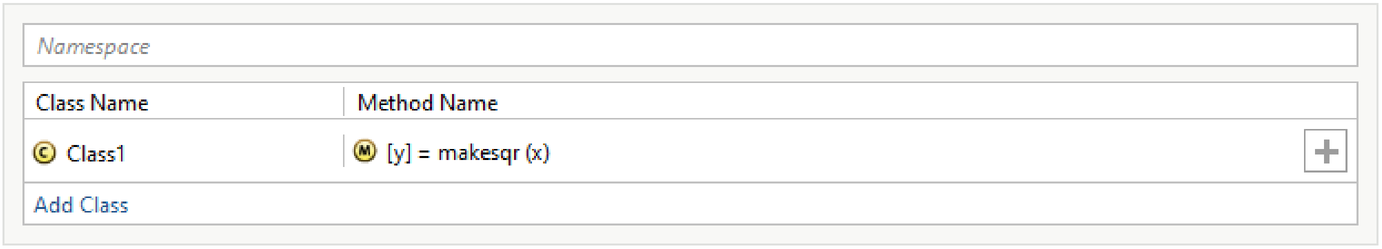
- "Generate Java Package and Build Java Application"

# Map Functions to Java Class Methods

| In this section... |
|---|
| “Map Functions to Java Classes with the Library Compiler App” on page 5-13 |
| “Map Functions to Java Classes with mcc” on page 5-14 |

## Map Functions to Java Classes with the Library Compiler App

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



The **Namespace** field at the top of the class browser specifies the name of the 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 5-13.
3. Add one or more methods to the class as described in “Add a Method to a Java Class” on page 5-14.

### Rename a Java Class

To rename a Java class:

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

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

### Delete a Class from a JavaApplication

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 a Method to a Java Class

To add a method to a Java class:

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

### Delete a Method from a Java Class

To delete a method from a Java class:

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

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

## Map Functions to Java Classes with mcc

When using `mcc` to generate Java 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`.

## Distribute Java Applications to Application Developers

Distribute the following to the application developer integrating the application:

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

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

# Package a Python Application

## Generate a Python Package and Build a Python Application

**Supported platforms:** Windows, Linux, Mac

This example shows how to create a Python package from a MATLAB function and integrate the generated package into a Python application.

### Prerequisites

- Verify that you have a version of Python installed that is compatible with MATLAB Compiler SDK. For details, see MATLAB Supported Interfaces to Other Languages.
- End users must have an installation of MATLAB Runtime to run the application. For testing purposes, you can use an installation of MATLAB instead of MATLAB Runtime. For details, see “Install and Configure MATLAB Runtime”.

### Create Function in MATLAB

In MATLAB, examine the MATLAB code that you want packaged. For this example, create a function named `makesqr.m` that contains the following code:

```
function y = makesqr(x)
y = magic(x);
```

At the MATLAB command prompt, enter `makesqr(5)`.

The output is a 5-by-5 matrix.

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

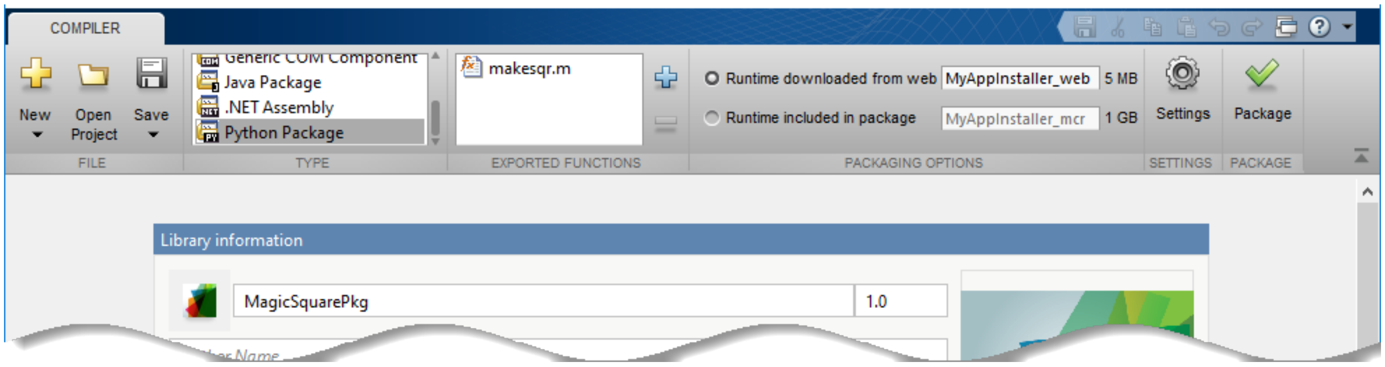
### Create Python Application Using Library Compiler App

Compile the function into a Python package using the **Library Compiler** app. Alternatively, if you want to create a Python package from the MATLAB command window using a programmatic approach, see “Create Python Package Using compiler.build.pythonPackage”.

1. On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Library Compiler**.

   Alternatively, you can open the **Library Compiler** app from the MATLAB command prompt.

   ```
   libraryCompiler
   ```



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

In the **Library Compiler** app project window, specify the files of the MATLAB application that you want to deploy.

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

**b** In the **Add Files** window, browse to the example folder, and select the function you want to package. Click **Open**.

The function is added to the list of exported function files. Repeat this step to package multiple files in the same application.

For this example, select the `makesqr.m` file that you wrote earlier.

**3** In the **Packaging Options** section of the toolstrip, decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the options:

- **Runtime downloaded from web** — Generate an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application. You can specify the file name of the installer.
- **Runtime included in package** — Generate an application that includes the MATLAB Runtime installer. You can specify the file name of the installer.

**Note** The first time you select this option, you are prompted to download the MATLAB Runtime installer.

### Specify Package Settings

Next, define the name of your Python package.

- Choose a name for your package. The **Library Name** field is automatically populated with `makesqr` as the name of the package. Rename it as `MagicSquarePkg`. For more information on naming requirements for the Python package, see “Import Compiled Python Packages”.

### Create Sample Driver File

You can add MATLAB files to the project to generate sample Python driver files. Although Python driver files are not necessary to create a package, you can use them to implement a Python application, as shown in “Install and Run MATLAB Generated Python Application”.

In the **Samples** section, select **Create New Sample**, and click `makesqr.m`. A MATLAB file opens for you to edit.

```
% Sample script to demonstrate execution of function y = makesqr(x)
x = 0; % Initialize x here
y = makesqr(x);
```

Change `x = 0` to `x = 5`, save the file, and return to the **Library Compiler** app.

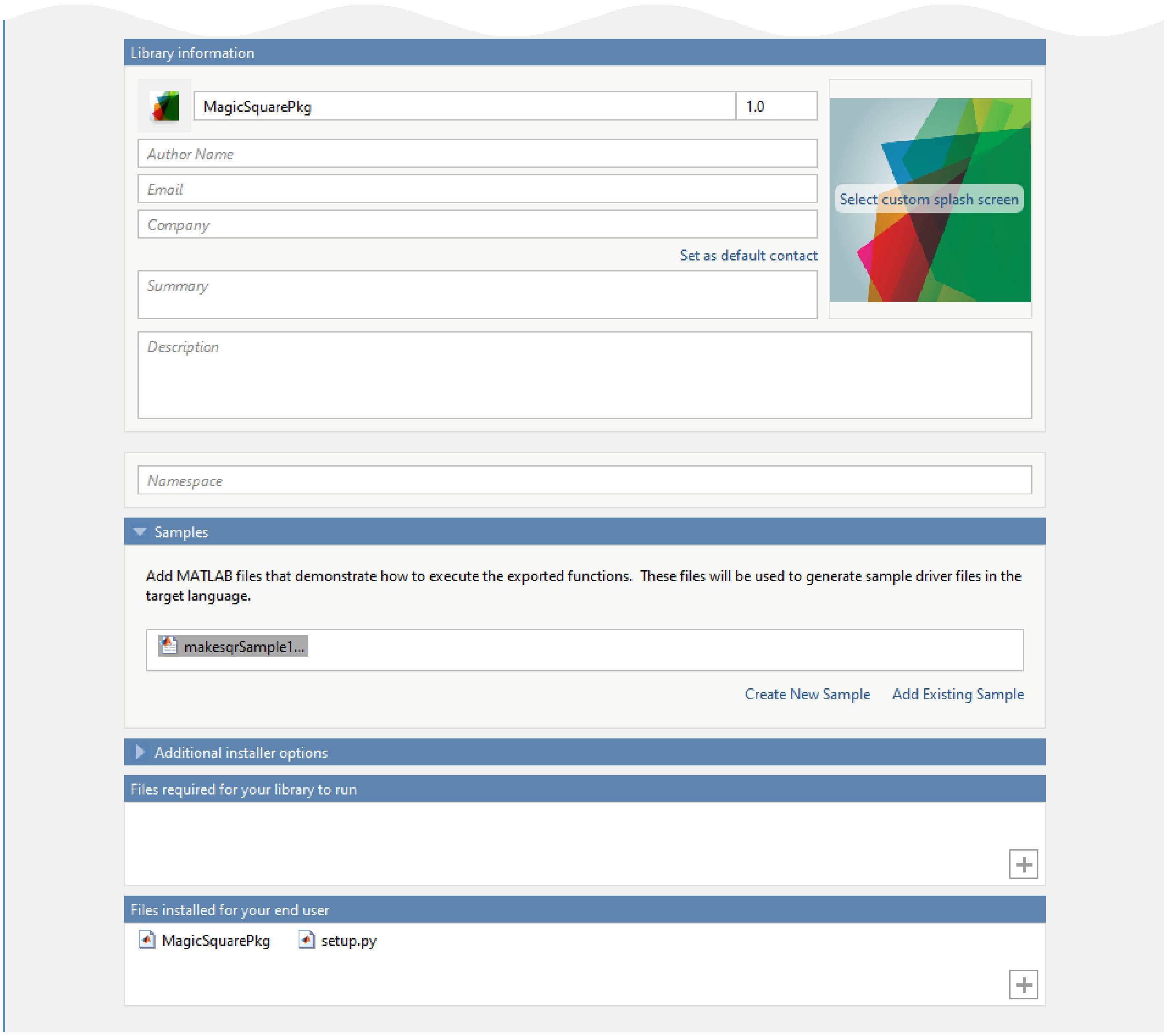
For more information and limitations, see “Sample Driver File Creation”.

### Customize the Application and Its Appearance

In the **Library Compiler** app, you can customize the installer, customize your application, and add more information about the application.

- **Library information** — Information about the deployed application. You can also customize the appearance of the application by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata. See “Customize the Installer”.
- **Additional installer options** — Default installation path for the generated installer and custom logo selection. See “Change the Installation Path”.
- **Files required for your library to run** — Additional files required by the generated application to run. These files are included in the generated application installer. See “Manage Required Files in Compiler Project”.
- **Files installed for your end user** — Files that are installed with your application.

  See “Specify Files to Install with Application”.


Library information

MagicSquarePkg 1.0

Author Name

Email

Company

Set as default contact

Summary

Description

Select custom splash screen

Namespace

Samples

Add MATLAB files that demonstrate how to execute the exported functions. These files will be used to generate sample driver files in the target language.

makesqrSample1...

Create New Sample Add Existing Sample

Additional installer options

Files required for your library to run

Files installed for your end user

MagicSquarePkg setup.py

## Package the Application

When you are finished selecting your packaging options, save your **Library Compiler** project and generate the packaged application.

**1** Click **Package**.

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

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

When the packaging process is complete, examine the generated output in the target folder.

- Three folders are generated: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`.

  For more information about the files generated in these folders, see "Files Generated After Packaging MATLAB Functions".
- The log file `PackagingLog.html` contains packaging results.

## Create Python Package Using compiler.build.pythonPackage

As an alternative to the **Library Compiler** app, you can create a Python package using a programmatic approach. If you have already created a package using the **Library Compiler**, see "Install and Run MATLAB Generated Python Application".

1 Save the following code in a sample file named `makesqrSample1.m`:

```
x = 5;
y = makesqr(x);
```
2 Build the Python package using the `compiler.build.pythonPackage` function and the `makesqr.m` file that you wrote earlier. Use name-value arguments to specify the package name and add a sample file.

```
buildResults = compiler.build.pythonPackage('makesqr.m', ...
'PackageName','MagicSquarePkg', ...
'SampleGenerationFiles','makesqrSample1.m', ...
'Verbose','on');
```

   You can specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.build.pythonPackage`.

   The `compiler.build.Results` object `buildResults` contains information on the build type, generated files, included support packages, and build options.
3 The function generates the following files within a folder named `MagicSquarePkgpythonPackage` in your current working directory:

   - `samples\makesqrSample1.py` — Python sample application file.
   - `GettingStarted.html` — HTML file that contains information on integrating your package.
   - `includedSupportPackages.txt` — Text file that lists all support files included in the package.
   - `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations.
   - `readme.txt` — Text file that contains packaging and interface information.
   - `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
   - `setup.py` — Python file that installs the package.
   - `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

**Note** The generated package does not include MATLAB Runtime or an installer. To create an installer using the `buildResults` object, see `compiler.package.installer`.

