MATLAB® Compiler

The Language of Technical Computing

- Computation
- Visualization
- Programming

User's Guide

Version 4

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup

`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 Phone

508-647-7001 Fax

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

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB Compiler User's Guide

Printing History:	Sept 1995	First printing	
	March 1997	Second printing	
	Jan 1998	Third printing	Revised for Version 1.2
	Jan 1999	Fourth printing	Revised for Version 2.0 (Release 11)
	Sept 2000	Fifth printing	Revised for Version 2.1 (Release 12)
	Oct 2001	Online only	Revised for Version 2.3
	July 2002	Sixth printing	Revised for Version 3.0 (Release 13)

June 2004	Online only	Revised for Version 4.0 (Release 14)
August 2004	Online only	Revised for Version 4.0.1 (Release 14+)

Contents

1 Getting Started

2 Installation and Configuration

3 Compilation Process

4 Working with mcc

5 Stand-Alone Applications

6 Libraries

7 COM and Excel Components

8 Reference

A MATLAB Compiler Quick Reference

B Error and Warning Messages

C Troubleshooting

D C++ Utility Library Reference

Index

1

Getting Started

This chapter introduces the MATLAB Compiler and its uses. It also highlights differences between this version and previous versions.

Introduction

MATLAB® Compiler Version 4 takes M-files as input and generates redistributable, stand-alone applications or software components. These resulting applications and components are platform specific. The MATLAB Compiler can generate these kinds of applications or components:

- Stand-alone applications. Stand-alone applications do not require MATLAB at run-time; they can run even if MATLAB is not installed on the end-user's system.
- C and C++ shared libraries (dynamically linked libraries, or DLLs, on Microsoft Windows). These can be used without MATLAB on the end-user's system.
- Excel add-ins; requires MATLAB Builder for Excel
- COM objects; requires MATLAB Builder for COM

The MATLAB Compiler supports all the functionality of MATLAB, including objects. In addition, no special considerations are necessary for private and method functions; they are handled by the Compiler.

Note Some toolboxes will not compile with MATLAB Compiler 4. Particular functionality of some toolboxes will also not compile. For more information regarding the compilability of toolboxes, see the MATLAB Compiler product page on the Web.

Before You Begin

Before reading this book, you should already be comfortable writing M-files. If you are not, see Programming in the MATLAB documentation.

It is useful to note the distinction between the MATLAB Compiler and the MATLAB interpreter. The *MATLAB interpreter* refers to the application that accepts MATLAB commands, executes M-files and MEX-files, and behaves as described in the MATLAB documentation. When you use MATLAB, you are using the MATLAB interpreter. The *MATLAB Compiler* refers to this product, which takes M-files as input and generates stand-alone applications or software components. The MATLAB Compiler is invoked with the `mcc` command.

Note This book distinguishes references to the MATLAB Compiler by using the word “Compiler” with a capital C. References to “compiler” with a lowercase c refer to your C or C++ compiler.

Upgrading from Previous Compiler Releases

MATLAB Compiler 4 is compatible with previous releases of the Compiler. M-files that you compiled with a previous version of the MATLAB Compiler should compile with this version.

You can recompile any of your existing M-files that are compatible with MATLAB Release 14 products. There are no restrictions on the contents of your M-files other than compatibility with Release 14.

In Compiler 4, the generated API is different from previous versions of the Compiler. If you develop software components, you will need to adjust to the new API. See Appendix D, “C++ Utility Library Reference,” for more information.

Differences Between MATLAB Compiler 4 and Previous Versions of the MATLAB Compiler

This section highlights significant differences between MATLAB Compiler 4 and previous versions of the MATLAB Compiler.

Compiler 4 is a deployment tool for creating deliverables that can be distributed to other users. This version of the MATLAB Compiler fully supports all features of the MATLAB language including objects.

- Compiler 4 uses the new MATLAB Component Runtime (MCR) instead of the MATLAB C/C++ Math and Graphics Libraries. The MCR is a stand-alone set of shared libraries that enable the execution of encrypted M-files created using the MATLAB Compiler. For more information, see "Overview of the MATLAB Compiler Technology" on page 3-2.
- Compiler 4 only generates code for interface functions (wrappers), whereas previous versions generated code for the entire M-file. For more information on interface/wrapper differences, see "Wrapper Code Differences" on page 1-6.
- Compiler 4 is not intended for code translation, but rather, as a deployment tool. Some C code is generated to start the backend run-time environment. Previous versions of the MATLAB Compiler used to generate full C or C++ code, but to gain the ability to compile the entire MATLAB language, the product was changed.
- Compiler 4 has deprecated options that involve code generation and formatting. For more information, see "Deprecated Compiler Options" on page 1-7.
- Compiler 4 has deprecated some wrapper options and their associated bundle files. For more information, see "Deprecated Wrapper Options" on page 1-9.
- Compiler 4 does not use the `mglinstaller` for deploying applications. It has been replaced by the `MCRInstaller`.
- Compiler 4 is supported on Microsoft Windows and Linux only. We expect to add additional platforms in a future release.
- Compiler 4 includes several new options. For more information, see "New Compiler Options" on page 1-9.
- Compiler 4 does not include the MATLAB Add-in for Visual Studio.

- Compiler 4 does not speed up applications. There is no speed difference between a compiled application and one running in MATLAB. The compiled application will run as fast as MATLAB with the JIT Accelerator.
- Compiler 4 does not support the compilation of MEX-files and Simulink S-functions from M-functions because features in MATLAB 7 make this functionality redundant. The MATLAB JIT Accelerator makes compilation for speed obsolete, and the MATLAB `pcode` (preparsed code) function enables you to hide your proprietary algorithms.
- Compiler 4 has deprecated the set of imputed functions including `mbchar`, `mbcharscalar`, `mbcharvector`, `mbint`, `mbintscalar`, `mbintvector`, `mbreal`, `mbrealscalar`, `mbrealvector`, `mbscalar`, and `mbvector`. Compiler 4 makes the need for these functions. obsolete.
- In previous versions of the MATLAB Compiler, you needed to use `mccsavepath` if the MATLAB Compiler was going to be invoked from a shell (DOS or UNIX) prompt. With this release of the Compiler, this step is no longer needed. Consequently, `mccsavepath` is no longer available with Compiler 4.
- MATLAB does not support the loading of MATLAB Compiler-generated libraries via the `loadlibrary` function.

Wrapper Code Differences

Wrappers, or wrapper files, contain the required interface between the MATLAB Compiler-generated code and the supported component type such as executable or library. Compiler 4 only generates code for interface functions (wrappers), whereas previous versions generated code for the entire M-file. There are several differences to be aware of when calling Release 14 Compiler functions from C or C++:

- Since Compiler 4 does not use the MATLAB C/C++ Math and Graphics libraries, the various `mlf` functions previously available with the libraries are no longer available.
- The `initialize` routine now returns a status flag that can be used to test if the library was initialized properly.

Difference in the Exported Function Signature

The interface to the `mlf` functions generated by the Compiler from your M-file routines has changed in this version of the Compiler. Unlike previous versions

of the MATLAB Compiler, all the return values are passed as input to the function. The return value of these functions is `void`. The generic signature of the exported `mlf` functions is

- M-functions with no return values

```
void mlf<function-name>(<list_of_input_variables>);
```

- M-functions with at least one return value

```
void mlf<function-name>(int number_of_return_values,
<list_of_pointer_to_return_variables>,
<list_of_input_variables>);
```

Refer to the header file generated for your library for the exact signature of the exported function.

Note These wrapper file differences only affect users who build libraries; they do not affect users who build executables.

Deprecated Compiler Options

MATLAB Compiler 4 is easier to use than previous versions, yet is a more powerful tool. In simplifying the Compiler, some of the options that were available in earlier releases became unnecessary and are not available in this release. The following options are no longer supported and will produce errors.

Table 1-1: Deprecated Compiler Options

Option	Description
`A`	Code annotation
`B pcode`	Generate P-code
`F`	Format parameters
`h`	Helper functions
`i`	Include specified M-files

Table 1-1: Deprecated Compiler Options (Continued)

Option	Description
`l`	Line/file numbers (This option has changed and now means “library.”)
`L`	Target language
`O`	Optimized code
`p`	Generate C++ code (This option has changed and now means “add directory to compilation path in an order-sensitive context.”)
`S`	Macro to generate Simulink S-function
`t`	Translate M-code to C/C++ code
`u`	Specifies number of inputs for Simulink S-function
`x`	Macro to generate MEX-function
`y`	Specifies number of outputs for Simulink S-function

Deprecated Wrapper Options

The following wrapper options and their associated bundle files are deprecated and are replaced by the new ones.

Wrapper Option/Bundle File	Replaced By

`B csglcom`
`B csglexcel`
`B csglsharedlib`
`B cppsglcom`
`B cppsglexcel`
	`B ccom`
	`B cexcel`
	`B csharedlib`
	`B cppcom`
	`B cppexcel`

`W comhg`
`W excelhg`
`W libhg`
`W mainhg`
	`W com`
	`W excel`
	`W lib`
	`W main`

Note You no longer need to use `-B sgl` and `-B sglcpp` to access Handle Graphics® functions. All compiled applications have access to graphics by default.

New Compiler Options

The following options are new in Compiler 4.

Option	Description
`a filename`	Add `filename` to archive; specifies files to be directly added to the CTF archive.
`l`	Macro that generates a function library. (The meaning of this option has changed since Release 13.)
`N`	Clears the path of all but a minimal, required set of directories.

Option	Description
`p <directory>`	Add `directory` to compilation path in an order-sensitive context; requires `-N` option.
`R -nojvm` `R -nojit`	Run-time; overrides MCR options; same as MATLAB startup options of the same name; only used with executable target.

Add to Archive. Use the `-a` option to specify files that should be directly added to the Component Technology File (CTF) archive. The Compiler looks for these files on the MATLAB path, so specifying the full pathname is optional. These files are not passed to `mbuild`, so you can include files such as data files. For example,

```
mcc -m foo.m -a data.mat -a /docs/help.txt
```

Multiple `-a` options are permitted, but each instance must be followed by the name of a file (specified by full or partial path) to add to the CTF archive. You can place the `-a` option anywhere in the `mcc` command line.

Files that are not MATLAB files, such as `.c`, `.cpp`, or `.obj`, that appear on the `mcc` command line are passed directly to `mbuild`.

Generate Library. Use the `-l` macro to generate a function library. The `-l` option is equivalent to

```
-W lib -T link:lib
```

It generates library wrapper functions for each M-file on the command line and calls your C compiler to build a shared library, which exports these functions. The library name is the component name, which is derived from the name of the first M-file on the command line.

Clear Path of All But Minimal Directories. Use the `-N` option to clear the path of all directories except the following necessary ones (this list is subject to change over time):

- `<matlabroot>/toolbox/matlab`
- `<matlabroot>/toolbox/local`
- `<matlabroot>/toolbox/compiler`

This option also retains all subdirectories of the above list that appear on the MATLAB path at compile time. Including `-N` on the `mcc` command line also allows you to replace directories from the original path, while retaining the relative ordering of the included directories. All subdirectories of the included directories that appear on the original path are also included.

Note This book uses `<matlabroot>` to represent the MATLAB root directory.

Add Directory to Compilation Path. Use the `-P` option to add a directory to the compilation path in an order-sensitive context. The syntax is

```
p <directory>
```

where `<directory>` is the directory to be included. If `<directory>` is not passed as an absolute path, it is assumed to be under the current working directory. The rules for how these directories are included are as follows:

- If a directory is included with `-p` that is on the original MATLAB path, the directory and all its subdirectories that appear on the original path are added to the compilation path in an order-sensitive context.
- If a directory is included with `-p` that is not on the original MATLAB path, that directory is not included in the compilation.
- If a path is added with the `-I` option while this feature is active (`-N` has been passed) and it is already on the MATLAB path, it is added in the order-sensitive context as if it were included with `-p`. Otherwise, the path is added to the head of the path, as it normally would be with `-I`.

Note The `-p` option requires the `-N` option on the `mcc` command line.

Run-Time. Use the `-R` option to override the MCR run-time options.

Note The `-R` option is *only* available when building stand-alone *applications*. It is not available when building components or libraries.

You can override any of these run-time options. These options are on by default.

MCR Option	Description
`-nojvm`	Do not use the Java Virtual Machine (JVM).
`-nojit`	Do not use the JIT Accelerator (binary code generation used to accelerate M-file execution).

These are valid uses of the `-R` option.

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

This example is invalid.

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

Uses of the MATLAB Compiler

This section introduces the various targets that the MATLAB Compiler can generate. You can find complete information about these targets and how to generate them in the corresponding sections throughout this book.

- Stand-Alone Applications
- Libraries
- COM Object
- Excel Add-in

Wrapper Files

The MATLAB Compiler (`mcc`) generates redistributable, stand-alone applications or software components from M-files. The specific final target can be any of the supported component types including stand-alone executable, library, or component. The Compiler generates the appropriate *wrapper* file based on the desired target. A wrapper file contains the required interface between the compiled application and the supported executable type.