## Install and Run MATLAB Generated Python Application

After creating your Python package, you can call it from a Python application. This example uses the sample Python code generated during packaging. You can use this sample Python application code as a guide to write your own application.

**1** Copy and paste the generated Python file `makesqrSample1.py` from the `samples` folder into the folder that contains the `setup.py` file.

The program listing for `makesqrSample1.py` is shown below.

```
#!/usr/bin/env python
"""
Sample script that uses the MagicSquarePkg module created using
MATLAB Compiler SDK.

Refer to the MATLAB Compiler SDK documentation for more information.
"""

from __future__ import print_function
import MagicSquarePkg
import matlab

my_MagicSquarePkg = MagicSquarePkg.initialize()

xIn = matlab.double([5.0], size=(1, 1))
yOut = my_MagicSquarePkg.makesqr(xIn)
print(yOut, sep='\n')

my_MagicSquarePkg.terminate()
```

**2** At the system command prompt, navigate to the folder that contains `makesqrSample1.py` and `setup.py`.

**3** Install the application using the `python` command.

```
python setup.py install
```

To install to a location other than the default, consult "Installing Python Modules" in the official Python documentation.

**4** Run the application at the system command prompt.

```
python makesqrSample1.py
```

If you used sample MATLAB code in the packaging steps, this application returns the same output as the sample code.

```
[[17.0,24.0,1.0,8.0,15.0],[23.0,5.0,7.0,14.0,16.0],[4.0,6.0,13.0,20.0,22.0],
[10.0,12.0,19.0,21.0,3.0],[11.0,18.0,25.0,2.0,9.0]]
```

**Note** On macOS, you must use the `mwpython` script instead of `python`. For example, `mwpython makesqrSample1.py`.

The `mwpython` script is located in the *`matlabroot`*`/bin` folder, where *`matlabroot`* is the location of your MATLAB or MATLAB Runtime installation.

## See Also

`mwpython` | `libraryCompiler` | `compiler.build.pythonPackage` | `mcc` | `deploytool`

## Package Python Applications from Command Line

**In this section...**

**Note** MATLAB Compiler SDK cannot package MATLAB code that uses the MATLAB Python interface.

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"

## Distribute Python Applications to Application Developers

Distribute the following to the application developer integrating the application:

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

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

# 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;
```

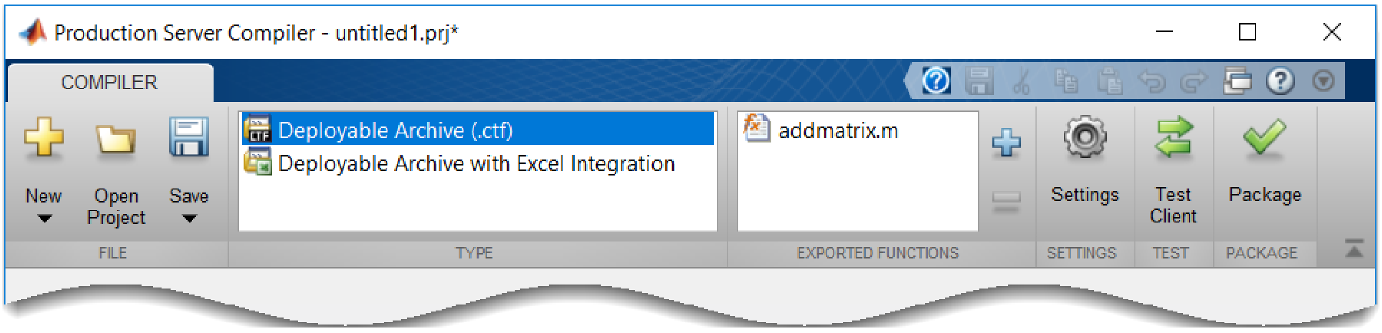
At the MATLAB command prompt, enter `addmatrix([1 4 7; 2 5 8; 3 6 9], [1 4 7; 2 5 8; 3 6 9])`.

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)**.


Production Server Compiler - untitled1.prj*
COMPILER
New
Open Project
Save
Deployable Archive (.ctf)
Deployable Archive with Excel Integration
addmatrix.m
Settings
Test Client
Package
FILE
TYPE
EXPORTED FUNCTIONS
SETTINGS
TEST
PACKAGE

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.

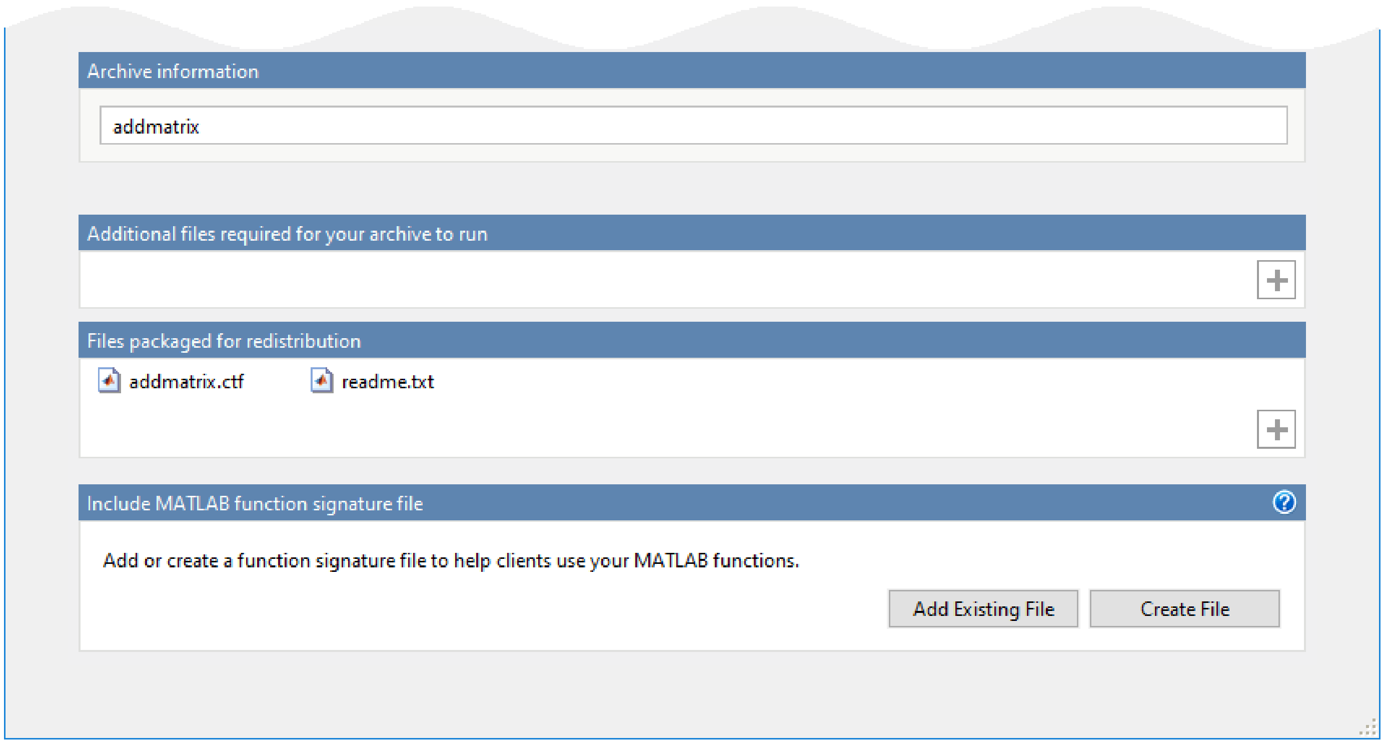
## Customize the Application and Its Appearance

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`

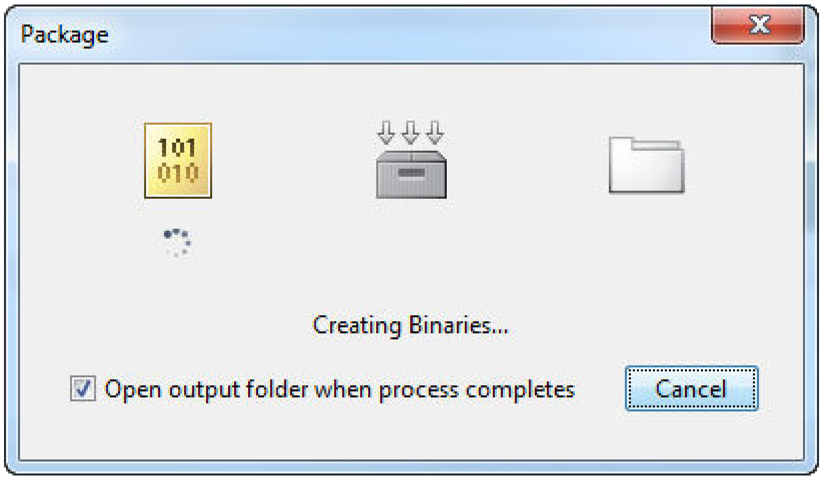
  See “Specify Files to Install with Application”
- **Include MATLAB function signature file** — Add or create a function signature file to help clients use your MATLAB functions.


Archive information
addmatrix
Additional files required for your archive to run
Files packaged for redistribution
addmatrix.ctf
readme.txt
Include MATLAB function signature file
Add or create a function signature file to help clients use your MATLAB functions.
Add Existing File
Create File

## 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

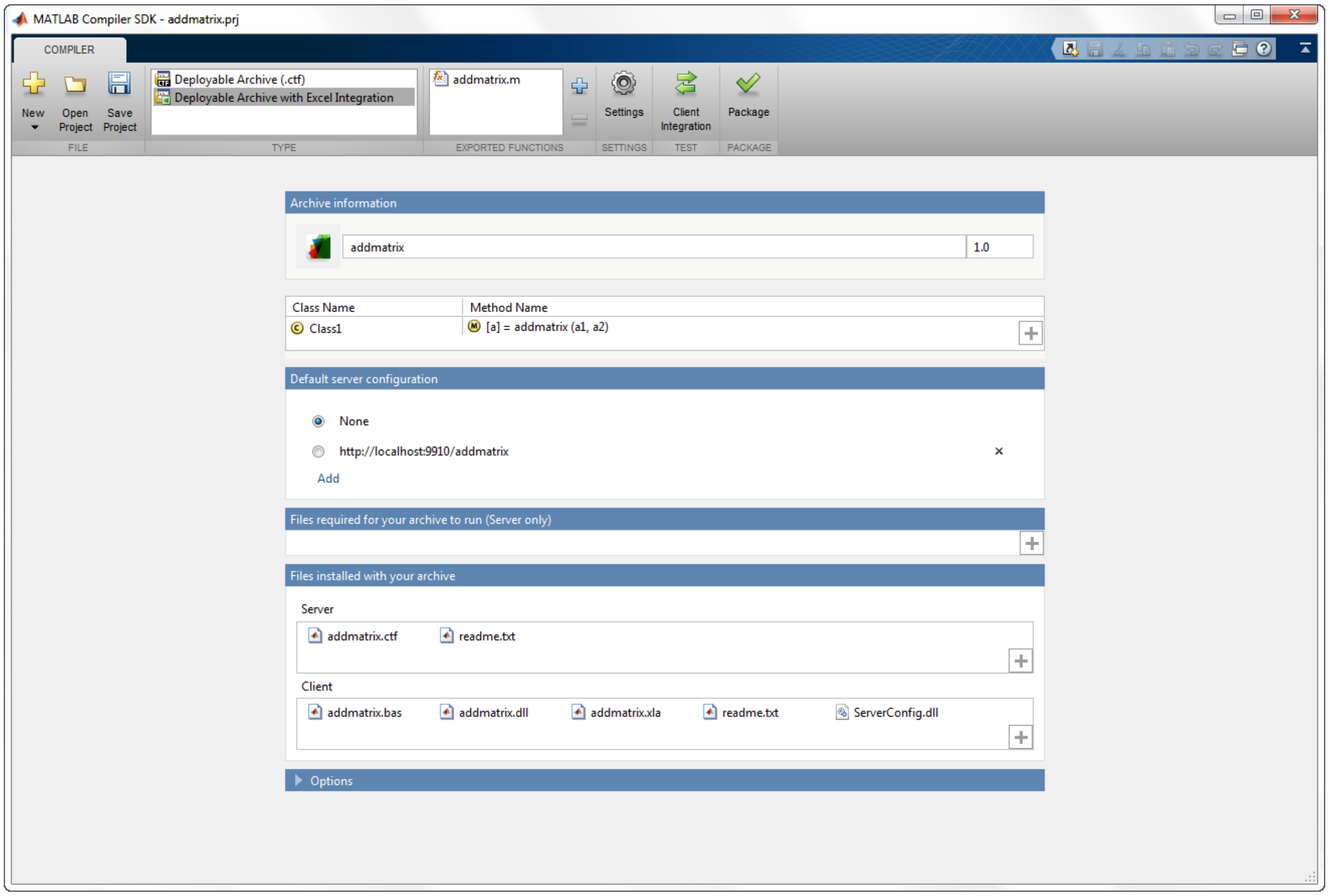
## Build Excel Add-In and Deployable Archive

**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.



Deployable Archive (.ctf)
Deployable Archive with Excel Integration
addmatrix.m

Archive information
addmatrix 1.0

| Class Name | Method Name |
|---|---|
| Class1 | [a] = addmatrix (a1, a2) |

Default server configuration
[✓ None]
http://localhost:9910/addmatrix

Files required for your archive to run (Server only)

Files installed with your archive
Server
addmatrix.ctf readme.txt
Client
addmatrix.bas addmatrix.dll addmatrix.xla readme.txt ServerConfig.dll

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.

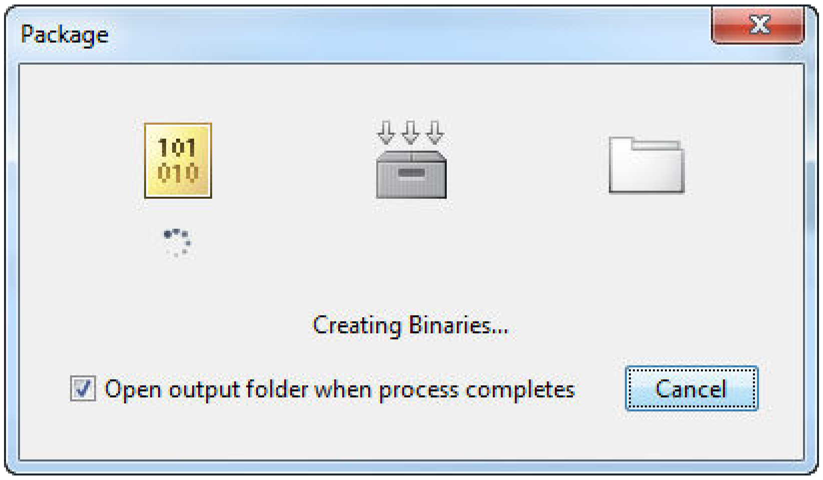
   The server configuration defines the connection to the MATLAB Production Server instance running the MATLAB code.

   - **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

# Create a Generic COM Component with MATLAB Code

**Supported platform:** Windows

This example shows how to create a generic COM component using a MATLAB function and integrate it into an application. The target system does not require a licensed copy of MATLAB.

## Prerequisites

- Verify that you have the Windows 10 SDK kit installed. For details, see Windows 10 SDK.
- Verify that you have MinGW-w64 installed. To install it from the MathWorks File Exchange, see MATLAB Support for MinGW-w64 C/C++ Compiler.

  To ensure that MATLAB detects the Windows 10 SDK kit and MinGW-w64, use the following command:

  ```
  mbuild -setup -client mbuild_com
  ```
- Verify that you have Microsoft Visual Studio installed.
- End users must have an installation of MATLAB Runtime to run the application. For details, see “Install and Configure MATLAB Runtime”.

  For testing purposes, you can use an installation of MATLAB instead of MATLAB Runtime.

## Create Function in MATLAB

In MATLAB, examine the MATLAB code that you want packaged. For this example, open `makesquare.m` located in `*matlabroot*\toolbox\dotnetbuilder\Examples\VS*Version*\COM\MagicSquareExample\MagicSquareComp`.

```
function y = makesquare(x)
y = magic(x);
```

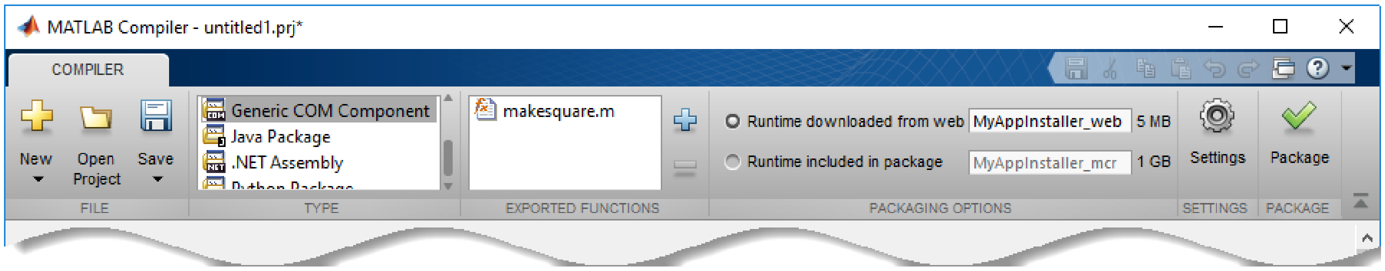
At the MATLAB command prompt, enter `makesquare(5)`.

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

## Create Generic COM Component Using Library Compiler App

Package the function into a COM component using the **Library Compiler** app. Alternatively, if you want to create a COM component from the MATLAB command window using a programmatic approach, see “Create COM Component Using compiler.build.COMComponent”.

1. On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Library Compiler**. In the **MATLAB Compiler** project window, click **Generic COM Component**.



Alternately, you can open the **Library Compiler** app by entering `libraryCompiler` at the MATLAB prompt.

**2** In the **Library Compiler** app project window, specify the files of the MATLAB application that you want to deploy.

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

   **b** In the **Add Files** window, browse to the example folder, and select the function you want to package. Click **Open**.

   The function is added to the list of exported function files. Repeat this step to package multiple files in the same application.

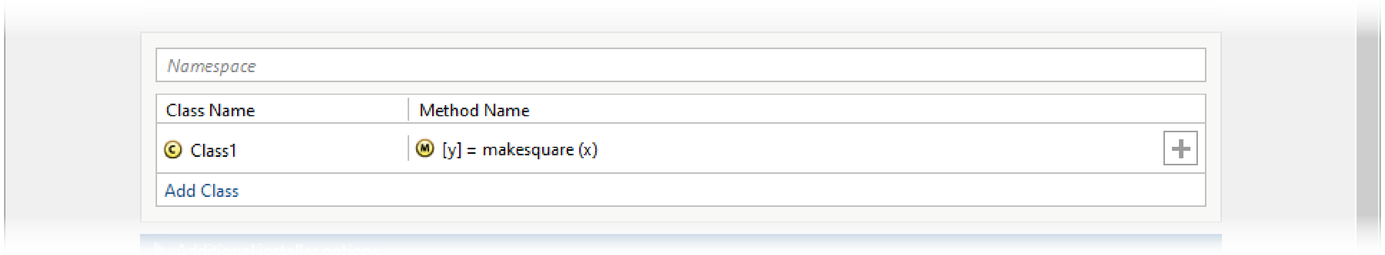
**3** In the **Packaging Options** section of the toolstrip, decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the options:

   - **Runtime downloaded from web** — Generate an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application. You can specify the file name of the installer.
   - **Runtime included in package** — Generate an application that includes the MATLAB Runtime installer. You can specify the file name of the installer.

   **Note** The first time you select this option, you are prompted to download the MATLAB Runtime installer.

**4** In the **Library Name** field, replace `makesquare` with `MagicSquareComp`.

**5** Verify that the function defined in `makesquare.m` is mapped into `Class1`.



| Class Name | Method Name |
|---|---|
| Class1 | [y] = makesquare (x) |

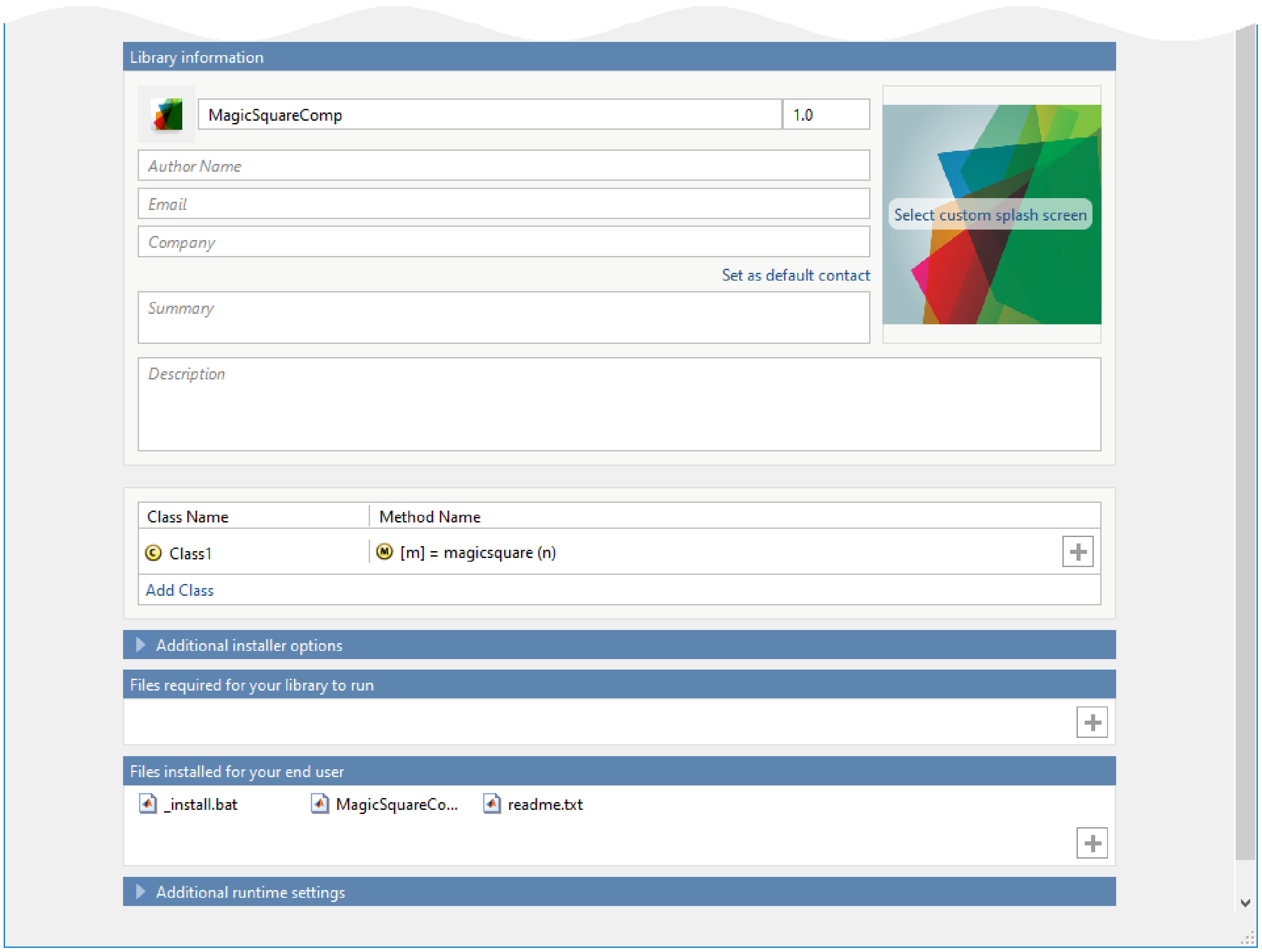
## Customize the Application and Its Appearance

In the **Library Compiler** app, you can customize the installer, customize your application, and add more information about the application.

- **Library information** — Information about the deployed application. You can also customize the appearance of the application by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata. See “Customize the Installer”.

- **Additional installer options** — Default installation path for the generated installer and custom logo selection. See “Change the Installation Path”.
- **Files required for your library to run** — Additional files required by the generated application to run. These files are included in the generated application installer. See “Manage Required Files in Compiler Project”.
- **Files installed for your end user** — Files that are installed with your application.

  See “Specify Files to Install with Application”.
- **Additional runtime settings** — Platform-specific options for controlling the generated executable. See “Additional Runtime Settings”.


Library information
MagicSquareComp
1.0
Author Name
Email
Company
Set as default contact
Select custom splash screen
Summary
Description
Add Class
Additional installer options
Files required for your library to run
Files installed for your end user
Additional runtime settings

| Class Name | Method Name |
|---|---|
| Class1 | [m] = magicsquare (n) |

_install.bat MagicSquareCo... readme.txt

## Package the Application

When you are finished selecting your packaging options, save your **Library Compiler** project and generate the packaged application.

1 Click **Package**.

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

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

   When the packaging process is complete, examine the generated output in the target folder.

   - Three folders are generated: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`.

     For more information about the files generated in these folders, see “Files Generated After Packaging MATLAB Functions”.
   - The log file `PackagingLog.html` contains packaging results.

## Create COM Component Using compiler.build.COMComponent

As an alternative to the **Library Compiler** app, you can create a COM component using a programmatic approach. If you have already created a component using the **Library Compiler**, see “Integrate into COM Application”.

1 Save the path to the file `makesquare.m` located in `*matlabroot*\toolbox\dotnetbuilder\Examples\VS*Version*\COM\MagicSquareExample\MagicSquareComp`. For example, if you are using Visual Studio version 15, type:

```
appFile = fullfile(matlabroot,'toolbox','dotnetbuilder','Examples', ...
    'VS15','COM','MagicSquareExample','MagicSquareComp','makesquare.m');
```

2 Build the COM component using the `compiler.build.comComponent` function. Use name-value arguments to specify the component name and class name.