Wrapper files differ depending on the execution environment. To provide the required interface, the wrapper

- Performs wrapper-specific initialization and termination.
- Defines data arrays containing path information, encryption keys, and other information needed by the MATLAB Component Runtime (MCR).
- Provides the necessary code to forward calls from the interface functions to the MATLAB functions in the MCR.

For example, the wrapper for a stand-alone executable contains the `main` function. The wrapper for a library contains the entry points for each public M-file function. You can find additional details specific to each wrapper type in each target's chapter throughout this book.

The Component Technology File (CTF) produced by the MATLAB Compiler is independent of the final target type — executable or library, but the CTF archive is platform specific. The wrapper file provides the necessary interface to the target type.

Stand-Alone Applications

The MATLAB Compiler, when invoked with the `-m` macro option, takes the input M-files and produces the required wrapper file suitable for a stand-alone application. Then, your C or C++ compiler compiles this code and links against the MCR, which is a stand-alone set of shared libraries that enable the execution of M-files. For example, to generate a stand-alone executable from the file `example.m`, use

```
mcc -m example
```

For overview information on creating stand-alone applications and deploying them, see "Quick Start" on page 1-16. For detailed information on stand-alone applications, see Chapter 5, "Stand-Alone Applications."

Libraries

You can use the `-l` option to create a C shared library from a set of M-files. For example

```
mcc -l file1.m file2.m file3.m
```

The `-l` option is a bundle option that expands into

```
-W lib -T link:lib
```

The `-W lib` option tells the MATLAB Compiler to generate a function wrapper for a shared library and call it `libfile1`. The `-T link:lib` option specifies the target output as a shared library. For overview information on creating shared libraries and deploying them, see "Quick Start" on page 1-16. For detailed information on creating libraries, see Chapter 6, "Libraries."

Builder Products

MATLAB Builder for COM

With the optional MATLAB Builder for COM, you can create COM components that can be used in any application that works with COM objects.

MATLAB Builder for COM uses the Compiler to create Component Object Model (COM) objects from MATLAB M-files. The collection of M-files is translated into a single COM class. MATLAB Builder for COM supports multiple classes per component. For more information, see "COM Object Target" on page 7-3.

MATLAB Builder for Excel

With the optional MATLAB Builder for Excel, you can automatically generate a Visual Basic Application file (`.bas`) and a plug-in DLL from your MATLAB model that can be imported into Excel as a stand-alone function.

MATLAB Builder for Excel compiles MATLAB M-files into a COM object that can be used as an Excel plug-in. The collection of M-files is translated into a single Excel plug-in. MATLAB Builder for Excel supports one class per component. For more information, see “Excel Plug-In Target” on page 7-8.

Note MATLAB Builder for COM and MATLAB Builder for Excel are available only on Windows.

Quick Start

Note From time to time, additional Compiler examples may be added to the File Exchange section of MATLAB Central on the MathWorks Web site. You can check for these examples at the File Exchange.

This section provides the basic steps of how to use the MATLAB Compiler to create components (applications and libraries) from MATLAB M-files. These components can then be deployed to machines that do not have MATLAB installed on them.

This quick start is meant for users who only want to know the steps to follow to create components. If your needs are more advanced or you want background information on the process, you will need to browse through the book and locate what you need.

Note Before you can use the Compiler, you must have it installed and configured properly on your system.

Compiling a Stand-Alone Application

To create a stand-alone application from the M-file `mymfunction`, use the command

```
mcc  m mymfunction.m
```

This creates a stand-alone executable named `mymfunction.exe` on Windows and `mymfunction` on Linux.

Compiling a Shared Library

To create a shared library from the M-file `mymfunction`, use the command

```
mcc  l mymfunction.m
```

This creates a shared library named `mymfunction.dll` on Windows and `mymfunction.so` on Linux.

Stand-Alone Application that Calls a Library

To create a stand-alone (driver) application from a wrapper file named `mywrappercode` that calls the above library `mymfunction`, use the command

```
mbuild mywrappercode.c mymfunction.lib          (On Windows)
mbuild mywrappercode.c mymfunction.so           (On Linux)
```

Note You must call `mclInitializeApplication` once at the beginning of your driver application. You must make this call before calling any other MathWorks functions. See “Calling a Shared Library” on page 6-9 for more information.

Testing Components on Development Machine

To test either component on your development machine, make sure you have your path set properly.

Windows. Add the following directory to your system `PATH` environment variable.

```
<matlabroot>\bin\win32
```

Linux. Add the following directories to your dynamic library path.

Note For readability, the following command appears on separate lines, but you must enter it all on one line.

```
setenv LD_LIBRARY_PATH
   <matlabroot>/bin/glnx86:
   <matlabroot>/sys/os/glnx86:
   <matlabroot>/sys/java/jre/glnx86/jre1.4.2/lib/i386/client:
   <matlabroot>/sys/java/jre/glnx86/jre1.4.2/lib/i386:
   <matlabroot>/sys/opengl/lib/glnx86:${LD_LIBRARY_PATH}

setenv XAPPLRESDIR <matlabroot>/X11/app-defaults
```

You can then run the compiled applications on your development machine to test them.

Deploying Components to Other Machines

To deploy your component to a target machine that does not have the same version of MATLAB installed as your development machine (or any version of MATLAB for that matter), you need to package the components and configure the target machines as follows.

Windows

- Your component, i.e., the stand-alone executable or shared library
- The CTF archive that is created for your component (`<component_name>.ctf`)
- `MCRInstaller.exe` (This file is located in the `<matlabroot>\toolbox\compiler\deploy\win32` directory.)

On the target machine, do the following:

1. Install the MCR by running the MCR Installer in a directory. For example, run `MCRInstaller.exe` in `C:\MCR`.
2. Copy the component and CTF archive to your application root directory, for example, `C:\approot`.
3. Add the following directory to your system path:

   ```
   <mcr_root>\runtime\win32
   ```

4. Test the component.

Linux

- Your component, i.e., the stand-alone executable or shared library
- The CTF archive that is created for your component (`<component_name>.ctf`)
- `MCRInstaller.zip`

To create `MCRInstaller.zip`, execute the following command at the MATLAB command prompt.

```
buildmcr
```

This command puts `MCRInstaller.zip` in the directory `<matlabroot>/toolbox/compiler/deploy`. If you do not have write access to this directory, you can specify the destination directory as an input to `buildmcr`.

On the target machine, do the following:

1. Install the MCR by unzipping `MCRInstaller.zip` in a directory, for example, `/home/<user>/MCR`. You may choose any directory except `<matlabroot>` or any directory below `<matlabroot>`.
2. Copy the component and CTF archive to your application root directory, for example, `/home/<user>/approot`.
3. Update your dynamic library path.

Note For readability, the following command appears on separate lines, but you must enter it all on one line.

```
setenv LD_LIBRARY_PATH
   <mcr_root>/runtime/glnx86:
   <mcr_root>/sys/os/glnx86:
   <mcr_root>/sys/java/jre/glnx86/jre1.4.2/lib/i386/client:
   <mcr_root>/sys/java/jre/glnx86/jre1.4.2/lib/i386:
   <mcr_root>/sys/opengl/lib/glnx86:${LD_LIBRARY_PATH}

setenv XAPPLRESDIR <matlabroot>/X11/app-defaults
```

4. Test the component.

Limitations and Restrictions

Compiling MATLAB and Toolboxes

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

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

The code generated by the MATLAB Compiler is not suitable for embedded applications.

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

MATLAB Code

MATLAB Compiler 4.0 supports much of the functionality of MATLAB. However, there are some limitations and restrictions that you should be aware of. This version of the Compiler cannot create interfaces for script M-files (See “Converting Script M-Files to Function M-Files” on page 4-17 for further details.)

Incorporating Updated M-Files into an Application

From time to time, MathWorks Technical Support distributes new versions of M-files to correct bugs via the Web. To incorporate these changes into your deployed applications, you must first apply the patch and then rerun `buildmcr` to generate an up-to-date version of the `MCRInstaller`. To deploy the bug fixes to your customers, you must ship this new `MCRInstaller` with your new applications and make the installer available to current customers so they may update their installation.

Stand-Alone Applications

The restrictions noted in the previous section also apply to stand-alone applications. In addition, there is a set of functions that is not supported in stand-alone mode. These functions fall into the these categories:

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

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

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

Table 1-2: Unsupported Functions

`add_block`	`add_line`	`applescript`	`close_system`
`dbclear`	`dbcont`	`dbdown`	`dbquit`
`dbstack`	`dbstatus`	`dbstep`	`dbstop`
`dbtype`	`dbup`	`delete_block`	`delete_line`
`echo`	`edit`	`fields`	`get_param`
`help`	`home`	`inmem`	`keyboard`
`linmod`	`mislocked`	`mlock`	`more`
`munlock`	`new_system`	`open_system`	`pack`
`rehash`	`set_param`	`sim`	`simget`
`simset`	`sldebug`	`type`	

Fixing Callback Problems: Missing Functions

When the Compiler creates a stand-alone application, it compiles the M-file(s) you specify on the command line and, in addition, it compiles any other M-files that your M-file(s) calls. The Compiler uses a dependency analysis, which determines all the functions on which the supplied M-files, MEX-files, and P-files depend. The dependency analysis may not locate a function if the only place the function is called in your M-file is a call to the function either

- In a callback string, or
- In a string passed as an argument to the `feval` function or an ODE solver.

The Compiler does not look in these text strings for the names of functions to compile.

Symptom

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

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

Workaround

To eliminate this error, create a list of all the functions that are specified only in callback strings and pass these functions using separate `%#function` pragma statements. (See “Finding Missing Functions in an M-File” on page 1-23 for suggestions about finding functions in callback strings.) This overrides the Compiler's dependency analysis and instructs it to explicitly include the functions listed in the `%#function` pragmas.

For example, the call to the `change_colormap` function in the sample application, `my_test`, illustrates this problem. To make sure the Compiler processes the `change_colormap` M-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');
```

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

Finding Missing Functions in an M-File

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

To find text strings used as callback strings, search for the characters "Callback" or "fcn" in your M-file. This will find all the `Callback` properties defined by Handle Graphics objects, such as `uicontrol` and `uimenu`. In addition, this will find the properties of figures and axes that end in `Fcn`, such as `CloseRequestFcn`, that also support callbacks.

Suppressing Warnings on Linux

Several warnings may appear when you run a stand-alone application on Linux. This section describes how to suppress these warnings.

To suppress the `app-defaults` warnings, set `XAPPLRESDIR` to point to `<mcr>/X11/app-defaults`.

To suppress the `libjvm.so` warning, place the following on the `LD_LIBRARY_PATH`.

```
<mcr>/sys/java/jre/glnx86/jre1.4.1/lib/i386/i386
<mcr>/sys/java/jre/glnx86/jre1.4.1/lib/i386/client
```

Alternately, you can use the MATLAB Compiler option `-R -nojvm` to set your application's `nojvm` run-time option, if the application is capable of running without Java.

MATLAB Compiler Licensing

Deployed Applications

Before you deploy applications or components to your users, you should be aware of the license conditions. Consult the Deployment Addendum in The MathWorks License Agreement at `www.mathworks.com/license` for terms and conditions of deployment.

Using MATLAB Compiler Licenses for Development

You can run the MATLAB Compiler from the MATLAB command prompt (MATLAB mode) or the DOS/UNIX prompt (stand-alone mode).

Running the Compiler in MATLAB Mode

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

Running the Compiler in Stand-Alone Mode

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

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

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

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

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

This message is given when no licenses are available. As long as licenses are available, the user gets the license and no message is displayed. The best way

to guarantee that all MATLAB Compiler users have constant access to the Compiler is to have an adequate supply of licenses for your users.

2

Installation and Configuration

This chapter describes the system requirements for the MATLAB Compiler. It also contains installation and configuration information for both Microsoft Windows and Linux.

When you install your ANSI C or C++ compiler, you may be required to provide specific configuration details regarding your system. This chapter contains information for each platform that can help you during this phase of the installation process.

System Requirements

To install the MATLAB Compiler 4, you must have MATLAB 7 (Release 14) installed on your system. The MATLAB Compiler imposes no operating system or memory requirements beyond those that are necessary to run MATLAB. The MATLAB Compiler consumes a small amount of disk space.

The MATLAB Compiler requires that a supported ANSI C or C++ compiler be installed on your system. Certain output targets require particular compilers.

In general, the MATLAB Compiler supports the current release of a third-party compiler and its previous release. Since new versions of compilers are released on a regular basis, it is important to check our Web site for the latest supported compilers.

Supported Third-Party Compilers

For an up-to-date list of all the compilers supported by MATLAB and the MATLAB Compiler, see the MathWorks Technical Support Department's Technical Notes at