```
buildResults = compiler.build.comComponent(appFile, ...
'ComponentName','MagicSquareComp', ...
'ClassName','Class1');
```

   You can specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.build.comComponent`.

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

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

   - `magicsquare.def`
   - `magicsquare.rc`
   - `magicsquare_1_0.dll`
   - `readme.txt`
   - `requiredMCRProducts.txt`
   - `unresolvedSymbols.txt`

   - `Class1_com.cpp` — C++ source code file that defines the class.
   - `Class1_com.hpp` — C++ header file that defines the class.
   - `dlldata.c` — C source code file that contains entry points and data structures required by the class factory for the DLL.

- `GettingStarted.html` — HTML file that contains steps on installing COM components.
- `includedSupportPackages.txt` — Text file that contains information on included support packages.
- `MagicSquareComp.def` — Module definition file that defines which functions to include in the DLL export table.
- `MagicSquareComp.rc` — Resource script file that describes the resources used by the component.
- `MagicSquareComp_1_0.dll` — Dynamic-link library file.
- `MagicSquareComp_dll.cpp` — C++ source code file that contains helper functions.
- `MagicSquareComp_idl.h` — C++ header file.
- `MagicSquareComp_idl.idl` — Interface definition language file.
- `MagicSquareComp_idl.tlb` — Type library file that contains information about the COM object properties and methods.
- `MagicSquareComp_idl_i.c` — C source code file that contains the IIDs and CLSIDs for the IDL interface.
- `MagicSquareComp_idl_p.c` — C source code file that contains the proxy stub code for the IDL interface.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations.
- `mwcomtypes.h` — C++ header file that contains the definitions for the interfaces.
- `mwcomtypes_i.c` — C source code file that contains the IIDs and CLSIDs.
- `mwcomtypes_p.c` — C source code file that contains the proxy stub code.
- `readme.txt` — Text file that contains deployment information.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

**Note** The generated component does not include MATLAB Runtime or an installer. To create an installer using the `buildResults` object, see `compiler.package.installer`.

## Integrate into COM Application

To integrate your COM component into an application, see “Creating the Microsoft Visual Basic Project”.

## See Also

`libraryCompiler` | `compiler.build.comComponent` | `mcc` | `deploytool`

## More About

- “Call COM Objects in Visual C++ Programs”

# 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`<br><br>The `W` option with `com` as the type controls the generation of wrapper files, which you can use to support components. |
| | **Syntax**<br><br>`mcc -W 'com:<component_name>[,<class_name>[,<major>.<minor>]]'`<br><br>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**<br><br>`mcc -W 'com:mycomponent,myclass,1.0' -T link:lib foo.m bar.m`<br><br>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<br><br>A separate COM named `<class_name>` is created for each class argument that is passed.<br><br>Following the `<class_name>` parameter is a comma-separated list of source files that are encapsulated as methods for the class. |
| | **Syntax**<br><br>`class{<class_name>:[file, [file,...]]}` |
| | **Example**<br><br>`mcc -B 'com:mycomponent,myclass,1.0' foo.m bar.m class{myclass2:foo2.m, bar2.m}`<br><br>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:`<br><br>Uses the bundle. |
| | **Syntax**<br><br>`mcc -B '<bundle>'[:<a1>,<a2>,...,<an>]` |
| | **Example**<br><br>`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`<br><br>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.<br><br>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.<br><br>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.<br><br>**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**<br><br>`mcc -S -B 'com:mycomponent,myclass,1.0' foo.m bar.m`<br><br>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.<br><br>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`<br><br>The `\src` and `\distrib` subfolders are used to package components. |
| | **Syntax**<br><br>`-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"

## Distribute COM Components to Application Developers

Distribute the following to the application developer integrating the component:

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

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

# Customizing a Compiler Project

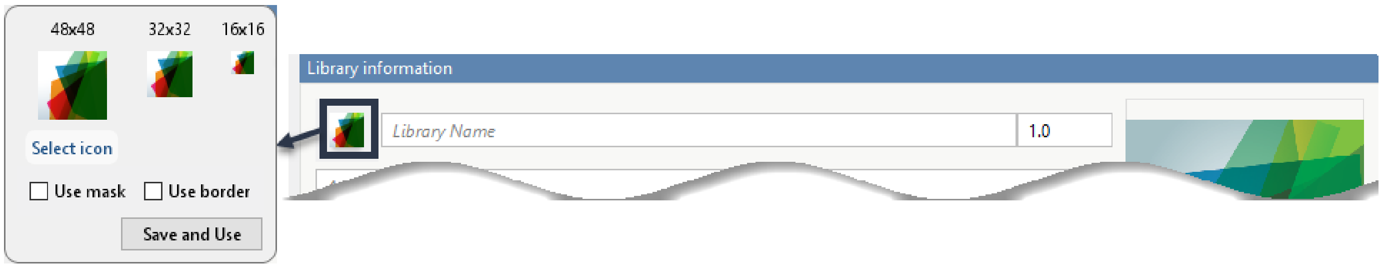
## Customize an Application

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**.

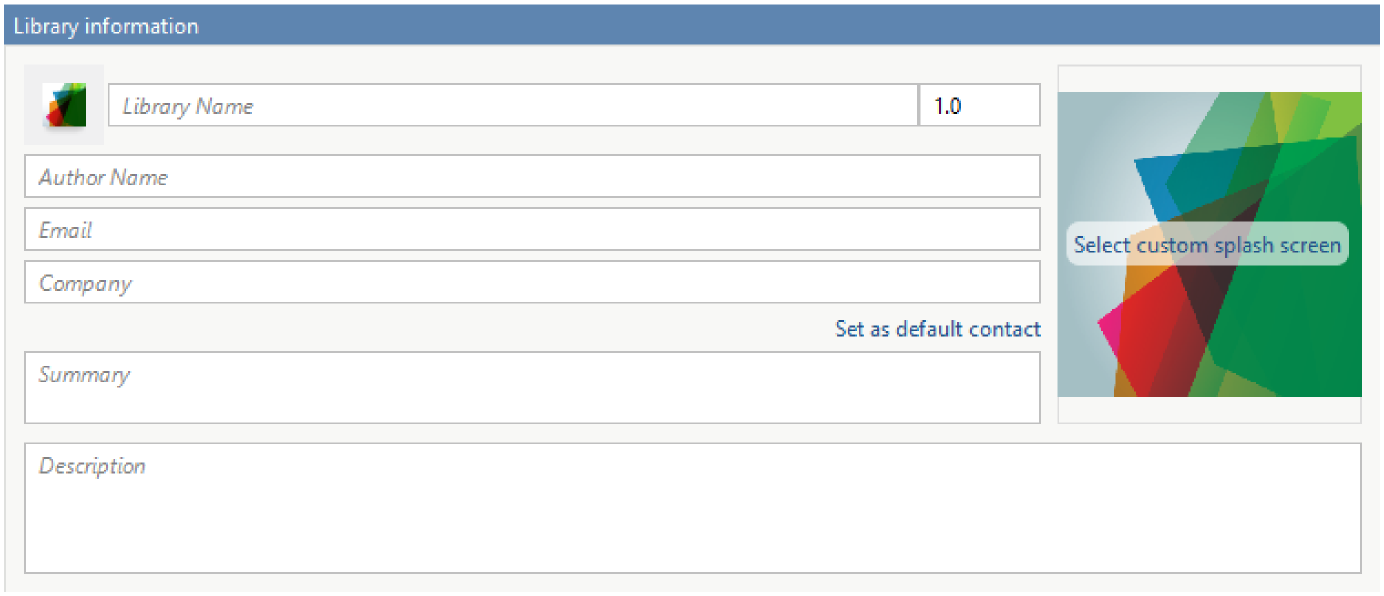
#### Add Library or Application Information

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.

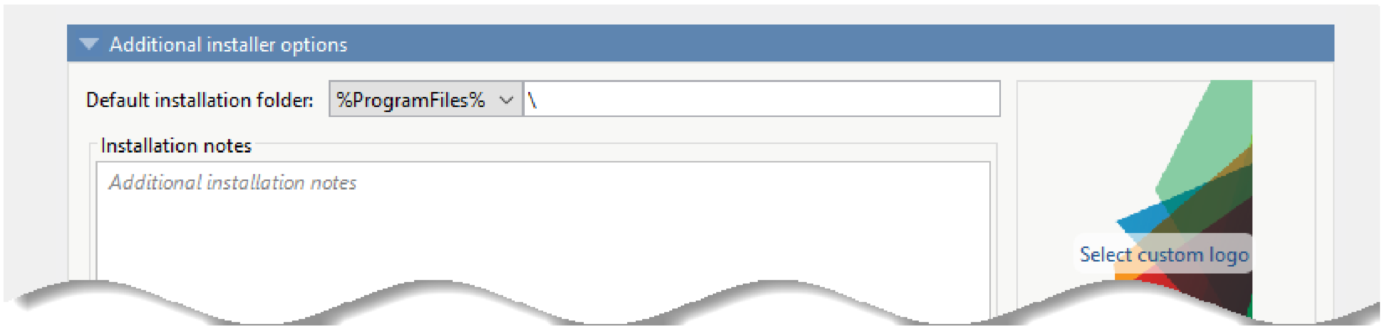
You can drag and drop a custom image onto the default splash screen.

### 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**.


Additional installer options
Default installation folder:
%ProgramFiles%
\
Installation notes
Additional installation notes
Select custom logo

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

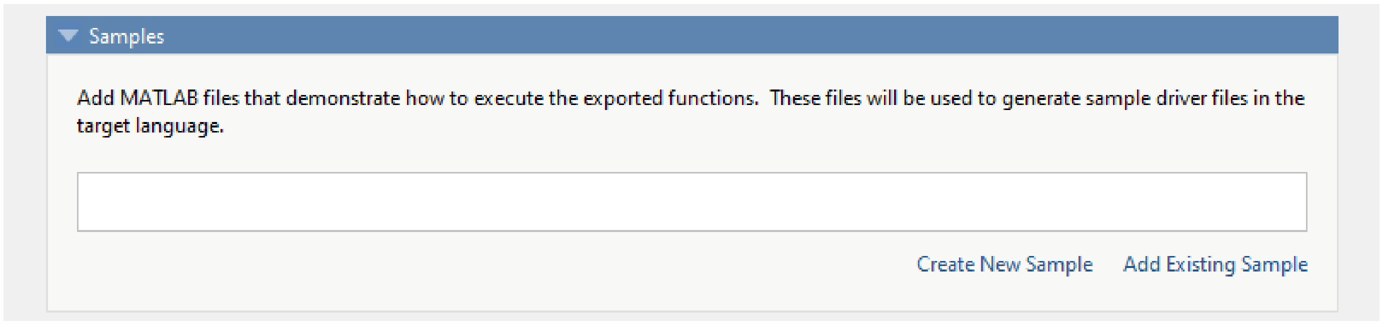
If you are using `mcc` to package your MATLAB code, the compiler does not display a list of required files before running. Instead, it packages all the required files that are discovered by the dependency analysis function and adds them to the generated binary file.

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

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

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



Add MATLAB files that demonstrate how to execute the exported functions. These files will be used to generate sample driver files in the target language.

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

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

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

Additional considerations specific to the target language are as follows:

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

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

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

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

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

## Specify Files to Install with Application

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

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

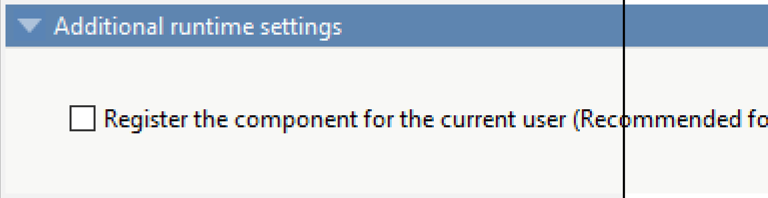
To add files to the list, click ➕, and select the file from the file explorer.

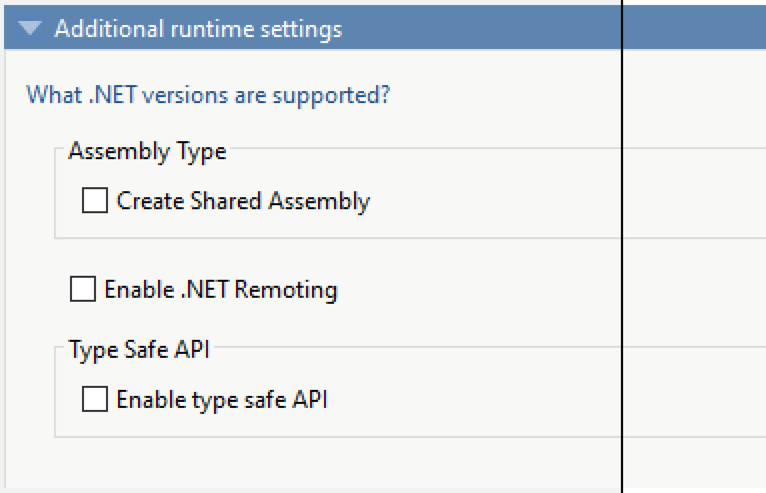
JAR files are added to the application class path as if you had called `javaaddpath`.

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

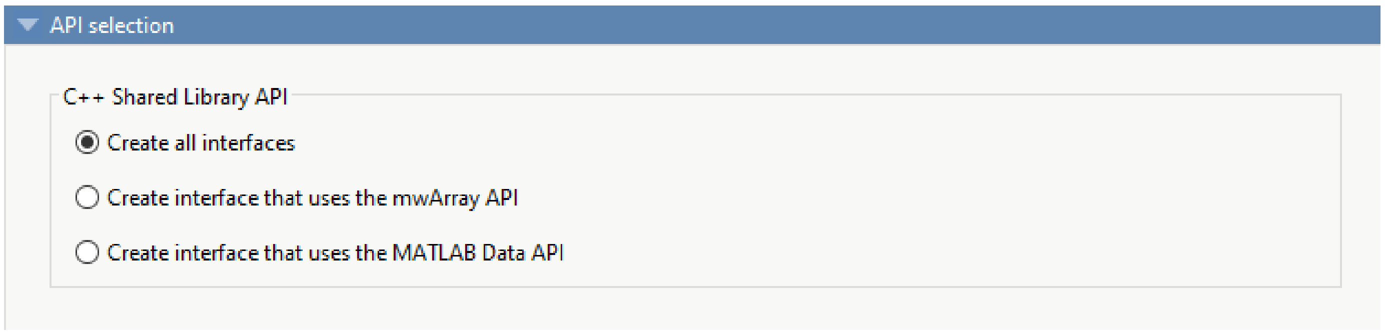
When installed on a target computer, the files listed in the **Files installed for your end user** are saved in the `application` folder.

## Additional Runtime Settings

| Type of Packaged Application | Description | Additional Runtime Settings Options |
|---|---|---|
| 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. |  Additional runtime settings<br>☐ Register the component for the current user (Recommended fo |

| Type of Packaged Application | Description | Additional Runtime Settings Options |
|---|---|---|