```
http://www.mathworks.com/support/tech-notes/1600/1601.shtml
```

Supported ANSI C and C++ Windows Compilers

Use one of the following 32-bit C/C++ compilers that create 32-bit Windows dynamically linked libraries (DLLs) or Windows NT applications:

- Lcc C version 2.4 (included with MATLAB). This is a C only compiler; it does *not* work with C++.
- Borland C++ versions 5.3, 5.4, 5.5, 5.6, and free 5.5. (You may see references to these compilers as Borland C++Builder versions 3.0, 4.0, 5.0, and 6.0.) For more information on the free Borland compiler and its associated command line tools, see `http://community.borland.com`.
- Microsoft Visual C/C++ (MSVC) versions 6.0, 7.0, and 7.1.

Note The only compilers that support the building of COM objects are Borland C++Builder (versions 3.0, 4.0, 5.0, and 6.0) and Microsoft Visual C/C++ (versions 6.0, 7.0, and 7.1). The Borland C++Builder products require you to have the MIDL Compiler provided by Microsoft to create COM objects.

Supported ANSI C and C++ Linux Compilers

The MATLAB Compiler supports

- The GNU C compiler, `gcc`
- The system's native ANSI C compiler
- The GNU C++ compiler, `g++`

Installation

The MATLAB Compiler requires a supported ANSI C or C++ compiler installed on your system. This section describes the installation of the MATLAB Compiler and an ANSI C or C++ compiler.

Installing the MATLAB Compiler

Windows

To install the MATLAB Compiler on Windows, follow the instructions in the Installation Guide for Windows. If you have a license to install the MATLAB Compiler, it will appear as one of the installation choices that you can select as you proceed through the installation process.

If the Compiler does not appear in your list of choices, contact The MathWorks to obtain an updated License File (`license.dat`) for multiuser network installations, or an updated Personal License Password (PLP) for single-user, standard installations.

You can contact the MathWorks:

- Via the Web at `www.mathworks.com`. On the MathWorks home page, click on Support, then click on the Access Login, and follow the instructions.
- Via e-mail at `service@mathworks.com`

Linux

To install the MATLAB Compiler on Linux workstations, follow the instructions in the Installation Guide for UNIX. If you have a license to install the MATLAB Compiler, it appears as one of the installation choices that you can select as you proceed through the installation process. If the MATLAB Compiler does not appear as one of the installation choices, contact The MathWorks to get an updated license file (`license.dat`).

Installing an ANSI C or C++ Compiler

To install your ANSI C or C++ compiler, follow the vendor's instructions that accompany your C or C++ compiler. Be sure to test the C or C++ compiler to make sure it is installed and configured properly. Typically, the compiler vendor provides some test procedures.

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

When you install your C or C++ compiler, you might encounter configuration questions that require you to provide particular details. These tables provide information on some of the more common issues.

Windows

Issue	Comment
Installation options	We recommend that you do a full installation of your compiler. If you do a partial installation, you may omit a component that the MATLAB Compiler relies on.
Installing debugger files	For the purposes of the MATLAB Compiler, it is not necessary to install debugger (DBG) files. However, you may need them for other purposes.
Microsoft Foundation Classes (MFC)	This is not required.
16-bit DLL/executables	This is not required.
ActiveX	This is not required.
Running from the command line	Make sure you select all relevant options for running your compiler from the command line.

Windows (Continued)

Issue	Comment
Updating the registry	If your installer gives you the option of updating the registry, you should do it.
Installing Microsoft Visual C/C++ Version 6.0	If you need to change the location where this compiler is installed, you must change the location of the `Common` directory. Do not change the location of the `VC98` directory from its default setting.

Linux

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

Configuration

This section describes how to configure a C or C++ compiler to work with the MATLAB Compiler. There is a MATLAB utility called `mbuild` that simplifies the process of setting up a C or C++ compiler. Typically, you only need to use the `mbuild` utility's `setup` option to initially specify which third-party compiler you want to use. For more information on the `mbuild` utility, see the `mbuild` reference page.

Note In many cases, especially if you have the latest release of a third-party C/C++ compiler installed in its default location, you do not need to run `mbuild -setup`. However, there is no harm in doing so.

Introducing the mbuild Utility

The MathWorks utility, `mbuild`, lets you customize the configuration and build process. The `mbuild` script provides an easy way for you to specify an options file that lets you

- Set your compiler and linker settings
- Change compilers or compiler settings
- Build your application

The MATLAB Compiler (`mcc`) automatically invokes `mbuild` under certain conditions. In particular, `mcc -m` or `mcc -l` invokes `mbuild` to perform compilation and linking.

Configuring an ANSI C or C++ Compiler

Compiler Options Files

Options files contain flags and settings that control the operation of your installed C and C++ compiler. Options files are compiler-specific, i.e., there is a unique options file for each supported C/C++ compiler, which The MathWorks provides.

When you select a compiler to use with the MATLAB Compiler, the corresponding options file is activated on your system. To select a default compiler, use

```
mbuild -setup
```

Additional information on the options files is provided in this chapter for those users who may need to modify them to suit their own needs. Many users never have to be concerned with the inner workings of the options files and only need the `setup` option to initially designate a C or C++ compiler. If you need more information on options files, see "Options Files" on page 2-12.

Windows. Executing the command on Windows gives

```
mbuild -setup
Please choose your compiler for building standalone MATLAB
applications:

Would you like mbuild to locate installed compilers [y]/n? n

Select a compiler:
[1] Borland C++Builder version 6.0
[2] Borland C++Builder version 5.0
[3] Borland C++Builder version 4.0
[4] Borland C++Builder version 3.0
[5] Borland C/C++ version 5.02
[6] Borland C/C++ version 5.0
[7] Borland C/C++ (free command line tools) version 5.5
[8] Lcc C version 2.4
[9] Microsoft Visual C/C++ version 7.1
[10] Microsoft Visual C/C++ version 7.0
[11] Microsoft Visual C/C++ version 6.0

[0] None

Compiler: 11

Your machine has a Microsoft Visual C/C++ compiler located at
D:\Applications\Microsoft Visual Studio. Do you want to use this
compiler [y]/n? y
```

```
Please verify your choices:

Compiler: Microsoft Visual C/C++ 6.0
Location: D:\Applications\Microsoft Visual Studio

Are these correct?([y]/n): y

Try to update options file:
C:\WINNT\Profiles\username\Application
Data\MathWorks\MATLAB\R14\compopts.bat
From template:
\\sys\MATLAB\BIN\WIN32\mbuildopts\msvc60compp.bat

Done . . .
 .
 .
Updated ...
```

The preconfigured options files that are included with MATLAB for Windows are shown below.

Options File	Compiler
`lcccompp.bat`	Lcc C, Version 2.4 (included with MATLAB)

`msvc60compp.bat`
`msvc70compp.bat`
`msvc71compp.bat`
	Microsoft Visual C/C++, Version 6.0
	Microsoft Visual C/C++, Version 7.0
	Microsoft Visual C/C++, Version 7.1

`bcc53compp.bat`
`bcc54compp.bat`
`bcc55compp.bat`
`bcc56compp.bat`
	Borland C++ Builder 3
	Borland C++ Builder 4
	Borland C++ Builder 5
	Borland C++ Builder 6

Linux. Executing the command on Linux gives

```
mbuild -setup

Using the 'mbuild -setup' command selects an options file that is
placed in ~/.matlab/R14 and used by default for 'mbuild'. An
options file in the current working directory or specified on the
command line overrides the default options file in ~/.matlab/R14.

Options files control which compiler to use, the compiler and link
command options, and the runtime libraries to link against.

To override the default options file, use the 'mbuild -f' command
(see 'mbuild -help' for more information).

The options files available for mbuild are:

1: <matlabroot>/bin/mbuildopts.sh :
Build and link with the MATLAB Compiler and default C/C++ compiler

<matlabroot>/bin/mbuildopts.sh is being copied to
/home/user/.matlab/R14/mbuildopts.sh
```

The preconfigured options file that is included with MATLAB for Linux is `mbuildopts.sh`, the system native ANSI compiler.

Special Compiler Notes

This section describes known limitations of particular compilers.

Known Windows Compiler Limitations

There are several known restrictions regarding the use of supported compilers:

- The Lcc C compiler does not support C++.
- The only compilers that support the building of COM objects are Borland C++Builder (versions 3.0, 4.0, 5.0, and 6.0) and Microsoft Visual C/C++ (versions 6.0, 7.0, and 7.1).

Note The Borland C++Builder products require you to have the MIDL Compiler provided by Microsoft to create COM objects.

- There is a limitation with the Borland C++ Compiler. In your M-code, if you use a constant number that includes a leading zero and contains the digit "8" or "9" before the decimal point, the Borland compiler will display the error message

  ```
  Error <file>.c <line>: Illegal octal digit in function
  <functionname>
  ```

  For example, the Borland compiler considers `009.0` an illegal octal integer as opposed to a legal floating-point constant, which is how it is defined in the ANSI C standard.

  As an aside, if all the digits are in the legal range for octal numbers (0-7), then the compiler will incorrectly treat the number as a floating-point value. So, if you have code such as

  ```
  x = [007 06 10];
  ```

  and want to use the Borland compiler, you should edit the M-code to remove the leading zeros and write it as

  ```
  x = [7 6 10];
  ```

Options Files

This information is provided for users who need to know more about how options files work.

Locating the Options File

Windows

To locate your options file on Windows, the `mbuild` script searches the following locations:

- Current directory
- The `user profile` directory (see the following section for more information about this directory)

`mbuild` uses the first occurrence of the options file it finds. If no options file is found, `mbuild` searches your machine for a supported C compiler and uses the factory default options file for that compiler. If multiple compilers are found, you are prompted to select one.

The User Profile Directory Under Windows. The Windows `user profile` directory is a directory that contains user-specific information such as desktop appearance, recently used files, and **Start** menu items. The `mbuild` utility stores its options files, `compopts.bat`, which is created during the `-setup` process, in a subdirectory of your `user profile` directory, named `Application Data\MathWorks\MATLAB\R14`. Under Windows with user profiles enabled, your `user profile` directory is `%windir%\Profiles\username`. Under Windows with user profiles disabled, your `user profile` directory is `%windir%`. You can determine whether or not user profiles are enabled by using the **Passwords** control panel.

Linux

To locate your options file on Linux, the `mbuild` script searches the following locations:

- Current directory
- `$HOME/.matlab/R14`
- `<matlabroot>/bin`

`mbuild` uses the first occurrence of the options file it finds. If no options file is found, `mbuild` displays an error message.

Changing the Options File

Although it is common to use one options file for all of your Compiler-related work, you can change your options file at anytime. The `setup` option resets your default compiler so that the new compiler is used every time. Use

```
mbuild -setup
```

to reset your C or C++ compiler for future sessions.

Windows

Modifying the Options File. You can use of the `setup` option to change your options file settings on Windows. The `setup` option copies the appropriate options file to your `user profile` directory.

To modify your options file on Windows:

1. Use `mbuild -setup` to make a copy of the appropriate options file in your local area.

2. Edit your copy of the options file in your `user profile` directory to correspond to your specific needs and save the modified file.

After completing this process, the `mbuild` script will use the new options file everytime with your modified settings.

Linux

The `setup` option creates a user-specific, `matlab` directory in your individual home directory and copies the appropriate options file to the directory. (If the directory already exists, a new one is not created.) This `matlab` directory is used for your individual options files only; each user can have his or her own default options files (other MATLAB products may place options files in this directory). Do not confuse these user-specific `matlab` directories with the system `matlab` directory, where MATLAB is installed.

Modifying the Options File. You can use the `setup` option to change your options file settings on Linux. For example, if you want to make a change to the current

linker settings, or you want to disable a particular set of warnings, you should use the `setup` option.

To modify your options file on Linux:

1. Use `mbuild -setup` to make a copy of the appropriate options file in your local area.
2. Edit your copy of the options file to correspond to your specific needs and save the modified file.

This sets your default compiler's options file to your specific version.

3

Compilation Process

This chapter provides an overview of how the MATLAB Compiler works. In addition, it lists the various sets of input and output files used by the Compiler.

Overview of the MATLAB Compiler Technology

MATLAB Component Runtime

MATLAB Compiler 4 uses the MATLAB Component Runtime (MCR), which is a stand-alone set of shared libraries that enables the execution of M-files. The MCR provides complete support for all features of the MATLAB language.

Component Technology File

Compiler 4 also uses a Component Technology File (CTF) archive to house the deployable package. All M-files are encrypted in the CTF archive using the Advanced Encryption Standard (AES) cryptosystem where symmetric keys are protected by 1024-bit RSA keys.

Each application or shared library produced by the MATLAB Compiler has an associated CTF archive. The archive contains all the MATLAB based executable content (M-files, MEX-files) associated with the component.

Additional Details

Multiple CTF archives, such as COM or Excel components, can coexist in the same user application, but you cannot mix and match the M-files they contain. You cannot combine encrypted and compressed M-files from multiple CTF archives into another CTF archive and distribute them.

All the M-files from a given CTF archive are locked together with a unique cryptographic key. M-files with different keys will not execute if placed in the same CTF archive. If you want to generate another application with a different mix of M-files, you must recompile these M-files into a new CTF archive.

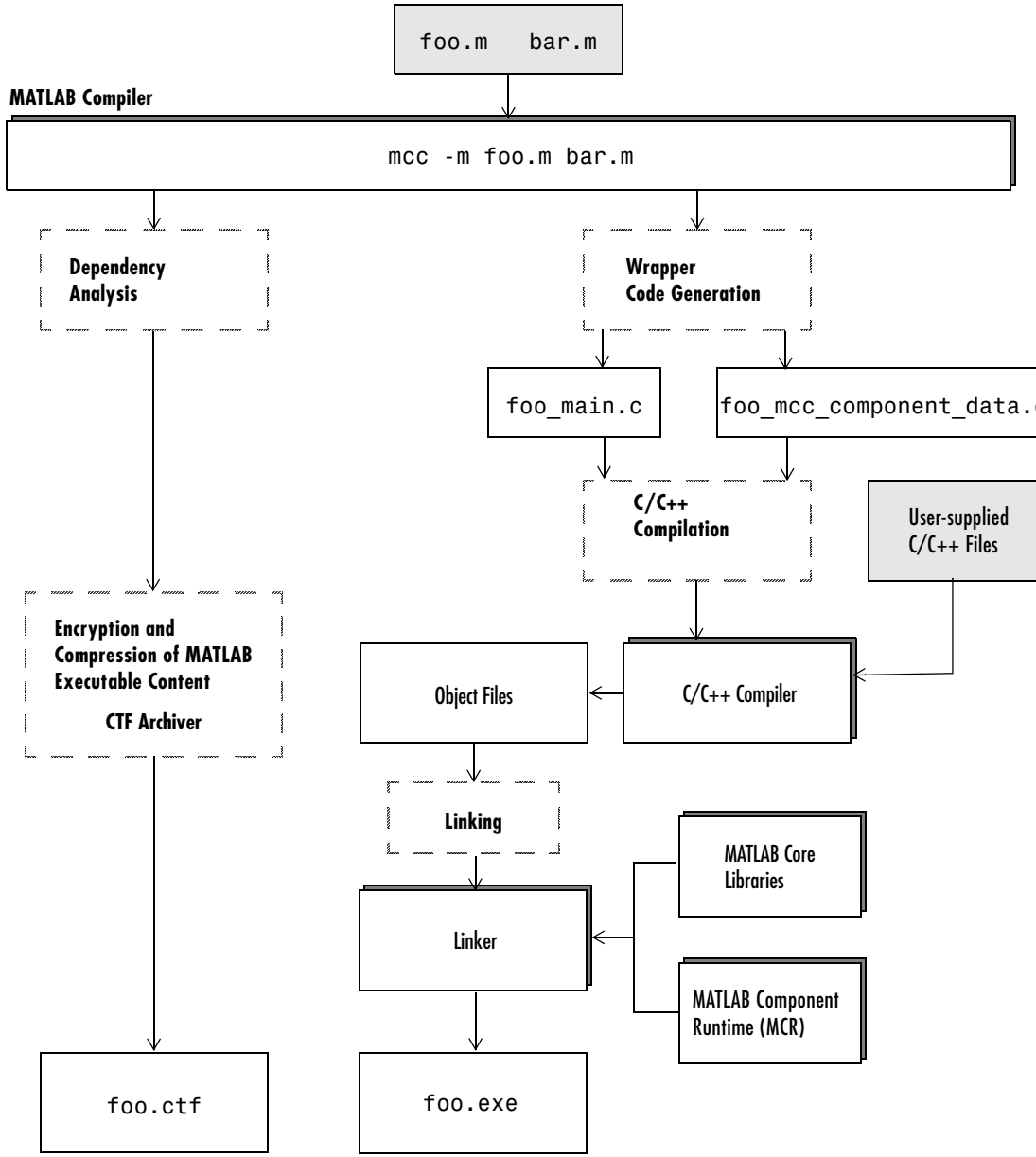
Build Process

The process of creating software components with the MATLAB Compiler are completely automatic. For example, to create a stand-alone MATLAB application, you supply the list of M-files that comprise the application. The Compiler then performs the following operations:

- Dependency analysis
- Code generation
- Archive creation

- Compilation
- Linking

This figure illustrates how the Compiler takes user code and generates a stand-alone executable.


foo.m bar.m
MATLAB Compiler
mcc -m foo.m bar.m
Dependency Analysis
Wrapper Code Generation
foo_main.c
foo_mcc_component_data.c
C/C++ Compilation
User-supplied C/C++ Files
Encryption and Compression of MATLAB Executable Content
CTF Archiver
Object Files
C/C++ Compiler
Linking
MATLAB Core Libraries
Linker
MATLAB Component Runtime (MCR)
foo.ctf
foo.exe

Dependency Analysis

The first step determines all the functions on which the supplied M-files, MEX-files, and P-files depend. This list includes all the M-files called by the given files as well as files that they call, and so on. Also included are all built-in functions and MATLAB objects.

Wrapper Code Generation

This step generates all the source code needed to create the target component, including

- The C/C++ interface code to those M-functions supplied on the command line (`foo_main.c`). For libraries and components, this file includes all of the generated interface functions.
- A component data file that contains information needed to execute the M-code at run-time. This data includes path information and encryption keys needed to load the M-code stored in the component's CTF archive.

Archive Creation

The list of MATLAB executable files (M-files and MEX-files) created during dependency analysis is used to create a CTF archive that contains the files needed by the component to properly execute at run-time. The files are encrypted and compressed into a single file for deployment. Directory information is also included so that the content is properly installed on the target machine.

C/C++ Compilation

This step compiles the generated C/C++ files from wrapper code generation into object code. For targets that support the inclusion of user-supplied C/C++ code on the `mcc` command line, this code is also compiled at this stage.

Linking

The final step links the generated object files with the necessary MATLAB libraries to create the finished component.

The C/C++ compilation and linking steps use the `mbuild` utility that is included with the MATLAB Compiler.

Input and Output Files

This section describes the files created during the compilation process.

Stand-Alone Executable

In this example, the MATLAB Compiler takes the M-files `foo.m` and `bar.m` as input and generates a stand-alone executable called `foo`.

```
mcc -m foo.m bar.m
```

File	Description
`foo_main.c`	The main-wrapper C source file containing the program's `main` function. The `main` function takes the input arguments that are passed on the command line and passes them as strings to the `foo` function.
`foo_mcc_component_data.c`	C source file containing data needed by the MCR to run the application. This data includes path information, encryption keys, and other initialization information for the MCR.
`foo.ctf`	The CTF archive. This file contains a compressed and encrypted archive of the M-files that make up the application (`foo.m` and `bar.m`). It also contains other files called by the two main M-files as well as any other executable content and data files needed at run-time.
`foo`	The main executable file of the application. This file reads and executes the content stored in the CTF archive. On Windows, this file is `foo.exe`.

C Shared Library

In this example, the Compiler takes the M-files `foo.m` and `bar.m` as input and generates a C shared library called `libfoo`.

```
mcc -W lib:libfoo -T link:lib foo.m bar.m
```

File	Description
`libfoo.c`	The library wrapper C source file containing the exported functions of the library representing the C interface to the the two M-functions (`foo.m` and `bar.m`) as well as library initialization code.
`libfoo.h`	The library wrapper header file. This file is included by applications that call the exported functions of `libfoo`.
`libfoo_mcc_component_data.c`	C source file containing data needed by the MCR to initialize and use the library. This data includes path information, encryption keys, and other initialization for the MCR.
`libfoo.exports`	The exports file used by `mbuild` to link the library.
`libfoo.ctf`	The CTF archive. This file contains a compressed and encrypted archive of the M-files that make up the library (`foo.m` and `bar.m`). This file also contains other files called by the two main M-files as well as any other executable content and data files needed at run-time.
`libfoo`	The shared library binary file. On Windows, this file is `libfoo.dll`. (Note: UNIX extensions vary depending on platform. See the External Interfaces documentation for additional information.)

Deployment Process

After creating your component with the MATLAB Compiler, you can distribute, or deploy, it to others so that they can use it on their machines, independent of MATLAB.

The deployment process requires that you

1. Package the necessary components depending on the type of generated application.
2. Distribute them to your end user.
3. Have the end users install them on their systems. During this phase of the installation process, the end users run `MCRInstaller` *once* on their target machine, that is, the machine where they will run the application or library. On Windows, `MCRInstaller` is a self-extracting executable that installs the necessary components to run your application. On Linux, `MCRInstaller` is a ZIP file. See “Installing the MCR on a Deployment Machine” on page 3-12 for additional details on `MCRInstaller`.

There are specific examples of how to deploy each target in their corresponding chapters in this book.

Porting Generated Code to a Different Platform

Since binary formats are different on each platform, the various components generated by the MATLAB Compiler cannot be moved from platform to platform as is. You can distribute a MATLAB Compiler-generated application to any target machine that has the same operating system as the machine on which the application was compiled. For example, if you want to deploy an application to a Windows machine, you must use the Windows version of the MATLAB Compiler to build the application on a Windows machine.

To deploy an application to a machine whose operating system is different than the machine used to develop the application, requires recompiling. You must recompile the application on the desired targeted platform. For example, If you want to deploy the previous application that was developed on a Windows machine to a Linux machine, you must use the MATLAB Compiler on a Linux machine and completely rebuild the application. Consequently, you must have

a valid MATLAB Compiler license on both platforms in order to be able to do this.

Extracting a CTF Archive without Executing the Component

CTF archives contain executable content (M-files and MEX-files) that need to be extracted from the archive before they can be executed. The CTF archive automatically expands the first time you run the MATLAB Compiler-based component (a MATLAB Compiler-based stand-alone application or an application that calls a MATLAB Compiler-based shared library or COM component).

To expand an archive without running the application, you can use the `extractCTF` (`.exe` on Windows) stand-alone utility provided in the `<matlabroot>/toolbox/compiler/deploy/<ARCH>` directory, where `<ARCH>` is `win32` on Windows and `glnx86` on Linux. This utility takes the name of the CTF archive as input and expands the archive into the current working directory. For example, this command expands `hello.ctf` into the current working directory.

```
extractCTF hello.ctf
```

The archive expands into a directory called `hello_mcr`. In general, the name of the directory containing the expanded archive is `<componentname>_mcr`, where `componentname` is the name of the CTF archive without the extension.

Note To run `extractCTF` from any directory, you must add `<matlabroot>/toolbox/compiler/deploy/<ARCH>` to your `PATH` environment variable.

User Interaction with the Compilation Path

The MATLAB Compiler uses a dependency analysis function (`depfun`) to determine the list of necessary files to include in the CTF package. In some cases, this process includes an excessive number of files, for example, when MATLAB OOPS classes are included in the compilation and it cannot resolve overloaded methods at compile time. The dependency analysis is an iterative process that also processes `include/exclude` information on each pass.

Consequently, this process can lead to very large CTF archives resulting in long compilation times for relatively small applications.

The most effective way to reduce the number of files is to constrain the MATLAB path that `depfun` uses at compile time. The Compiler includes features that enable you to manipulate the path. Currently, there are three ways to interact with the compilation path:

- `addpath` and `rmpath` in MATLAB
- Passing `-I <directory>` on the `mcc` command line
- Passing `-N` and `-p` directories on the `mcc` command line (new feature)

addpath and rmpath in MATLAB

If you run the Compiler from the MATLAB prompt, you can use the `addpath` and `rmpath` commands to modify the MATLAB path before doing a compilation. There are two disadvantages:

- The path is modified for the current MATLAB session only.
- If the Compiler is run outside of MATLAB, this doesn't work unless a `savepath` is done in MATLAB.

Note The path is also modified for any interactive work you are doing in MATLAB as well.

Passing `-I<` directory> on the Command Line

You can use the `-I` option to add a directory to the head of the path used by the current compilation. This feature is useful when you are compiling files that are in directories currently not on the MATLAB path.

Passing -N and -p <directory> on the Command Line

There are now two new Compiler options that provide more detailed manipulation of the path. This new feature acts like a "filter" applied to the MATLAB path for a given compilation. The first new option is `-N`. Passing `-N` on the `mcc` command line effectively clears the path of all directories except the following core directories (this list is subject to change over time):

- `<matlabroot>/toolbox/matlab`

- <matlabroot>/toolbox/local
- <matlabroot>/toolbox/compiler

It also retains all subdirectories of the above list that appear on the MATLAB path at compile time. Including -N on the command line also allows you to replace directories from the original path, while retaining the relative ordering of the included directories. All subdirectories of the included directories that appear on the original path are also included.

Use the -p option to add a directory to the compilation path in an order-sensitive context, i.e., the same order in which they are found on your MATLAB path. The syntax is