| .NET Assembly | • **Create Shared Assembly** — Enables sharing MATLAB Runtime installer instances for multiple .NET assemblies. <br>• **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". <br>• **Enable Type Safe API** — Enables the type safe API for the packaged .NET assembly. |  Additional runtime settings<br>What .NET versions are supported?<br>Assembly Type<br>☐ Create Shared Assembly<br>☐ Enable .NET Remoting<br>Type Safe API<br>☐ Enable type safe API |

## API Selection for C++ Shared Library


API selection

C++ Shared Library API

[✓ Create all interfaces]

Create interface that uses the mwArray API

Create interface that uses the MATLAB Data API

- **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

`libraryCompiler`

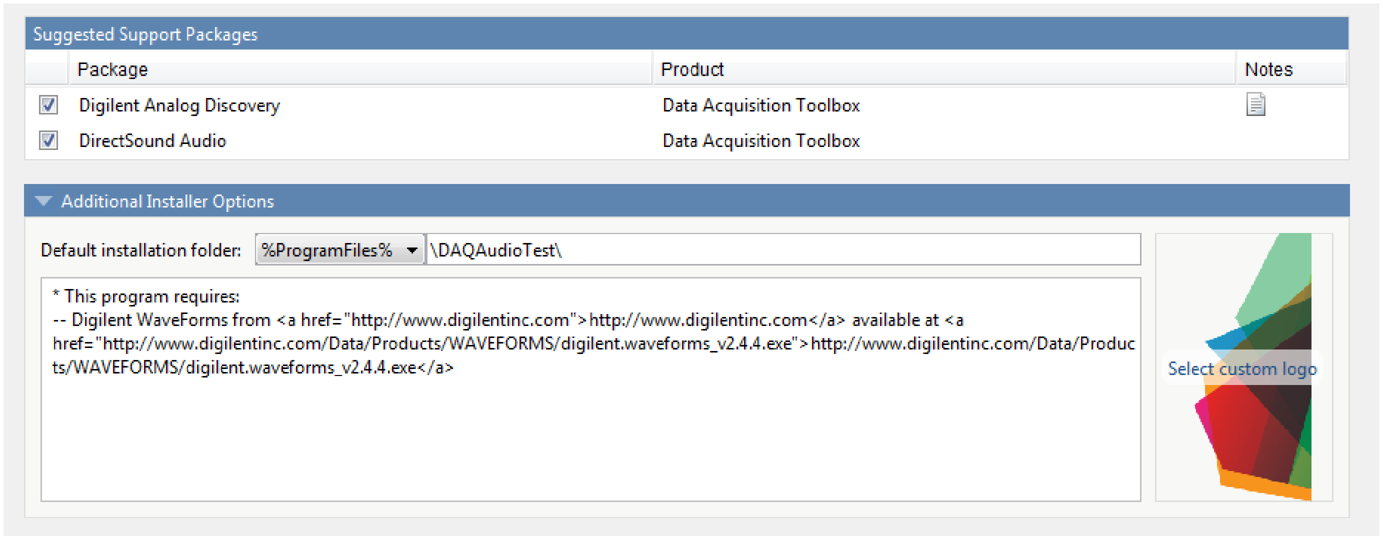
## More About

- “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.



| | Package | Product | Notes |
|---|---|---|---|
| ✓ | Digilent Analog Discovery | Data Acquisition Toolbox | 🗎 |
| ✓ | DirectSound Audio | Data Acquisition Toolbox | |

Default installation folder: %ProgramFiles% \DAQAudioTest\

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

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

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

Deselect support packages that are not required by your application.

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

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

## Using the Command Line

Many MATLAB toolboxes use support packages to interact with hardware or to provide additional processing capabilities. If your MATLAB code uses a toolbox with an installed support package, use the `-a` flag with `mcc` command when packaging your MATLAB code to specify supporting files in the support package folder. For example, if your function uses the `OS Generic Video Interface` support package, run the following command:

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

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

# Advanced Uses of the Command Line Compiler

## Simplify Compilation Using Macros

| In this section… |
|---|
| “Macros” on page 10-2 |
| “Working With Macros” on page 10-2 |

### Macros

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

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

| **Macro** | **Bundle** | **Creates** | **Option Equivalence**<br><br>**`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 “Deployable Archive”.

## mcc Command Arguments Listed Alphabetically

| Option | Description | Comment |
|---|---|---|
| `-?` | Display help message. | Cannot be used in a `deploytool` app. |
| `-a` *`path`* | Add `path` to the deployable archive. | If you specify a folder name, all files in the folder are added. If you use a wildcard (*), all files matching the wildcard are added. |
| `-b` | Generate Excel compatible formula function. | Requires MATLAB Compiler for Excel add-ins. Cannot be used in a `deploytool` app. |
| `-B` *`filename`*`[:arg[,arg]]` | Replace `-B filename` on the `mcc` command line with the contents of `filename`. | The file should contain only `mcc` command-line options. These are MathWorks included options files:<br>• `-B csharedlib:foo` (C shared library)<br>• `-B cpplib:foo` (C++ library)<br><br>Cannot be used in a `deploytool` app. |
| `-c` | Generate C wrapper code. | Equivalent to `-T codegen`. |
| `-C` | Direct `mcc` to not embed the deployable archive in generated binaries. | |
| `-d` *`directory`* | Place output in specified folder. | The specified folder must already exist. Cannot be used in a `deploytool` app. |
| `-e` | Suppresses appearance of the MS-DOS Command Window when generating a standalone application. | Use *`-e`* in place of the *`-m`* option. Available for Windows only. Use with `-R` option to generate error logging. Equivalent to `-W WinMain -T link:exe`. Cannot be used in a `deploytool` app.<br><br>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. |
| `-g` | Generate debugging information. | |
| `-G` | Same as `-g`. | |
| `-I` *`directory`* | Add folder to search path for MATLAB files. | |
| `-K` | Directs `mcc` to not delete output files if the compilation ends prematurely, due to error. | Default behavior is to dispose of any partial output if the command fails to execute successfully. |
| `-l` | Create a function library. | Equivalent to `-W lib -T link:lib`. Cannot be used in a `deploytool` app. |

| Option | Description | Comment |
|---|---|---|
| `-m` | Generate a standalone application. | Equivalent to `-W main -T link:exe`. Cannot be used in a `deploytool` app. |
| `-M` *`string`* | Pass string to `mbuild`. | Use to define compile-time options. |
| `-n` | Automatically treat numeric inputs as MATLAB doubles. | Cannot be used in a `deploytool` app. |
| `-N` | Clear the path of all but a minimal, required set of folders. | Uses the following folders:<br>• *`matlabroot`*`\toolbox\matlab`<br>• *`matlabroot`*`\toolbox\local`<br>• *`matlabroot`*`\toolbox\compiler`<br>• *`matlabroot`*`\toolbox\shared\bigdata` |
| `-o` *`outputfile`* | Specify name of final output file. | Adds appropriate extension. Cannot be used in a `deploytool` app. |
| `-p` *`directory`* | Add folder to compilation path in an order-sensitive context. | Requires `-N` option. |
| `-R` *`option`* | Specify run-time options for MATLAB Runtime. | Valid only for standalone applications using MATLAB Compiler.<br><br>*`option`* `=` `-nojvm`, `-nodisplay`, `'-logfile` *`filename`*`'`, `-startmsg`, and `-completemsg` *`filename`* |
| `-s` | Obfuscate folder structures and file names in the deployable archive (`.ctf` file) from the end user. | |
| `-S` | Create singleton MATLAB Runtime. | Default for generic COM components. Valid for Microsoft Excel and Java packages. |
| `-T` | Specify the output target phase and type. | Default is `codegen`. Cannot be used in a `deploytool` app. |
| `-u` | Registers COM component for current user only on development machine. | Valid only for generic COM components and Microsoft Excel add-ins. |
| `-v` | Verbose; display compilation steps. | |
| `-w` *`option`* | Display warning messages. | *`option`* = `list`, *`level`*, or *`level`*:*`string`*<br><br>where<br><br>*`level`* = disable, enable, error, off:*`string`*, or on:*`string`* |
| `-W` *`type`* | Control the generation of function wrappers. | *`type`* = `main cpplib:<string> lib:<string> none com:compname,clname,version`<br><br>Cannot be used in a `deploytool` app. |

| Option | Description | Comment |
|---|---|---|
| `-X` | Ignore data files detected by dependency analysis. | For more information, see “Dependency Analysis Using MATLAB Compiler”. |
| `-Y licensefile` | Use `licensefile` when checking out a MATLAB Compiler license. | The `-Y` flag works only with the command-line mode.<br><br>`>>!mcc -m foo.m -Y license.lic` |
| `-Z` *`option`* | Specify method of including support packages. | *`option`* = `'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

## MATLAB Runtime Startup Options

### Retrieve MATLAB Runtime Startup Options

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

| Function and Signature | When to Use | Return Value |
|---|---|---|
| `bool mclIsMCRInitialized()` | Use `mclIsMCRInitialized()` to determine whether or not the MATLAB Runtime has been properly initialized. | Boolean (`true` or `false`). Returns `true` if MATLAB Runtime is already initialized, else returns `false`. |
| `bool mclIsJVMEnabled()` | Use `mclIsJVMEnabled()` to determine if the MATLAB Runtime 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`.<br><br>**Note** `false` is always returned on Windows systems since the `-nodisplay` option is not supported on Windows systems.<br><br>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);
```