```
p <directory>
```

where <directory> is the directory to be included. If <directory> is not an absolute path, it is assumed to be under the current working directory. The rules for how these directories are included are

- If a directory is included with -p that is on the original MATLAB path, the directory and all its subdirectories that appear on the original path are added to the compilation path in an order-sensitive context.
- If a directory is included with -p that is not on the original MATLAB path, that directory is not included in the compilation. (You can use -I to add it.)
- If a path is added with the -I option while this feature is active (-N has been passed) and it is already on the MATLAB path, it is added in the order-sensitive context as if it were included with -p. Otherwise, the directory is added to the head of the path, as it normally would be with -I.

Note The -p option requires the -N option on the mcc command line.

Working with the MCR

Installing the MCR on a Deployment Machine

Before end users can run MATLAB Compiler-generated components on their machines, they need to install the MCR, if it is not already present. You only need to install the MCR one time on a deployment machine.

End users on Windows can use the `MCRInstaller` utility (`MCRInstaller.exe`) to prepare the deployment machine. Linux users must execute the `MCRInstaller`, which is a ZIP file, and then manually set the path and environment variables as required. To prepare the deployment machine, you need to

- Install the MCR
- Set the path properly
- Set the necessary environment variables

Note If the Linux `MCRInstaller.zip` file is not present on your machine, you can generate it using the `buildmcr` function in MATLAB. For more information on using `buildmcr`, see "Deploying the Application" on page 5-5.

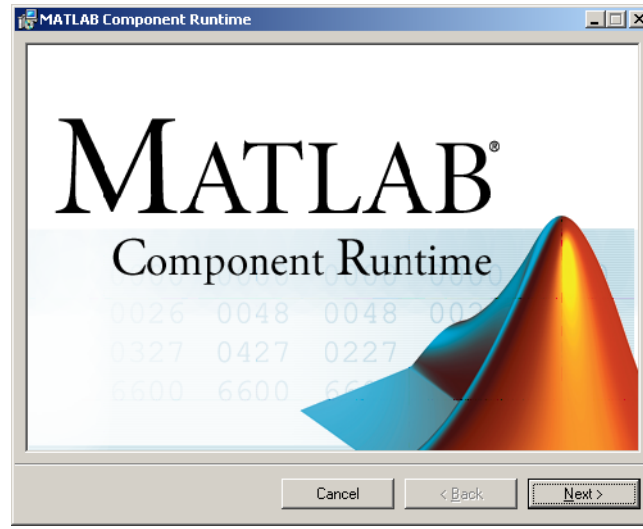
Windows

1. Locate the `MCRInstaller` utility in the `<matlabroot>\toolbox\compiler\deploy\win32` directory and copy it to a new directory on your machine. Run the utility to start the installation.

   ```
   MCRInstaller.exe
   ```

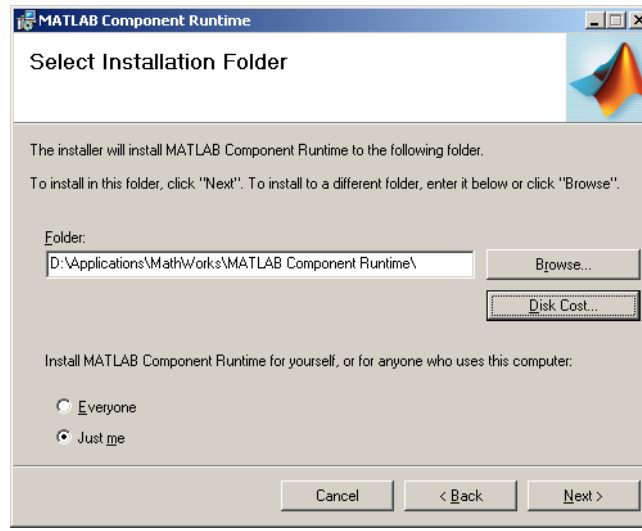
   The `MCRInstaller` opens a command window and begins preparation for the installation.

2 When the **MATLAB Component Runtime** startup screen appears, click **Next** to begin the installation.



3 The setup wizard starts. Click **Next** to continue.

4 The **Select Installation Folder** dialog lets you choose where you want to install the MCR. This dialog also lets you view available and required disk space on your system. You can also choose whether you want to install the MCR for just yourself or others. Select your options, and then click **Next** to continue.



5 Confirm your selections by clicking **Next**.

The installation begins. The process takes some time due to the quantity of files that are installed.

6 When the installation completes, click **Close** on the **Installation Completed** dialog to exit.

Note The **Install MATLAB Component Runtime for yourself, or for anyone who uses this computer** option is not implemented for this release. The current default is **Everyone**.

Linux

1 Locate the `MCRInstaller.zip` file and copy it to a new directory on your machine. This new directory will become the installation directory for your Compiler-generated components. To install the MCR, unzip `MCRInstaller.zip`.

2 Update your dynamic library path.

Note For readability, the following command appears on separate lines, but you must enter it all on one line.

```
setenv LD_LIBRARY_PATH
   <mcr_root>/runtime/glnx86:
   <mcr_root>/sys/os/glnx86:
   <mcr_root>/sys/java/jre/glnx86/jre1.4.2/lib/i386/client:
   <mcr_root>/sys/java/jre/glnx86/jre1.4.2/lib/i386:
   <mcr_root>/sys/opengl/lib/glnx86:${LD_LIBRARY_PATH}

setenv XAPPLRESDIR <matlabroot>/X11/app-defaults
```

4

Working with mcc

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

Command Overview

mcc is the MATLAB command that invokes the MATLAB Compiler. You can issue the mcc command either from the MATLAB command prompt (MATLAB mode) or the DOS or UNIX command line (stand-alone mode).

Compiler Options

You may specify one or more MATLAB Compiler option flags to mcc. Most option flags have a one-letter name. You can list options separately on the command line, for example,

```
mcc -m -g myfun
```

Macros are MathWorks supplied Compiler options that simplify the more common compilation tasks. Instead of manually grouping several options together to perform a particular type of compilation, you can use a simple macro option. You can always use individual options to customize the compilation process to satisfy your particular needs. For more information on macros, see “Using Macros to Simplify Compilation” on page 4-4.

Combining Options

You can group options that do not take arguments by preceding the list of option flags with a single dash (-), for example:

```
mcc -mg myfun
```

Options that take arguments cannot be combined unless you place the option with its arguments last in the list. For example, these formats are valid.

```
mcc -v -W main -T link:exe myfun   % Options listed separately
mcc -vW main -T link:exe myfun     % Options combined
```

This format is *not* valid.

```
mcc -Wv main -T link:exe myfun
```

In cases where you have more than one option that takes arguments, you can only include one of those options in a combined list and that option must be last. You can place multiple combined lists on the mcc command line.

If you include any C or C++ filenames on the mcc command line, the files are passed directly to mbuild, along with any Compiler-generated C or C++ files.

Conflicting Options on Command Line

If you use conflicting options, the Compiler resolves them from left to right, with the rightmost option taking precedence. For example, using the equivalencies in Table 4-1, Macro Options, on page 4-4,

```
mcc -m -W none test.m
```

is equivalent to

```
mcc -W main -T link:exe -W none test.m
```

In this example, there are two conflicting `-W` options. After working from left to right, the Compiler determines that the rightmost option takes precedence, namely, `-W none`, and the Compiler does not generate a wrapper.

Note Macros and regular options may both affect the same settings and may therefore override each other depending on their order in the command line.

Setting Up Default Options

If you have some command line options that you wish always to pass to `mcc`, you can do so by setting up an `mccstartup` file. Create a text file containing the desired command line options and name the file `mccstartup`. Place this file in one of two directories:

- The current working directory, or
- Your preferences directory (`$HOME/.matlab/R14` on UNIX, `<system root>\profiles\<user>\application data\mathworks\matlab\R14` on Windows)

`mcc` searches for the `mccstartup` file in these two directories in the order shown above. If it finds an `mccstartup` file, it reads it and processes the options within the file as if they had appeared on the `mcc` command line before any actual command line options. Both the `mccstartup` file and the `-B` option are processed the same way.

Using Macros to Simplify Compilation

The MATLAB 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 option, i.e., *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.

Table 4-1: Macro Options

Macro Option	Bundle File	Creates	Option Equivalence Function Wrapper | Output Stage | |
`-l`	`macro_option_l`	Library	`-W lib  -T link:lib`
`-m`	`macro_option_m`	Stand-alone C application	`-W main  -T link:exe`

Understanding a Macro Option

The `-m` option tells the Compiler to produce a stand-alone C 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.

Table 4-2: The -m Macro

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

Changing Macro Options

You can change the meaning of a macro option by editing the corresponding `macro_option` file bundle file in `<matlabroot>/toolbox/compiler/bundles`. For example, to change the `-m` macro, edit the file `macro_option_m` in the `bundles` directory.

Using Pathnames

If you specify a full pathname to an M-file on the `mcc` command line, the MATLAB Compiler

1. Breaks the full name into the corresponding pathname and filenames (`<path>` and `<file>`).
2. Replaces the full pathname in the argument list with "`-I <path> <file>`". 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 M-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 M-file the Compiler parses. The `-v` option prints the full pathname to the M-file during the dependency analysis phase.

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

Using Bundle Files

Bundle files provide a convenient way to group sets of MATLAB Compiler options and recall them as needed. The syntax of the bundle file option is

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

When used on the `mcc` command line, the bundle option `-B` replaces the entire string with the contents of the specified file. The file should contain only `mcc` command line options and corresponding arguments and/or other filenames. The file may contain other `-B` options.

A bundle file can include replacement parameters for Compiler options that accept names and version numbers. For example, there is a bundle file for C shared libraries, `csharedlib`, that consists of

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

To invoke the Compiler to produce a C shared library using this bundle, you could use

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

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

You can place options that you always set in an `mccstartup` file. For more information, see “Setting Up Default Options” on page 4-3.

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

This table shows the available bundle files.

Bundle File Name	Creates	Contents
`ccom`	COM Object	`-W com:<component_name>,<class_name>,<version>` `-T link:lib`
`cexcel`	Excel COM Object	`-W excel:<component_name>,<class_name>,<version>` `-T link:lib -b`
`cppcom`	COM Object (same as `ccom`)	`-B ccom:<component_name>,<class_name>,<version>`
`cppexcel`	Excel COM Object (same as `cexcel`)	`-B cexcel:<component_name>,<class_name>,` `<version>`
`cpplib`	C++ Library	`-B csharedlib:<shared_library_name>` `-T compile:lib`
`csharedlib`	C Shared Library	`-W lib:<shared_library_name> -T link:lib`

Bundle File Name	Creates	Contents
`macro_option_l`	N/A	`-W lib -T link:lib`
`macro_option_m`	N/A	`-W main -T link:exe`

Note To create COM components with the MATLAB Compiler, you must have the MATLAB Builder for COM product installed on your system. To create Microsoft Excel Builder components with the MATLAB Compiler, you must have the MATLAB Builder for Excel product installed on your system.

Using Wrapper Files

Wrapper files, which contain wrapper functions, create a link between the MATLAB Compiler-generated code and a supported executable type such as stand-alone executable (main) or library by providing the required interface that allows the code to operate in the desired execution environment.

To provide the required interface, the wrapper

- Performs wrapper-specific initialization and termination
- Provides the dispatching of function calls to the MCR

To specify the type of wrapper to generate, use the syntax

```
-W <type>
```

The following sections detail the available wrapper types.

Main File Wrapper

The `-W main` option generates wrappers that are suitable for building stand-alone applications. These POSIX-compliant main wrappers accept strings from the POSIX shell and return a status code. They pass these command line strings to the M-file function(s) as MATLAB strings. They are meant to translate “command-like” M-files into POSIX main applications.

POSIX Main Wrapper

Consider this M-file, `sample.m`.

```
function y = sample(varargin)
varargin{:}
y = 0;
```

You can compile `sample.m` into a POSIX main application. If you call `sample` from MATLAB, you get

```
sample hello world

ans =
hello

ans =
world
```

```
ans =
     0
```

If you compile `sample.m` and call it from the DOS shell, you get

```
C:\> sample hello world

ans =
hello

ans =
world

C:\>
```

The difference between the MATLAB and DOS/UNIX environments is the handling of the return value. In MATLAB, the return value is handled by printing its value; in the DOS/UNIX shell, the return value is handled as the return status code. When you compile a function into a POSIX main application, the first return value from the function is coerced to a scalar and is returned to the POSIX shell.

C Library Wrapper

The `-l` option, or its equivalent `-W lib:libname`, produces a C library wrapper file. This option produces a shared library from an arbitrary set of M-files. The generated header file contains a C function declaration for each of the compiled M-functions. The export list contains the set of symbols that are exported from a C shared library.

Note You must generate a library wrapper file when calling any Compiler-generated code from a larger application.

C++ Library Wrapper

The `-W cpplib:libname` option produces the C++ library wrapper file. This option allows the inclusion of an arbitrary set of M-files into a library. The generated header file contains all of the entry points for all of the compiled M-functions.

Note You must generate a library wrapper file when calling any Compiler-generated code from a larger application.

COM Component Wrapper

The COM component wrapper file allows you to create COM components from MATLAB M-files. The options that generate COM wrappers are

```
-W com:<component_name>[,<class_name>[,<major>.<minor>]]
-W excel:<component_name>[,<class_name>[,<major>.<minor>]]
```

The COM wrapper options create a superset of the files created when producing a C or C++ library wrapper.

Interfacing M-Code to C/C++ Code

The MATLAB Compiler supports calling arbitrary C/C++ functions from your M-code. You simply provide an M-function stub that determines how the code will behave in M, and then provide an implementation of the body of the function in C or C++.

C Example

Suppose you have a C function that reads data from a measurement device. In M-code, you want to simulate the device by providing a sine wave output. In production, you want to provide a function that returns the measurement obtained from the device. You have a C function called `measure_from_device()` that returns a `double`, which is the current measurement.

`collect.m` contains the M-code for the simulation of your application.