## 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 12-7.

### 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 12-7

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

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

Functions that cannot be compiled fall into the following categories:

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

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

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

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

#### List of Unsupported Functions and Programs

`add_block`

`add_line`

`checkcode`

`close_system`

`colormapeditor`

`commandwindow`

Control System Toolbox™ prescale GUI

`createClassFromWsdl`

`dbclear`

`dbcont`

`dbdown`

`dbquit`

`dbstack`

`dbstatus`

`dbstep`

`dbstop`

`dbtype`

`dbup`

`delete_block`

`delete_line`

`depfun`

`doc`

`echo`

`edit`

`fields`

`figure_palette`

`get_param`

`help`

`home`

`inmem`

`keyboard`

`linkdata`

`linmod`

`matlab.unittest.TestSuite.fromProject`

`mislocked`

`mlock`

`more`

`munlock`

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

13

# Functions

## %#function

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

### Syntax

`%#function` *`function1`* `[`*`function2 ... functionN`*`]`

`%#function` *`object_constructor`*

### Description

The `%#function` pragma informs MATLAB Compiler that the specified function(s) will be called through an `feval`, `eval`,Handle Graphics callback, or objects loaded from MAT-files.

Use the `%#function` pragma in standalone applications to inform MATLAB Compiler that the specified function(s) should be included in the compilation, whether or not MATLAB Compiler's dependency analysis detects the function(s). It is also possible to include objects by specifying the object constructor.

Without this pragma, the product's dependency analysis will not be able to locate and compile all MATLAB files used in your application. This pragma adds the top-level function as well as all the local functions in the file to the compilation.

### Examples

#### Example 1

```
 function foo
   %#function bar

      feval('bar');

   end %function foo
```

By implementing this example, MATLAB Compiler is notified that function `bar` will be included in the compilation and is called through `feval`.

#### Example 2

```
function foo
   %#function bar foobar

      feval('bar');
      feval('foobar');

   end %function foo
```

In this example, multiple functions (`bar` and `foobar`) are included in the compilation and are called through `feval`.

### Example 3

```
function foo
   %#function ClassificationSVM

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

    end %function foo
```

In this example, an object from the class `ClassificationSVM` is loaded from a MAT-file. For more information, see “MATLAB Data Files in Compiled Applications”.

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

**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('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\R2022a\toolbox\toolbox\compiler\packagingRes
             InstallerIcon: 'C:\Program Files\MATLAB\R2022a\toolbox\compiler\packagingResources\c
             InstallerLogo: 'C:\Program Files\MATLAB\R2022a\toolbox\compiler\packagingResources\c
                AuthorName: 'Frog'
               AuthorEmail: ''
             AuthorCompany: 'Boston Common'
                   Summary: 'Generates a magic square.'
               Description: ''
         InstallationNotes: ''
                  Shortcut: ''
                   Version: '1.0'
             InstallerName: 'MagicSquare_Installer'
           ApplicationName: 'MagicSquare_Generator'
                 OutputDir: '.\MagicSquare_Generatorinstaller'
    DefaultInstallationDir: 'C:\Program Files\MagicSquare_Generator'
```

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

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

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

### Create Installer Using Files

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

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

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

Build a standalone application using the `mcc` command.

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

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

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

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

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\R2022a\toolbox\toolbox\compiler\packagingRes
             InstallerIcon: 'C:\Program Files\MATLAB\R2022a\toolbox\compiler\packagingResources\
             InstallerLogo: 'C:\Program Files\MATLAB\R2022a\toolbox\compiler\packagingResources\
                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.

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

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

Example: `'MagicSquare_Generator'`

Data Types: `char` | `string`

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

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

Example: `'Boston Common'`

Data Types: `char` | `string`

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

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

Example: `'frog@bostoncommon.com'`

Data Types: `char` | `string`

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

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

Example: `'Frog'`

Data Types: `char` | `string`

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

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

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

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

Example: On Windows: `C:\Program Files\MagicSquare_Generator`

Data Types: `char` | `string`

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

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

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

Data Types: `char` | `string`

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

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

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

Data Types: `char` | `string`

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

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

The default path is:

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

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

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

Path to an image file used as the installer's logo, specified as a character vector or a string scalar. The logo will be resized to 150 pixels by 340 pixels.

The default path is:

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

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

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

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

Example: `'MagicSquare_Installer'`

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

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

The default path is:

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

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

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

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

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

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

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

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

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

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

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

Example: `'installer'`

Data Types: `char` | `string`

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

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

Example: `'.\mymagic.exe'`

Data Types: `char` | `string`

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

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

Example: `'Generates a magic square.'`

Data Types: `char` | `string`

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

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

Example: `'2.0'`

Data Types: `char` | `string`

## See Also

`compiler.package.InstallerOptions` | `mcc`

**Introduced in R2020a**

## compiler.package.InstallerOptions

Options for creating MATLAB Compiler package installers

### Syntax

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

### Description

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

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

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

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

### Examples

#### Create an Installer Options Object Using Results

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

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

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

```
results = 

  Results with properties:
```

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

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

```
opts = 

  InstallerOptions with properties:

           RuntimeDelivery: 'web'
           InstallerSplash: 'C:\Program Files\MATLAB\R2022a\toolbox\toolbox\compiler\packagingRes
             InstallerIcon: 'C:\Program Files\MATLAB\R2022a\toolbox\compiler\packagingResources\c
             InstallerLogo: 'C:\Program Files\MATLAB\R2022a\toolbox\compiler\packagingResources\c
                AuthorName: 'Frog'
               AuthorEmail: ''
             AuthorCompany: ''
                   Summary: ''
               Description: ''
         InstallationNotes: ''
                  Shortcut: ''
                   Version: ''
             InstallerName: 'MyAppInstaller'
           ApplicationName: 'magicsquare'
                 OutputDir: '.\magicsquare'
    DefaultInstallationDir: 'C:\Program Files\magicsquare'
```

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

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

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

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

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

### Create an Installer Options Object Using Application Name

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

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

```
opts = 

  InstallerOptions with properties:

           RuntimeDelivery: 'web'
           InstallerSplash: 'C:\Program Files\MATLAB\R2022a\toolbox\toolbox\compiler\packagingRes
             InstallerIcon: 'C:\Program Files\MATLAB\R2022a\toolbox\compiler\packagingResources\c
             InstallerLogo: 'C:\Program Files\MATLAB\R2022a\toolbox\compiler\packagingResources\c
```

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

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

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

## Input Arguments

**`results` — Build results object**
`Results` object

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

**appName — Name of the installed application**
character vector | string scalar

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

Example: `'MagicSquare_Generator'`

Data Types: `char` | `string`

### Name-Value Pair Arguments

Specify optional 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.

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

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

Example: `'MagicSquare_Generator'`

Data Types: `char` | `string`

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

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

Example: `'Boston Common'`

Data Types: `char` | `string`

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

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

Example: `'frog@bostoncommon.com'`

Data Types: `char` | `string`

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

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

Example: `'Frog'`

Data Types: `char` | `string`

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

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

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

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

Example: On Windows: `C:\Program Files\MagicSquare_Generator`

Data Types: `char` | `string`

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

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

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

Data Types: `char` | `string`

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

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

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

Data Types: `char` | `string`

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

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

The default path is:

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

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

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

Path to an image file used as the installer's logo, specified as a character vector or a string scalar. The logo will be resized to 150 pixels by 340 pixels.

The default path is:

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

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

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

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

Example: `'MagicSquare_Installer'`

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

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

The default path is:

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

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

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

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

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

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

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

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

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

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

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

Example: `'installer'`

Data Types: `char` | `string`

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

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

Example: `'.\mymagic.exe'`

Data Types: `char` | `string`

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

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

Example: `'Generates a magic square.'`

Data Types: `char` | `string`

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

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

Example: `'2.0'`

Data Types: `char` | `string`

## Output Arguments

**`opts` — Installer options object**
`InstallerOptions` object

Installer options, returned as an `InstallerOptions` object.

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

**Introduced in R2020a**

# ctfroot

Location of files related to deployed application

## Syntax

`root = ctfroot`

## Description

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

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

## Examples

### Determine location of deployable archive

```
appRoot = ctfroot;
```

## Output Arguments

**`root` — Path to expanded deployable archive**
character vector

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

**Introduced in R2006a**

## deploytool

Open a list of application deployment apps

### Syntax

```
deploytool
deploytool project_name
```

### Description

`deploytool` opens a list of application deployment apps.

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

### Examples

#### Open a List of Application Deployment Apps

Open the list of apps.

```
deploytool
```

### Input Arguments

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

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

### Compatibility Considerations

**`-build` and `-package` options will be removed**
*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.

**Introduced in R2006b**

# figToImStream

Stream figure as byte array encoded in specified format

## Syntax

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

## Description

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

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

## Examples

### Convert current figure to a signed PNG formatted byte array

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

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

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

## Input Arguments

### Name-Value Pair Arguments

Specify optional 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 | bmp | gif

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

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

#### `outputType` — Type of bytes to store the image stream
int8 (default) | uint8

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

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

## Output Arguments

#### `output` — Encoded figure data
byte array

Encoded figure data returned as a byte array.

**Introduced in R2009b**

# getmcruserdata

Retrieve MATLAB array value associated with a given key

## Syntax

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

## Description

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

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

## Examples

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

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

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

## Input Arguments

**key — Key associated with MATLAB data**
string

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

## Output Arguments

**value — Value of MATLAB data**
any MATLAB data type including matrices, cell arrays, and Java objects

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

## See Also

`setmcruserdata`

**Introduced in R2008b**

# isdeployed

Determine whether code is running in deployed or MATLAB mode

## Syntax

`x = isdeployed`

## Description

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

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

## Examples

### Protect Use of `ADDPATH`

The path of a deployed application is fixed at compile time and cannot change. Use `isdeployed` to ensure that the application uses path modifying functions such as `addpath` before deployment.

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

### Send Data to Printer

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

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

### Display Documentation

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

```
if ~isdeployed
    doc(mfile);
else
```

```
   web('https://www.mathworks.com/support.html');
end
```

## Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

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

## See Also

`ismcc` | `mcc` | `deploytool`

**Topics**
"Write Deployable MATLAB Code"
"MATLAB Data Files in Compiled Applications"

# 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.

### Compatibility Considerations

**`-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.

**Introduced in R2013b**

# mbuild

Compile and link source files against MATLAB generated shared libraries

## Syntax

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

## Description

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

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

## Supported Source File Types

Supported types of source files are:

- `.c`
- `.cpp`

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

## Options

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

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

**Introduced before R2006a**

## mcc

Compile MATLAB functions for deployment

### Syntax

```
mcc options mfilename1 mfilename2...mfilenameN

mcc -l options mfilename1 mfilename2...mfilenameN

mcc -W cpplib:library_name[,{all|legacy|generic}] options mfilename1
mfilename2...mfilenameN

mcc -W com:component_name,className -T link:lib options
class{className:mfilename1,mfilename2,...,mfilenameN}

mcc -W dotnet:assembly_name,className,framework_version,security,remote_type
-T link:lib options mfilename1 mfilename2...mfilenameN
mcc -W dotnet:assembly_name,className,framework_version,security,remote_type
-T link:lib options class{className:mfilename1,mfilename2,...,mfilenameN}

mcc -W java:packageName,className options mfilename1 mfilename2...mfilenameN
mcc -W java:packageName,className options
class{className:mfilename1,mfilename2,...,mfilenameN}

mcc -W python:namespace.packageName -T link:lib options mfilename1
mfilename2...mfilenameN