```
function collect

y = zeros(1, 100); %Preallocate the matrix
for i = 1:100
   y(i) = collect_one;
end

function y = collect_one

persistent t;
if (isempty(t))
   t = 0;
end
t = t + 0.05;
y = sin(t);
```

The next step is to replace the implementation of the `collect_one` function with a C implementation that provides the correct value from the device each time it is requested. This is accomplished by using the `%#external` pragma.

The `%#external` pragma informs the MATLAB Compiler that the function will be hand written and will not be generated from the M-code. This pragma affects only the single function in which it appears. Any M-function may contain this pragma (local, global, private, or method). When using this

pragma, the Compiler will generate an additional header file called `fcn_external.h`, where `fcn` is the name of the initial M-function containing the `%#external` pragma. This header file will contain the `extern` declaration of the function that you must provide. This function must conform to the same interface as the Compiler-generated code.

The Compiler will generate the interface for any functions that contain the `%#external` pragma into a separate file called `fcn_external.h`. The Compiler-generated C or C++ file will include this header file to get the declaration of the function being provided.

In this example, place the pragma in the `collect_one` local function.

```
function collect

y = zeros(1, 100); % preallocate the matrix
for i = 1:100
   y(i) = collect_one;
end

function y = collect_one

%#external
persistent t;
if (isempty(t))
   t = 0;
end
t = t + 0.05;
end
y = sin(t);
```

When this file is compiled, the Compiler creates the additional header file `collect_one_external.h`, which contains the interface between the Compiler-generated code and your code. In this example, it would contain

```
extern void collect_one(int nlhs, mxArray *plhs[],
                        int nrhs, mxArray *prhs[]);
```

We recommend that you include this header file when defining the function. This function could be implemented in this C file, `measure.c`, using the `measure_from_device()` function.

```
#include "collect_one_external.h"
```

```
#include <math.h>

extern double measure_from_device(void);

void collect_one(int nlhs, mxArray *plhs[],
                 int nrhs, mxArray *prhs[]);

{
   plhs[0] = mxCreateDoubleMatrix(1,1,mxREAL);
   *(mxGetPr(plhs[0])) = measure_from_device()
}

double measure_from_device(void)
{
   static double t = 0.0;
   t = t + 0.05;
   return sin(t);
}
```

In general, the Compiler will use the same interface for this function as it would generate. To generate the application, use

```
mcc -m collect.m measure.c
```

Using Pragmas

Using feval

In stand-alone C and C++ modes, the pragma

```
%#function <function_name-list>
```

informs the MATLAB Compiler that the specified function(s) should be included in the compilation, whether or not the Compiler's dependency analysis detects it. Without this pragma, the Compiler's dependency analysis will not be able to locate and compile all M-files used in your application.

You cannot use the `%#function` pragma to refer to functions that are not available in M-code.

Example - Using %#function

A good coding technique involves using `%#function` in your code wherever you use `feval` statements. This example shows how to use this technique to help the Compiler find the appropriate files during compile time, eliminating the need to include all the files on the command line.

```
function ret = mywindow(data,fitlerName)
%MYWINDOW Applies the window specified on the data
%

% Get the length of the data
N= length(data);

% List all the possible windows
%#function bartlett, barthannwin, blackman, blackmanharris, ...
bohmanwin, chebwin, flattopwin, gausswin, hamming, hann,...
kaiser, nuttallwin, parzenwin, rectwin, tukeywin, ...
triang window = feval(fitlerName,N);

% Apply the window to the data.
ret = data.*window;
```

Script Files

Converting Script M-Files to Function M-Files

MATLAB provides two ways to package sequences of MATLAB commands:

- Function M-files
- Script M-files

These two categories of M-files differ in two important respects:

- You can pass arguments to function M-files but not to script M-files.
- Variables used inside function M-files are local to that function; you cannot access these variables from the MATLAB interpreter's workspace unless they are passed back by the function. By contrast, variables used inside script M-files are shared with the caller's workspace; you can access these variables from the MATLAB interpreter command line.

The MATLAB Compiler cannot compile script M-files, however, it can compile function M-files that call scripts. You may not specify a script M-file explicitly on the `mcc` command line, but you may specify function M-files that include scripts themselves.

Converting a script into a function is usually fairly simple. To convert a script to a function, simply add a `function` line at the top of the M-file.

For example, consider the script M-file `houdini.m`.

```
m = magic(4); % Assign 4x4 magic square to m.
t = m .^ 3;   % Cube each element of m.
disp(t);      % Display the value of t.
```

Running this script M-file from a MATLAB session creates variables `m` and `t` in your MATLAB workspace.

The MATLAB Compiler cannot compile `houdini.m` because `houdini.m` is a script. Convert this script M-file into a function M-file by simply adding a `function` header line.

```
function houdini(sz)
m = magic(sz); % Assign magic square to m.
t = m .^ 3;    % Cube each element of m.
disp(t)        % Display the value of t.
```

The MATLAB Compiler can now compile `houdini.m`. However, because this makes `houdini` a function, running `houdini.m` no longer creates variables `m` and `t` in the MATLAB workspace. If it is important to have `m` and `t` accessible from the MATLAB workspace, you can change the beginning of the function to

```
function [m,t] = houdini(sz)
```

The function now returns the values of `m` and `t` to its caller.

Including Script Files in Deployed Applications

Compiled applications consist of two layers of M-files. The top layer is the interface layer and consists of those functions that are directly accessible from C or C++.

In stand-alone applications, the interface layer consists of only the main M-file. In libraries, the interface layer consists of the M-files specified on the `mcc` command line.

The second layer of M-files in compiled applications includes those M-files that are called by the functions in the top layer. You can include scripts in the second layer, but not in the top layer.

5

Stand-Alone Applications

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

Introduction

Suppose you want to create an application that calculates the rank of a large magic square. One way to create this application is to code the whole application in C or C++; however, this would require writing your own magic square, rank, and singular value routines. An easier way to create this application is to write it as one or more M-files, taking advantage of the power of MATLAB and its tools.

You can create MATLAB applications that take advantage of the mathematical functions of MATLAB, yet do not require that end-users own MATLAB. Stand-alone applications are a convenient way to package the power of MATLAB and to distribute a customized application to your users.

The source code for stand-alone C applications consists either entirely of M-files or some combination of M-files, MEX-files, and C or C++ source code files.

The MATLAB Compiler takes your M-files and generates C source code functions that allow your M-files to be invoked from outside of interactive MATLAB. After compiling this C source code, the resulting object file is linked with the run-time libraries. A similar process is used to create C++ stand-alone applications.

You can call MEX-files from Compiler-generated stand-alone applications. The MEX-files will then be loaded and called by the stand-alone code.

C Stand-Alone Application Target

This section provides an example that illustrates the complete cycle of compiling an application and deploying it to a user's machine.

Magic Square Example

This example takes an M-file, `magicsquare.m`, and creates a stand-alone C application, `magicsquare`.

Compiling the Application

1. Copy the file `magicsquare.m` from

   ```
   <matlabroot>/extern/examples/compiler
   ```

   to your work directory.

2. To compile the M-code, use

   ```
   mcc -mv magicsquare.m
   ```

   The `-m` option tells the MATLAB Compiler (`mcc`) to generate a C stand-alone application. The `-v` option (verbose) displays the compilation steps throughout the process and helps identify other useful information such as which third-party compiler is used and what environment variables are referenced.

   This command creates the stand-alone application called `magicsquare` and additional files. The Windows platform appends the `.exe` extension to the name. See the table in "Stand-Alone Executable" on page 3-6 for the complete list of files created.

Testing the Application

These steps test your stand-alone application on your development machine.

Note Testing your application on your development machine is an important step to help ensure that your application is compilable. To verify that your application compiled properly, you must test all functionality that is available with the application. If you receive an error message similar to `Undefined function` or `Attempt to execute script` *`script_name`* `as a function`, it is likely that the application will not run properly on deployment machines. Most likely, your CTF archive is missing some necessary functions. Use `-a` to add the missing functions to the archive and recompile your code.

---

1 Update your dynamic library path as follows:

Windows. Add the following directory to your dynamic library path.

```
<matlabroot>\bin\win32
```

Linux. Add the following directories to your dynamic library path.

Note For readability, the following command appears on separate lines, but you must enter it all on one line.

```
setenv LD_LIBRARY_PATH
   <matlabroot>/bin/glnx86:
   <matlabroot>/sys/os/glnx86:
   <matlabroot>/sys/java/jre/glnx86/jre1.4.2/lib/i386/client:
   <matlabroot>/sys/java/jre/glnx86/jre1.4.2/lib/i386:
   <matlabroot>/sys/opengl/lib/glnx86:${LD_LIBRARY_PATH}

setenv XAPPLRESDIR <matlabroot>/X11/app-defaults
```

2 Run the stand-alone application from the system prompt (shell prompt on UNIX, DOS prompt on Windows) by typing the application name.

```
magicsquare.exe 4                    (On Windows)
```

```
magicsquare 4                    (On Linux)
```

The results are displayed as

```
ans =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Deploying the Application

You can distribute a MATLAB Compiler-generated stand-alone to any target machine that has the same operating system as the machine on which the application was compiled. For example, if you want to deploy an application to a Windows machine, you must use the MATLAB Compiler to build the application on a Windows machine. If you want to deploy the same application to a Linux machine, you must use the MATLAB Compiler on a Linux machine and completely rebuild the application. To deploy an application to multiple platforms requires MATLAB and MATLAB Compiler licenses on all the desired platforms.

These steps describe how to distribute a stand-alone application:

1. Generate the MATLAB Component Runtime (MCR) library archive on the development machine. You only need to do this step once per platform. To generate the MCR library archive, run

   ```
   buildmcr;
   ```

   This places the MCRInstaller archive `MCRInstaller.zip` in the `<matlabroot>/toolbox/compiler/deploy/<arch>` directory.

   Alternatively, to build the MCRInstaller archive as `filename` in the `path` directory, you can use

   ```
   buildmcr(path, filename);
   ```

   To return the full path to the file `zipfile`, use

   ```
   zipfile = buildmcr(path, filename);
   ```

To build the MCRInstaller archive in the current directory, use

```
zipfile = buildmcr('.');
```

2 Gather and package the following files and distribute them to the deployment machine.

Component	Description
`MCRInstaller.zip`	(Linux) MATLAB Component Runtime library archive; Platform-dependent file that must correspond to the end user's platform
`MCRInstaller.exe`	(Windows) Self-extracting MATLAB Component Runtime library utility; Platform-dependent file that must correspond to the end user's platform
`unzip`	(Linux) Utility to unzip `MCRInstaller.zip` (optional). The target machine must have an unzip utility installed.
`magicsquare.ctf`	Component Technology File archive; Platform-dependent file that must correspond to the end user's platform
`magicsquare`	Application; `magicsquare.exe` for Windows

Running the Application

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

Preparing Windows Machines.

1 Install the MCR by running the MCR Installer in a directory. For example, run `MCRInstaller.exe` in `C:\MCR`. For more information on running the MCR Installer utility, see "Installing the MCR on a Deployment Machine" on page 3-12.

2 Copy the component and CTF archive to your application root directory, for example, `C:\approot`.

3 Add the following directory to your dynamic library path.

```
<mcr_root>\runtime\win32
```

Preparing Linux Machines.

1 Unzip and install the MCR library archive (`MCRInstaller.zip`) on your deployment machine in a directory, say `<mcr_root>`. You may choose any directory for `<mcr_root>` except `<matlabroot>` or any directory below `<matlabroot>`.

Note This book uses `<mcr_root>` to refer to the directory where these MCR library archive files are installed on your machine.

---

2 Copy the component and CTF archive to your application root directory, for example, `/home/<user>/approot`.

3 Update your dynamic library path.

Note For readability, the following command appears on separate lines, but you must enter it all on one line.

```
setenv LD_LIBRARY_PATH
   <mcr_root>/runtime/glnx86:
   <mcr_root>/sys/os/glnx86:
   <mcr_root>/sys/java/jre/glnx86/jre1.4.2/lib/i386/client:
   <mcr_root>/sys/java/jre/glnx86/jre1.4.2/lib/i386:
   <mcr_root>/sys/opengl/lib/glnx86:${LD_LIBRARY_PATH}

setenv XAPPLRESDIR <matlabroot>/X11/app-defaults
```

Note There is a limitation regarding directories on your path. If the target machine has a MATLAB installation, the `<mcr_root>` directories must be first on the path in order to run the deployed application. To run MATLAB, the `matlabroot` directories must be first on the path. This restriction only applies to configurations involving an installed MCR and an installed MATLAB on the same machine.

Executing the Application.

Run the `magicsquare` stand-alone application from the system prompt and provide a number representing the size of the desired magic square, for example, 4.

```
magicsquare 4
```

The results are displayed as:

```
ans =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Coding with M-Files Only

One way to create a stand-alone application is to write all the source code in one or more M-files or MEX-files as in the previous magic square example. Coding an application in M-files allows you to take advantage of the MATLAB interactive development environment. Once the M-file version of your program works properly, compile the code and build it into a stand-alone application.

Example

Consider a simple application whose source code consists of two M-files, `mrank.m` and `main.m`. This example generates C code from your M-files.

mrank.m

`mrank.m` returns a vector of integers, `r`. Each element of `r` represents the rank of a magic square. For example, after the function completes, `r(3)` contains the rank of a 3-by-3 magic square.

```
function r = mrank(n)
r = zeros(n,1);
for k = 1:n
   r(k) = rank(magic(k));
end
```

In this example, the line `r = zeros(n,1)` preallocates memory to help the performance of the MATLAB Compiler.

main.m

`main.m` contains a "main routine" that calls `mrank` and then prints the results.

```
function main
r = mrank(5)
```

Compiling the Example

To compile these into code that can be built into a stand-alone application, invoke the MATLAB Compiler.

```
mcc -m main mrank
```

The `-m` option causes the MATLAB Compiler to generate C source code suitable for stand-alone applications. For example, the MATLAB Compiler generates C

source code files `main.c`, `main_main.c`, and `mrank.c`. `main_main.c` contains a C function named `main`; `main.c` and `mrank.c` contain C functions named `mlfMain` and `mlfMrank`.

To build an executable application, you can use `mbuild` to compile and link these files. Or, you can automate the entire build process (invoke the MATLAB Compiler on both M-files, use `mbuild` to compile the files with your ANSI C compiler, and link the code) by using the command

```
mcc -m main mrank
```

If you need to combine other code with your application (FORTRAN, for example, a language not supported by the MATLAB Compiler), or if you want to build a makefile that compiles your application, you can use the command

```
mcc -mc main mrank
```

The `-c` option inhibits invocation of `mbuild`.

Mixing M-Files and C or C++

The examples in this section illustrate how to mix M-files and C or C++ source code files:

- The first example is a simple application that mixes M-files and C code.
- The second example illustrates how to write C code that calls a compiled M-file.

One way to create a stand-alone application is to code some of it as one or more function M-files and to code other parts directly in C or C++. To write a stand-alone application this way, you must know how to

- Call the external C or C++ functions generated by the MATLAB Compiler.
- Handle the results these C or C++ functions return.

Note If you include compiled M-code into a larger application, you must produce a library wrapper file even if you do not actually create a separate library. For more information on creating libraries, see Chapter 6, "Libraries."

Simple Example

This example involves mixing M-files and C code. Consider a simple application whose source code consists of `mrank.m` and `mrankp.c`.

mrank.m

`mrank.m` contains a function that returns a vector of the ranks of the magic squares from 1 to n.

```
function r = mrank(n)
r = zeros(n,1);
for k = 1:n
   r(k) = rank(magic(k));
end
```

The Build Process

The steps needed to build this stand-alone application are

1. Compile the M-code.
2. Generate the library wrapper file.

To perform these steps, use

```
mcc -W lib:libPkg -T link:exe mrank printmatrix mrankp.c
```

The MATLAB Compiler generates the following C source code files:

- `libPkg.c`
- `libPkg.ctf`
- `libPkg.h`
- `libPkg_mcc_component_data.c`

This command invokes `mbuild` to compile the resulting Compiler-generated source files with the existing C source file (`mrankp.c`) and links against the required libraries.

The MATLAB Compiler provides two different versions of `mrankp.c` in the `<matlabroot>/extern/examples/compiler` directory:

- `mrankp.c` contains a POSIX-compliant `main` function. `mrankp.c` sends its output to the standard output stream and gathers its input from the standard input stream.
- `mrankwin.c` contains a Windows version of `mrankp.c`.

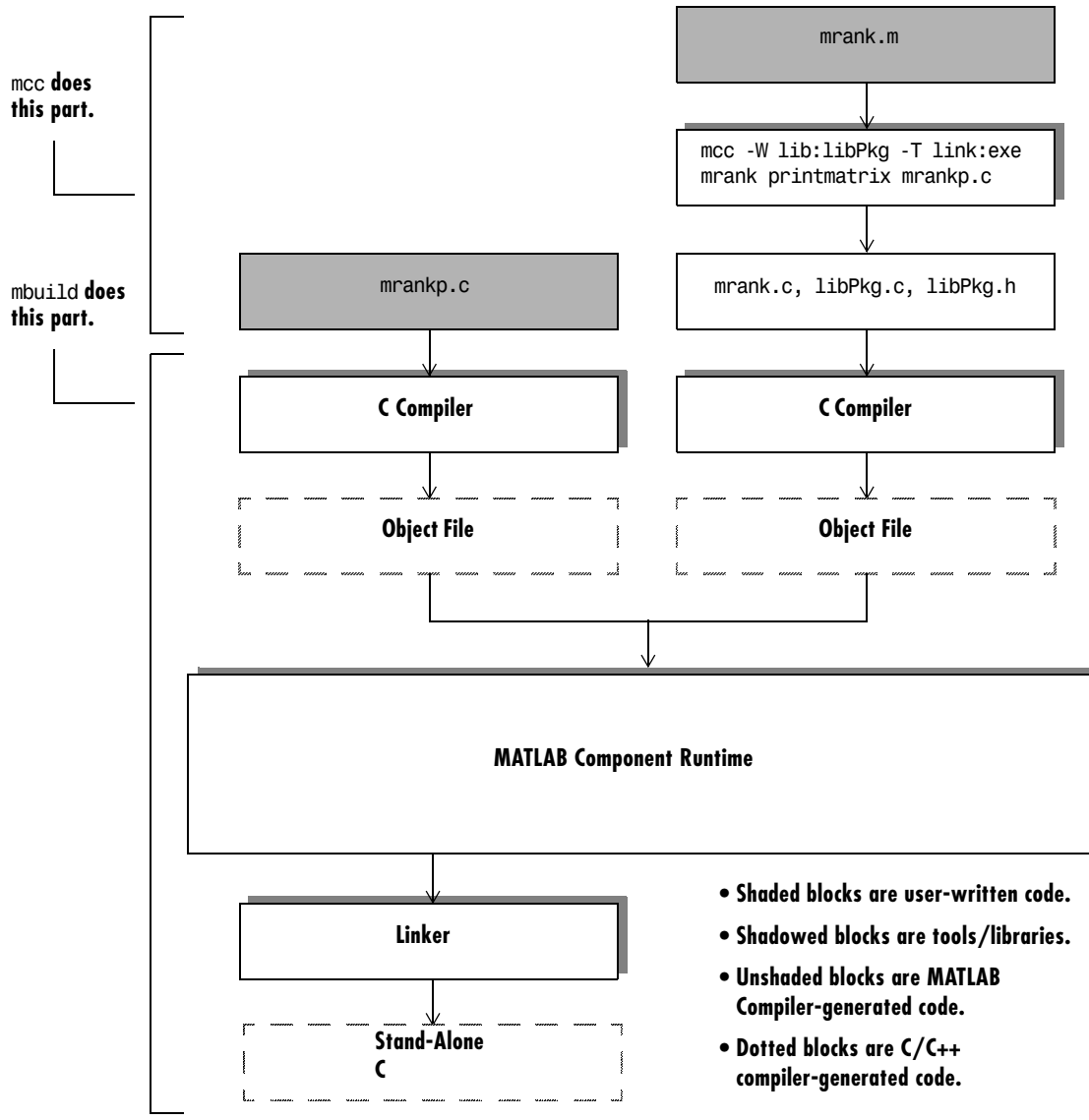


Figure 5-1: Mixing M-Files and C Code to Form a Stand-Alone Application

mrankp.c

The code in `mrankp.c` calls `mrank` and outputs the values that `mrank` returns.

```
/*
 * MRANKP.C
 * "Posix" C main program
 * Calls mlfMrank, obtained by using MCC to compile mrank.m.
 *
 * $Revision: 1.3.16.2 $
 *
 */

#include <stdio.h>
#include <math.h>
#include "libPkg.h"

main( int argc, char **argv )
{
    mxArray *N;        /* Matrix containing n. */
    mxArray *R = NULL; /* Result matrix. */
    int      n;        /* Integer parameter from command line. */

    /* Get any command line parameter. */
    if (argc >= 2) {
        n = atoi(argv[1]);
    } else {
        n = 12;
    }
    mclInitializeApplication(NULL,0);
   libPkgInitialize();/* Initialize the library of M-Functions */

    /* Create a 1-by-1 matrix containing n. */
    N = mxCreateScalarDouble(n);

    /* Call mlfMrank, the compiled version of mrank.m. */
    mlfMrank(1, &R, N);

    /* Print the results. */
    mlfPrintmatrix(R);
```

```
    /* Free the matrices allocated during this computation. */
    mxDestroyArray(N);
    mxDestroyArray(R);

    libPkgTerminate(); /* Terminate the library of M-functions */
    mclTerminateApplication();
}
```

An Explanation of mrankp.c

The heart of `mrankp.c` is a call to the `mlfMrank` function. Most of what comes before this call is code that creates an input argument to `mlfMrank`. Most of what comes after this call is code that displays the vector that `mlfMrank` returns. First, the code must initialize the MCR and the generated `libPkg` library.

```
mclInitializeApplication(NULL,0);
libPkgInitialize();   /* Initialize the library of M-Functions */
```

To understand how to call `mlfMrank`, examine its C function header, which is

```
void mlfMrank(int nargout, mxArray** r, mxArray* n);
```

According to the function header, `mlfMrank` expects one input parameter and returns one value. All input and output parameters are pointers to the `mxArray` data type. (See the External Interfaces documentation for details on the `mxArray` data type.)

To create and manipulate `mxArray *` variables in your C code, you can call the `mx` routines described in the External Interfaces documentation. For example, to create a 1-by-1 `mxArray *` variable named `N` with real data, `mrankp` calls `mxCreateScalarDouble`.

```
N = mxCreateScalarDouble(n);
```

`mrankp` can now call `mlfMrank`, passing the initialized `N` as the sole input argument.

```
R = mlfMrank(1,&R,N);
```

`mlfMrank` returns its output in a newly allocated `mxArray *` variable named `R`. The variable `R` is initialized to `NULL`. Output variables that have not been assigned to a valid `mxArray` should be set to `NULL`. The easiest way to display the contents of `R` is to call the `mlfPrintmatrix` function:

```
mlfPrintmatrix(R);
```

This function is defined in `Printmatrix.m`.

Finally, `mrankp` must free the heap memory allocated to hold matrices and call the termination functions.

```
mxDestroyArray(N);
mxDestroyArray(R);
libPkgTerminate();     /* Terminate the library of M-functions */
mclTerminateApplication();  /* Terminate the MCR */
```

Advanced C Example

This section illustrates an advanced example of how to write C code that calls a compiled M-file. Consider a stand-alone application whose source code consists of two files:

- `multarg.m`, which contains a function named `multarg`
- `multargp.c`, which contains a C function named `main`

`multarg.m` specifies two input parameters and returns two output parameters.

```
function [a,b] = multarg(x,y)
a = (x + y) * pi;
b = svd(svd(a));
```

The code in `multargp.c` calls `mlfMultarg` and then displays the two values that `mlfMultarg` returns.

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "libMultpkg.h"

/*
 * Function prototype; the MATLAB Compiler creates mlfMultarg
 *  from multarg.m
 */

void PrintHandler( const char *text )
{
    printf(text);
```

```
}

int main( )   /* Programmer written coded to call mlfMultarg */
{
#define ROWS  3
#define COLS  3
    mclOutputHandlerFcn PrintHandler;
    mxArray *a = NULL, *b = NULL, *x, *y;
    double  x_pr[ROWS * COLS] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    double  x_pi[ROWS * COLS] = {9, 2, 3, 4, 5, 6, 7, 8, 1};
    double  y_pr[ROWS * COLS] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    double  y_pi[ROWS * COLS] = {2, 9, 3, 4, 5, 6, 7, 1, 8};
    double *a_pr, *a_pi, value_of_scalar_b;

    /* Initialize with a print handler to tell mlfPrintMatrix
     * how to display its output.
     */
    mclInitializeApplication(NULL,0);
   libMultpkgInitializeWithHandlers(PrintHandler, PrintHandler);


    /* Create input matrix "x" */
    x = mxCreateDoubleMatrix(ROWS, COLS, mxCOMPLEX);
    memcpy(mxGetPr(x), x_pr, ROWS * COLS * sizeof(double));
    memcpy(mxGetPi(x), x_pi, ROWS * COLS * sizeof(double));

    /* Create input matrix "y" */
    y = mxCreateDoubleMatrix(ROWS, COLS, mxCOMPLEX);
    memcpy(mxGetPr(y), y_pr, ROWS * COLS * sizeof(double));
    memcpy(mxGetPi(y), y_pi, ROWS * COLS * sizeof(double));

    /* Call the mlfMultarg function. */
    mlfMultarg(2, &a, &b, x, y);

    /* Display the entire contents of output matrix "a". */
    mlfPrintmatrix(a);

    /* Display the entire contents of output scalar "b" */
    mlfPrintmatrix(b);
```

```
    /* Deallocate temporary matrices. */
    mxDestroyArray(a);
    mxDestroyArray(b);
    libMultpkgTerminate();
    mclTerminateApplication();
    return(0);
}
```

You can build this program into a stand-alone application by using the command

```
mcc -W lib:libMultpkg -T link:exe multarg printmatrix multargp.c
```

The program first displays the contents of a 3-by-3 matrix a and then displays the contents of scalar b.

```
 6.2832 +34.5575i  25.1327 +25.1327i  43.9823 +43.9823i
12.5664 +34.5575i  31.4159 +31.4159i  50.2655 +28.2743i
18.8496 +18.8496i  37.6991 +37.6991i  56.5487 +28.2743i

143.4164
```

An Explanation of This C Code

Invoking the MATLAB Compiler on `multarg.m` generates the C function prototype.

```
extern void mlfMultarg(int nargout, mxArray** a, mxArray** b,
mxArray* x, mxArray* y);
```

This C function header shows two input arguments (`mxArray* x` and `mxArray* y`) and two output arguments (the return value and `mxArray** b`).