mcc -W CTF:archive_name -U options mfilename1 mfilename2...mfilenameN

mcc -W mpsxl:addin_name,className,version input_marshaling_flags
output_marshaling_flags -T link:lib options mfilename1
mfilename2...mfilenameN
```

### Description

#### General Usage

`mcc options mfilename1 mfilename2...mfilenameN` compiles the functions as specified by the options. The options used depend on the intended results of the compilation.

For information on compiling:

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

#### C Shared Library

`mcc -l options mfilename1 mfilename2...mfilenameN` compiles the listed functions into a C shared library and generates C wrapper code for integration with other applications.

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

#### C++ Shared Library

`mcc -W cpplib:`*`library_name`*`[,{all|legacy|generic}] options mfilename1 mfilename2...mfilenameN` compiles the listed functions into a C++ shared library and generates C++ wrapper code for integration with other applications.

- *`library_name`* — Specifies the name of the shared library.
- `all`— Generates shared libraries using both the `mwArray` API and the generic interface that uses the MATLAB Data API. This is the default.
- `legacy`— Generates shared libraries using the `mwArray`API.
- `generic`— Generates shared libraries using the MATLAB Data API.

#### COM Component

`mcc -W com:`*`component_name`*`,`*`className`*` -T link:lib options class{className:mfilename1,mfilename2,...,mfilenameN}` compiles the listed functions into a generic Microsoft COM component.

---

**Note** File names listed in the `class{___}` argument must be separated by commas.

---

- *`component_name`* — Specifies the name of the COM component.
- *`className`* — Specifies the name of the class.

  ---

  **Note** You can include multiple class specifiers by adding additional `class{___}` arguments.

  ---

#### .NET Assembly

`mcc -W dotnet:`*`assembly_name`*`,`*`className`*`,`*`framework_version`*`,`*`security`*`,`*`remote_type`*` -T link:lib options mfilename1 mfilename2...mfilenameN` creates a .NET assembly with a single class from the specified files.

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

  - `0.0` — Use the latest supported version on the target machine.
  - *`version_major.version_minor`* — Use a specific version of the framework.

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

  - To create a private assembly, specify `Private`.
  - To create a shared assembly, specify the full path to the encryption key file used to sign the assembly.

- *`remote_type`* — Specifies the remoting type of the assembly. Values are `remote` and `local`.

`mcc -W dotnet:`*`assembly_name`*`,`*`className`*`,`*`framework_version`*`,`*`security`*`,`*`remote_type`* `-T link:lib options class{className:mfilename1,mfilename2,...,mfilenameN}` creates a .NET assembly with multiple classes from the specified files.

**Note** File names listed in the `class{___}` argument must be separated by commas.

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

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

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

**Java Package**

`mcc -W java:`*`packageName`*`,`*`className`* `options mfilename1 mfilename2...mfilenameN` creates a Java package from the specified files.

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

`mcc -W java:`*`packageName`*`,`*`className`* `options class{className:mfilename1,mfilename2,...,mfilenameN}` creates a Java package with multiple classes from the specified files.

**Note** File names listed in the `class{___}` argument must be separated by commas.

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

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

**Note** You can include multiple class specifiers by adding additional `class{___}` arguments.

#### Python Package

`mcc -W python:namespace.packageName -T link:lib options mfilename1 mfilename2...mfilenameN` creates a Python package from the specified files.

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

#### Deployable Archive for MATLAB Production Server

`mcc -W CTF:archive_name -U options mfilename1 mfilename2...mfilenameN` instructs the compiler to create a deployable archive (`.ctf` file) for use with a MATLAB Production Server instance.

The syntax also creates the server-side deployable archive (`.ctf` file) for Microsoft Excel add-ins.

#### Excel Add-In for MATLAB Production Server

`mcc -W mpsxl:addin_name,className,version input_marshaling_flags output_marshaling_flags -T link:lib options mfilename1 mfilename2...mfilenameN` creates a client-side Microsoft Excel add-in from the specified files that can be used to send requests to MATLAB Production Server from Excel. Creating the client-side add-in *must* be preceded by creating a server-side deployable archive (`.ctf` file) from the specified files. A purely client side add-in is not viable.

- `addin_name` — Specifies the name of the add-in.
- `className` — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the `addin_name` as the default.
- `version` — Specifies the version of the add-in specified as `major.minor`.
  - `major` — Specifies the major version number. If you do not specify a version number, `mcc` uses the latest version.
  - `minor` — Specifies the minor version number. If you do not specify a version number, `mcc` uses the latest version.
- `input_marshaling_flags` — Specifies options for how data is marshaled between Microsoft Excel and MATLAB.
  - `-replaceBlankWithNaN` — Specifies that a blank in Microsoft Excel is mashaled into NaN in MATLAB. If you do not specify this flag, blanks are marshaled into 0.
  - `-convertDateToString` — Specifies that dates in Microsoft Excel are marshaled into MATLAB character vectors. If you do not specify this flag, dates are marshaled into MATLAB doubles.
- `output_marshaling_flags` — Specifies options for how data is marshaled between MATLAB and Microsoft Excel.
  - `-replaceNaNWithZero` — Specifies that NaN in MATLAB is marshaled into a 0 in Microsoft Excel. If you do not specify this flag, NaN is marshalled into `#QNAN` in Visual Basic.

- `-convertNumericToDate` — Specifies that MATLAB numeric values are marshaled into Microsoft Excel dates. If you do not specify this flag, Microsoft Excel does not receive dates as output.

## Examples

### Create a standalone application and include MATLAB preferences

```
mcc -m helloWorld.m -a C:\Users\someuser\AppData\Roaming\MathWorks\MATLAB\R2022a\matlab.mlsettin
```

### Create a C shared library

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

### Create a C shared library with a system-level file version number (Windows only)

Create a C shared library in Windows with version number `4.3.1.7`.

```
mcc -W 'lib:myCSharedLib,version=4.3.1.7' -T link:lib mymagic.m
```

### Create a C++ shared library

Use the `mwArray` API

```
mcc -W 'cpplib:mymagic,legacy' mymagic.m
```

Use the MATLAB Data API

```
mcc -W 'cpplib:mymagic,generic' mymagic.m
```

Use both the `mwArray` API and the MATLAB Data API

```
mcc -W 'cpplib:mymagic,all' mymagic.m
```

OR

```
mcc -W 'cpplib:mymagic' mymagic.m
```

### Create a C++ shared library with a system-level file version number (Windows only)

Create a C++ shared library in Windows with version number `3.7.1.5`.

```
mcc -W 'cpplib:mymagic,all,version=3.7.1.5' -T link:lib mymagic.m
```

### Create a COM component

Create a COM component in Windows with version number `7.10.1.3`.

```
mcc -W 'com:myCOMComponent,myClass,version=7.10.1.3' -T link:lib class{myClass:mymagic.m}
```

### Create a Java package containing multiple classes

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

### Create a Python package

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

#### Create a deployable archive for MATLAB Production Server

```
mcc -W CTF:myDeployableArchive -U mymagic.m
```

#### Create an Excel add-in for MATLAB Production Server

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

## Input Arguments

#### `mfilename1 mfilename2...mfilenameN` — Files to be compiled

list of file names

One or more files to be compiled, specified as a space-separated list of file names.

#### `class{`*`className`*`:mfilename1,mfilename2,...,mfilenameN}` — Files to be included in a class

list of file names

One or more files to be included in the class *`className`*, specified as a comma-separated list of file names. You can repeat this argument to include multiple class specifiers. The argument applies only to the COM component, Java package, and .NET assembly targets.

#### `options` — Options for customizing the output

-a | -b | -B | -c | -C | -d | -f | -g | -G | -I | -K | -m | -M | -n | -N | -o | -p | -r | -R | -s | -S | -T | -u | -U | -v | -w | -W | -X | -Y | -Z

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

- **`-a`**

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

  Also, add MATLAB preferences to a deployed application using `-a path\mymatlab.mlsettings` to specify the preferences to be added.

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

  If a folder name is specified with the `-a` option, the entire contents of that folder are added recursively to the deployable archive. For example,

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

  specifies that all files in `testdir`, as well as all files in its subfolders, are added to the deployable archive. The folder subtree in `testdir` is preserved in the deployable archive.

  If the filename includes a wildcard pattern, only the files in the folder that match the pattern are added to the deployable archive and subfolders of the given path are not processed recursively. For example,

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

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

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

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

**Note** * is the only supported wildcard.

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

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

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

**Note** If you use the `-a` flag to include custom Java classes, standalone applications work without any need to change the `classpath` as long as the Java class is not a member of a package. The same applies for JAR files. However, if the class being added is a member of a package, the MATLAB code needs to make an appropriate call to `javaaddpath` to update the `classpath` with the parent folder of the package.

- **-b**

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

- **-B**

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

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

  The bundle `filename` should contain only `mcc` command-line options and corresponding arguments and/or other file names. The file might contain other `-B` options. A bundle can include replacement parameters for compiler options that accept names and version numbers. See “Using Bundles to Build MATLAB Code” on page 10-4.

- **-c**

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

- **`-C`**

  Do not embed the deployable archive in binaries.

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

  ---
- **`-d`**

  Place output in a specified folder. Use

  ```
  -d outFolder
  ```

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

  Use `-e` in place of the `-m` option to generate a standalone Windows application that does not open a Windows command prompt on execution. `-e` is equivalent to `-W WinMain -T link:exe`.

  This option works only on Windows operating systems.
- **`-f`**

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

  ```
  -f filename
  ```

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

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

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

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

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

  If used in conjunction with the `-N` option, the `-I` option adds the folder to the compilation path in the same position where it appeared in the MATLAB path rather than at the head of the path.
- **`-K`**

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

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

- **`-m`**

  Direct `mcc` to generate a standalone application.

- **`-M`**

  Define compile-time options. Use

  ```
  -M string
  ```

  to pass `string` directly to `mbuild`. This option provides a useful mechanism for defining compile-time options, for example, `-M "-Dmacro=value"`.

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

  To pass options such as `/bigobj`, delineate the string according to your platform.

  | Platform | Syntax |
  |---|---|
  | MATLAB | `-M 'COMPFLAGS=$COMPFLAGS /bigobj'` |
  | Windows command prompt | `-M COMPFLAGS="$COMPFLAGS /bigobj"` |
  | Linux and macOS command line | `-M CFLAGS='$CFLAGS /bigobj'` |

- **`-n`**

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

- **`-N`**

  Passing `-N` clears the path of all folders except the following core folders (this list is subject to change over time):

  - *`matlabroot`*`\toolbox\matlab`
  - *`matlabroot`*`\toolbox\local`
  - *`matlabroot`*`\toolbox\compiler`
  - *`matlabroot`*`\toolbox\shared\bigdata`

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

  When using the `–N` option, use the `–I` option to force inclusion of a folder, which is placed at the head of the compilation path. Use the `–p` option to conditionally include folders and their subfolders; if they are present in the MATLAB path, they appear in the compilation path in the same order.

- **`-o`**

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

```
-o outputfile
```

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

- **`-p`**

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

  ```
  -N -p directory
  ```

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

  - If a folder is included with `-p` that is on the original MATLAB path, the folder and all its subfolders that appear on the original path are added to the compilation path in the same order.
  - If a folder is included with `-p` that is not on the original MATLAB path, that folder is ignored. (You can use `-I` to force its inclusion.)

- **`-r`**

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

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

- **`-R`**

  Provide MATLAB Runtime options that are passed to the application at initialization time.

  **Note** This option is relevant only when building standalone applications or Excel add-ins using MATLAB Compiler.

  The syntax is as follows:

  ```
  -R option
  ```

| Option | Description | Target |
|---|---|---|
| `'-logfile,`*`filename`*`'` | Specify a log file name. The file is created in the application folder at runtime. Option must be in single quotes. Use double quotes when executing the command from a Windows Command Prompt. | MATLAB Compiler |
| `-nodisplay` | Suppress the MATLAB `nodisplay` run-time warning. | MATLAB Compiler |
| `-nojvm` | Do not use the Java Virtual Machine (JVM). | MATLAB Compiler |
| `-startmsg` | Customizable user message displayed at initialization time. | MATLAB Compiler Standalone Applications |

| Option | Description | Target |
|---|---|---|
| `-completemsg` | Customizable user message displayed when initialization is complete. | MATLAB Compiler Standalone Applications |
| `-singleCompThread` | Limit MATLAB to a single computational thread. | MATLAB Compiler |
| `-softwareopengl` | Use Mesa Software OpenGL® for rendering. | MATLAB Compiler |

**Caution** When running on macOS, if you use `-nodisplay` as one of the options included in `mclInitializeApplication`, then the call to `mclInitializeApplication` must occur before calling `mclRunMain`.

**Note** If you specify the `-R` option for libraries created from MATLAB Compiler SDK, `mcc` still compiles and generates the results, but the `-R` option doesn't apply to these libraries and does not do anything.

- **`-s`**

  Obfuscate folder structures and file names in the deployable archive (`.ctf` file) from the end user. Optionally encrypt additional file types.

  The `-s` option directs `mcc` to place user code and data contained in `.m`, `.p`, v7.3 `.mat`, and MEX files into a user package within the CTF. During runtime, MATLAB code and data is decrypted and loaded directly from the user package rather than extracted to the file system. MEX files are temporarily extracted from the user package before being loaded.

  To manually include additional file types in the user package, add each file type in a separate extension tag to the file *`matlabroot`*`/toolbox/compiler/advanced_package_supported_files.xml`.

  The following is not supported:

  - `ver` function
  - Out-of-process MATLAB Runtime ( C++ shared library for MATLAB Data Array)
  - Out-of-process MEX file execution (`mexhost`, `feval`, `matlab.mex.MexHost`)
- **`-S`**

  The standard behavior for the MATLAB Runtime is that every instance of a class gets its own MATLAB Runtime context. The context includes a global MATLAB workspace for variables, such as the path and a base workspace for each function in the class. If multiple instances of a class are created, each instance gets an independent context. This ensures that changes made to the global or base workspace in one instance of the class does not affect other instances of the same class.

  In a singleton MATLAB Runtime, all instances of a class share the context. If multiple instances of a class are created, they use the context created by the first instance which saves startup time and some resources. However, any changes made to the global workspace or the base workspace by one instance impacts all class instances. For example, if `instance1` creates a global variable `A` in a singleton MATLAB Runtime, then `instance2` can use variable `A`.

Singleton MATLAB Runtime is only supported by the following products on these specific targets:

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

- **`-T`**

  Specify the output target phase and type.

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

  | Target | Description |
  |---|---|
  | `compile:exe` | Generate a C/C++ wrapper file, and compile C/C++ files to an object form suitable for linking into a standalone application. |
  | `compile:lib` | Generate a C/C++ wrapper file, and compile C/C++ files to an object form suitable for linking into a shared library or DLL. |
  | `link:exe` | Same as `compile:exe` and also link object files into a standalone application. |
  | `link:lib` | Same as `compile:lib` and also link object files into a shared library or DLL. |

- **`-u`**

  Register COM component for the current user only on the development machine. The argument applies only to the generic COM component and Microsoft Excel add-in targets.
- **`-U`**

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

  Display the compilation steps, including:

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

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

  Display warning messages. Use the syntax

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

to control the display of warnings.

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

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

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

```
if isdeployed
    warning off
end
```

To turn on warnings for deployed applications, you write:

```
if isdeployed
    warning on
end
```

You can also specify multiple `-w` options.

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

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

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

- **`-W`**

  Control the generation of function wrappers. Use the syntax

  ```
  -W type
  ```

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

| **Target** | **Syntax** |
|---|---|
| C Shared Library | `-W 'lib:`*`libName`*`'` |
| C++ Shared Library | `-W 'cpplib:`*`libName`*`[,{all|legacy|generic}]'` |
| COM Component | `-W 'com:`*`comName`*`,`*`className`*`'` |
| .NET Assembly | `-W 'dotnet:`*`assemblyName`*`,`*`className`*`,`*`frameworkVersion`*`,`*`security`*`,{remote|local}'` |
| Java Package | `-W 'java:`*`packageName`*`,`*`className`*`'` |
| Python Package | `-W 'python:`*`packageName`*`,`*`className`*`'` |

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

- `-X`

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

- **`-Y`**

  Use

  ```
  -Y license.lic
  ```

  to override the default license file with the specified argument.

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

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

- **`-Z`**

  Use

  `-Z` *`option`*

  to specify the method of adding support packages to the deployable archive.

| **Syntax** | **Description** |
|---|---|
| `-Z 'autodetect'` | The dependency analysis process detects and includes the required support packages automatically. This is the default behavior of mcc. |

| **Syntax** | **Description** |
|---|---|
| `-Z 'none'` | No support packages are included. Using this option can cause runtime errors. |
| `-Z` *`packagename`* | Only the specified support package is included. To specify multiple support packages, use multiple `-Z` inputs. |

**Note** To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

## Tips

- On Windows, you can generate a system-level file version number for your target file by appending `version=`*`version_number`* to the target generating `mcc` syntax. For an example, see "Create a C ++ shared library with a system-level file version number (Windows only)" on page 13-40.

  *`version_number`* — Specifies the version of the target file as *`major.minor.bug.build`* in the file system. You are not required to specify a version number. If you do not specify a version number, `mcc` sets the version number, by default, to `1.0.0.0`.

  - *`major`* — Specifies the major version number. If you do not specify a version number, `mcc` sets *`major`* to `1`.
  - *`minor`* — Specifies the minor version number. If you do not specify a version number, `mcc` sets *`minor`* to `0`.
  - *`bug`* — Specifies the bug fix maintenance release number. If you do not specify a version number, `mcc` sets *`bug`* to `0`.
  - *`build`* — Specifies build number. If you do not specify a version number, `mcc` sets *`build`* to `0`.

  This functionality is supported for C shared libraries, C++ shared libraries, COM components, .NET assemblies, and Excel add-ins for MATLAB Production Server in MATLAB Compiler SDK. For supported targets in MATLAB Compiler, see the **Tips** section in `mcc`.

## See Also

`mbuild`

**Introduced before R2006a**

# mcrinstaller

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

## Syntax

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

## Description

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

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

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

See “Install and Configure MATLAB Runtime”for more information about the MATLAB Runtime installer.

## Examples

### Find MATLAB Runtime Installer Location

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

```
mcrinstaller
```

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

For example, for R2018b, the path would be:

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

## Output Arguments

**`installer_path` — Full path to the installer**
character vector

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

**`major` — Major version number**
positive integer scalar

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

**`minor` — Minor version number**
positive integer scalar

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

**`platform` — Name of the current platform**
character vector

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

## See Also

`mcrversion` | `compiler.runtime.download`

**Topics**
"Install and Configure MATLAB Runtime"

# 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”

# 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.

## Compatibility Considerations

**`-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.

**Introduced in R2014a**

## setmcruserdata

Associate MATLAB data value with a key

### Syntax

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

### Description

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

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

### Examples

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

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

### Input Arguments

**value — Value of MATLAB data**
any MATLAB data type including matrices, cell arrays, and Java objects

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

**key — Key associated with MATLAB data**
string

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

### See Also

`getmcruserdata`

**Introduced in R2008a**

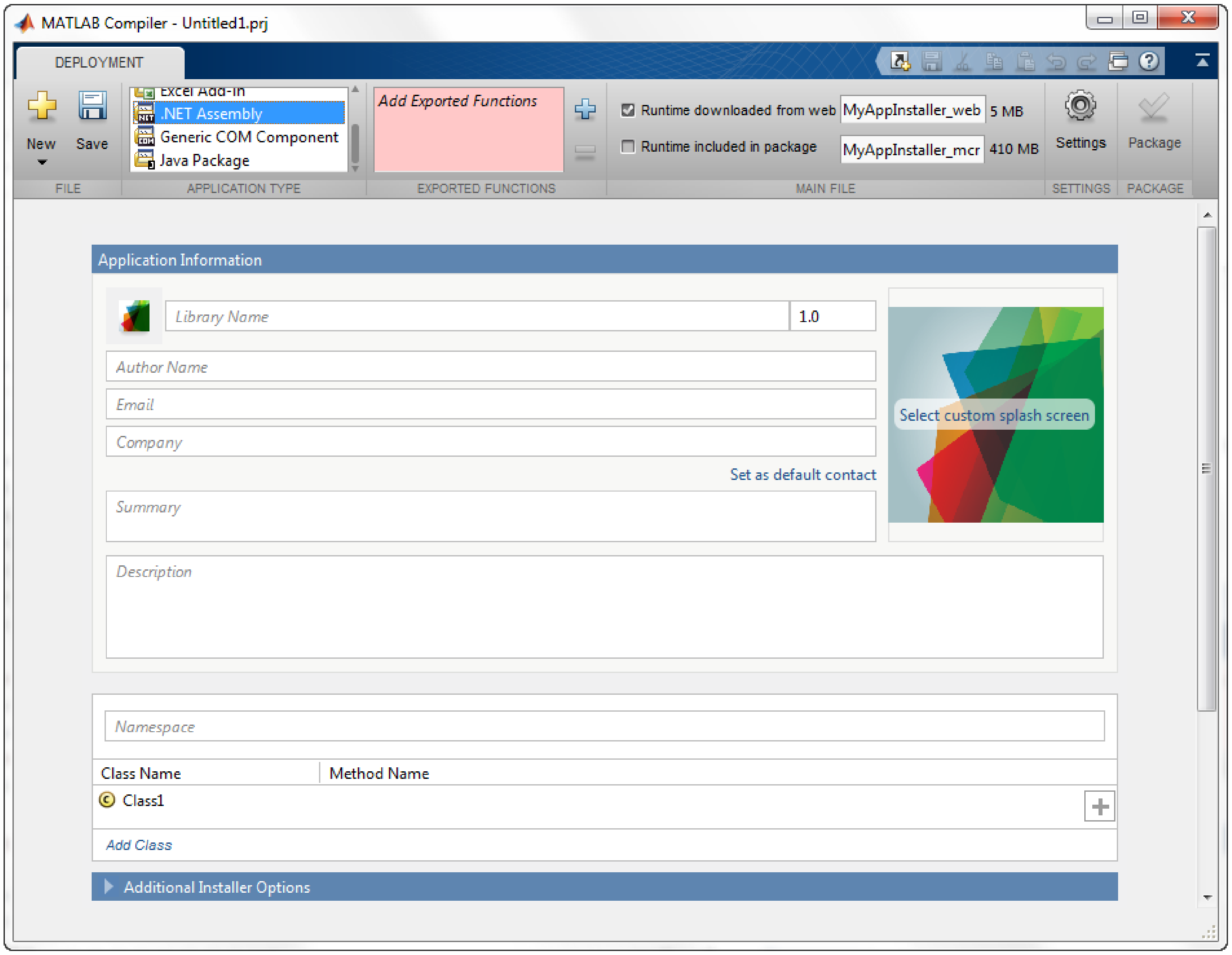
# Apps

## 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

`libraryCompiler`

## See Also

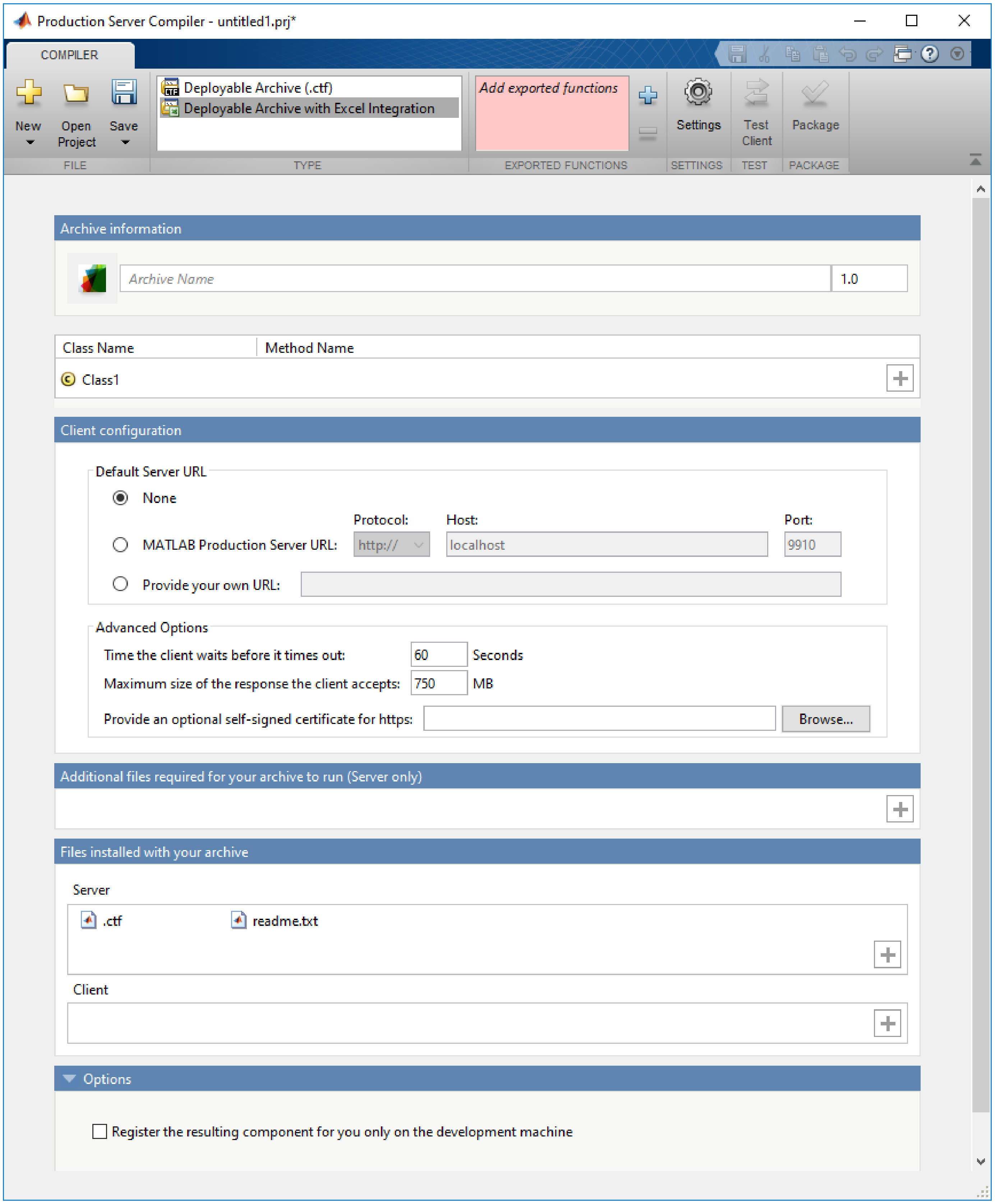
**Topics**
"Create Excel Add-In from MATLAB"
"Create a C Shared Library with MATLAB Code"
"Generate a C++ mwArray API Shared Library and Build a C++ Application"
"Generate a C++ MATLAB Data API Shared Library and Build a C++ Application"
"Generate .NET Assembly and Build .NET Application"
"Create a Generic 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.



Archive information

Archive Name: 1.0

| Class Name | Method Name |
|---|---|
| Class1 | |

Client configuration

Default Server URL

(●) None

( ) MATLAB Production Server URL: Protocol: http:// Host: localhost Port: 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 `productionServerCompiler`.

## Examples

- “Create a deployable archive for MATLAB Production Server” on page 13-41
- “Create and Install a Deployable Archive with Excel Integration for MATLAB Production Server”

## Parameters

**type — type of archive generated**
Deployable Archive | Deployable Archive with Excel Integration

Type of archive to generate as a character array.

**`exported functions` — functions to package**
list of character arrays

Functions to package as a list of character arrays.

**`archive information` — name of the archive**
character array

Name of the archive as a character array.

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

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

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

Optional files installed with archive as a list of files.

#### Settings

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

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

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

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

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

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

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

Folder where generated installers are stored as a character array.

## Programmatic Use

`productionServerCompiler`

## See Also

**Topics**
“Create a deployable archive for MATLAB Production Server” on page 13-41
“Create and Install a Deployable Archive with Excel Integration for MATLAB Production Server”

**Introduced in R2013b**