Use `mxCreateDoubleMatrix` to create the two input matrices (`x` and `y`). Both `x` and `y` contain real and imaginary components. The `memcpy` function initializes the components, for example:

```
x = mxCreateDoubleMatrix(ROWS, COLS, mxCOMPLEX);
memcpy(mxGetPr(x), x_pr, ROWS * COLS * sizeof(double));
memcpy(mxGetPi(y), x_pi, ROWS * COLS * sizeof(double));
```

The code in this example initializes variable `x` from two arrays (`x_pr` and `x_pi`) of predefined constants. A more realistic example would read the array values from a data file or a database.

After creating the input matrices, `main` calls `mlfMultarg`.

```
mlfMultarg(2, &a, &b, x, y);
```

The `mlfMultarg` function returns matrices `a` and `b`. `a` has both real and imaginary components; `b` is a scalar having only a real component. The program uses `mlfPrintmatrix` to output the matrices, for example:

```
mlfPrintmatrix(a);
```

6

Libraries

This chapter describes how to use the MATLAB Compiler to create libraries.

Introduction

You can use the MATLAB Compiler to create C or C++ shared libraries (DLLs on Windows) from your MATLAB algorithms. You can then write C or C++ programs that can call the MATLAB functions in the shared library, much like calling the functions from the MATLAB command line.

C Shared Library Target

You can use the MATLAB Compiler to build C or C++ shared libraries on both Windows and Linux. Many of the `mcc` options that pertain to creating stand-alone applications also pertain to creating C and C++ shared libraries.

C Shared Library Wrapper

The C library wrapper option allows you to create a shared library from an arbitrary set of M-files. The MATLAB Compiler generates a wrapper file, a header file, and an export list. The header file contains all of the entry points for all of the compiled M-functions. The export list contains the set of symbols that are exported from a C shared library.

Note Even if you are not producing a shared library, you must use `-W lib` or `-W cpplib` when including any Compiler-generated code into a larger application. For more information, refer to "Mixing M-Files and C or C++" on page 5-11.

C Shared Library Example

This example takes several M-files and creates a C shared library. It also includes a stand-alone driver application to call the shared library.

Building the Shared Library

1. Copy the following files from `<matlabroot>/extern/examples/compiler` to your work directory.

   ```
   <matlabroot>/extern/examples/compiler/addmatrix.m
   <matlabroot>/extern/examples/compiler/multiplymatrix.m
   <matlabroot>/extern/examples/compiler/eigmatrix.m
   <matlabroot>/extern/examples/compiler/matrixdriver.c
   ```

Note `matrixdriver.c` contains the stand-alone application's `main` function.

---

2 To create the shared library, use

```
mcc -B csharedlib:libmatrix addmatrix.m multiplymatrix.m
eigmatrix.m -v
```

The `-B csharedlib` option is a bundle option that expands into

```
-W lib:<libname> -T link:lib
```

The `-W lib:<libname>` option tells the MATLAB 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 Compiler puts the shared library because you will need it later on.

Writing the Driver Application

All programs that call MATLAB Compiler-generated shared libraries have roughly the same structure:

1 Declare variables and process/validate input arguments.

2 Call `mclInitializeApplication`, and test for success. This function sets up the global MCR state and enables the construction of MCR instances.

3 Call, once for each library, `<libraryname>Initialize`, to create the MCR instance required by the library.

4 Invoke functions in the library, and process the results. (This is the main body of the program.)

5 Call, once for each library, `<libraryname>Terminate`, to destroy the associated MCR.

6 Call `mclTerminateApplication` to free resources associated with the global MCR state.

7 Clean up variables, close files, etc., and exit.

This example uses `matrixdriver.c` as the driver application.

Note You must call `mclInitializeApplication` once at the beginning of your driver application. You must make this call before calling any other MathWorks functions. See "Calling a Shared Library" on page 6-9 for complete details on using a Compiler-generated library in your application.

Compiling the Driver Application

To compile the driver code, `matrixdriver.c`, you use your C/C++ compiler. Execute the following `mbuild` command that corresponds to your development platform. This command uses your C/C++ compiler to compile the code.

```
mbuild matrixdriver.c libmatrix.lib    (Windows)
mbuild matrixdriver.c -L. -lmatrix -I. (Linux)
```

Note This command assumes that the shared library and the corresponding header file created from step 2 are in the current working directory.

On Linux, if this is not the case, replace the "`.`" (dot) following the `-L` and `-I` options with the name of the directory that contains these files, respectively.

On Windows, if this is not the case, specify the full path to `libmatrix.lib`.

This generates a stand-alone application, `matrixdriver.exe`, on Windows, and `matrixdriver`, on Linux.

Difference in the Exported Function Signature. The interface to the `mlf` functions generated by the Compiler from your M-file routines has changed in this version of the Compiler. The generic signature of the exported `mlf` functions is

- M-functions with no return values

```
void mlf<function-name>(<list_of_input_variables>);
```

- M-functions with at least one return value

```
void mlf<function-name>(int number_of_return_values,
<list_of_pointer_to_return_variables>,
<list_of_input_variables>);
```

Refer to the header file generated for your library for the exact signature of the exported function. For example, in the library created in the previous section, the signature of the exported `addmatrix` function is

```
void mlfAddmatrix(int nlhs,mxArray **a,mxArray *a1,mxArray *a2);
```

Testing the Driver Application

These steps test your stand-alone driver application and shared library on your development machine.

Note Testing your application on your development machine is an important step to help ensure that your application is compilable. To verify that your application compiled properly, you must test all functionality that is available with the application. If you receive an error message similar to `Undefined function` or `Attempt to execute script` *`script_name`* `as a function`, it is likely that the application will not run properly on deployment machines. Most likely, your CTF archive is missing some necessary functions. Use `-a` to add the missing functions to the archive and recompile your code.

---

1. To run the stand-alone application, add the directory containing the shared library that was created in step 2 in "Building the Shared Library" on page 6-3 to your dynamic library path.

2. Update your dynamic library path as follows:

Windows. Add the following directory to your dynamic library path.

```
<matlabroot>\bin\win32
```

Linux. Add the following directories to your dynamic library path.

Note For readability, the following command appears on separate lines, but you must enter it all on one line.

```
setenv LD_LIBRARY_PATH
   <matlabroot>/bin/glnx86:
```

```
    <matlabroot>/sys/os/glnx86:
    <matlabroot>/sys/java/jre/glnx86/jre1.4.2/lib/i386/client:
    <matlabroot>/sys/java/jre/glnx86/jre1.4.2/lib/i386:
    <matlabroot>/sys/opengl/lib/glnx86:${LD_LIBRARY_PATH}

setenv XAPPLRESDIR <matlabroot>/X11/app-defaults
```

3 Run the driver application from the prompt (DOS prompt on Windows, shell prompt on Linux) by typing the application name.

```
matrixdriver.exe          (On Windows)
matrixdriver              (On Linux)
```

The results are displayed as:

```
The value of the added matrix is:
2.00   4.00    6.00
8.00   10.00   12.00
14.00  16.00   18.00

The value of the multiplied matrix is:
30.00  36.00   42.00
66.00  81.00   96.00
102.00 126.00  150.00

The eigenvalue of the first matrix is:
16.12  -1.12   -0.00
```

Deploying Stand-Alone Applications That Call MATLAB Compiler-Based Shared Libraries

Gather and package the following files and distribute them to the deployment machine.

Component	Description
`MCRInstaller.zip`	(Linux) MATLAB Component Runtime library archive; platform-dependent file that must correspond to the end user's platform
`MCRInstaller.exe`	(Windows) Self-extracting MATLAB Component Runtime library utility; platform-dependent file that must correspond to the end user's platform
`unzip`	(Linux) Utility to unzip `MCRInstaller.zip` (optional). The target machine must have an unzip utility installed.
`matrixdriver.ctf`	Component Technology File archive; platform-dependent file that must correspond to the end user's platform
`matrixdriver`	Application; `matrixdriver.exe` for Windows
`libmatrix`	Shared library; extension varies by platform, for example, `DLL` on Windows

Note You can distribute a MATLAB Compiler-generated stand-alone application to any target machine that has the same operating system as the machine on which the application was compiled. If you want to deploy the same application to a different platform, you must use the MATLAB Compiler on the different platform and completely rebuild the application.

Deploying Shared Libraries to be Used with Other Projects

To distribute the shared library for use with an external application, you need to distribute the following.

Component	Description
`MCRInstaller.zip`	(Linux) MATLAB Component Runtime library archive; platform-dependent file that must correspond to the end user's platform
`MCRInstaller.exe`	(Windows) Self-extracting MATLAB Component Runtime library utility; platform-dependent file that must correspond to the end user's platform
`unzip`	(Linux) Utility to unzip `MCRInstaller.zip` (optional). The target machine must have an unzip utility installed.
`libmatrix.ctf`	Component Technology File archive; platform-dependent file that must correspond to the end user's platform
`libmatrix`	Shared library; extension varies by platform, for example, `DLL` on Windows
`libmatrix.h`	Library header file

Calling a Shared Library

At run-time, there is an MCR instance associated with each individual shared library. Consequently, if an application links against two MATLAB Compiler-generated shared libraries, there will be two MCR instances created at run-time.

You can control the behavior of each MCR instance by using MCR options. The two classes of MCR options are global and local. Global MCR options are identical for each MCR instance in an application. Local MCR options may differ for MCR instances.

To use a shared library, you must use these functions:

- `mclInitializeApplication`
- `mclTerminateApplication`

`mclInitializeApplication` allows you to set the global MCR options. They apply equally to all MCR instances. You must set these options before creating your first MCR instance.

These functions are necessary because some MCR options such as whether or not to start Java, the location of the MCR itself, whether or not to use the MATLAB JIT feature, and so on, are set when the first MCR instance starts and cannot be changed by subsequent instances of the MCR.

Note You must call `mclInitializeApplication` once at the beginning of your driver application. You must make this call before calling any other MathWorks functions.

Function Signatures

The function signatures are

```
bool mclInitializeApplication(const char **options, int count);
bool mclTerminateApplication(void);
```

mclInitializeApplication. Takes an array of strings of user-settable options (these are the very same options that can be provided to `mcc` via the `-R` option) and a count of the number of options (the length of the option array). Returns `true` for success and `false` for failure.

mclTerminateApplication. Takes no arguments and can *only* be called after all MCR instances have been destroyed. Returns `true` for success and `false` for failure.

Note After you call `mclTerminateApplication`, you may not call `mclInitializeApplication` again. No MathWorks functions may be called after `mclTerminateApplication`.

This C example shows typical usage of the functions.

```
int main(){

    mxArray *in1, *in2; /* Define input parameters */
    mxArray *out = NULL;/* and output parameters to be passed to
                           the library functions */

    double data[] = {1,2,3,4,5,6,7,8,9};

    /* Call the library intialization routine and make sure that
       the library was initialized properly */
    mclInitializeApplication(NULL,0);
    if (!libmatrixInitialize()){
        fprintf(stderr,"could not initialize the library
                        properly\n");
        return -1;
    }

    /* Create the input data */
    in1 = mxCreateDoubleMatrix(3,3,mxREAL);
    in2 = mxCreateDoubleMatrix(3,3,mxREAL);
    memcpy(mxGetPr(in1), data, 9*sizeof(double));
    memcpy(mxGetPr(in2), data, 9*sizeof(double));

    /* Call the library function */
    mlfAddmatrix(1, &out, in1, in2);
    /* Display the return value of the library function */
    printf("The value of added matrix is:\n");
    display(out);
   /* Destroy the return value since this variable will be reused
       in the next function call. Since we are going to reuse the
       variable, we have to set it to NULL. Refer to MATLAB
       Compiler documentation for more information on this. */
    mxDestroyArray(out); out=0;
    mlfMultiplymatrix(1, &out, in1, in2);
    printf("The value of the multiplied matrix is:\n");
    display(out);
    mxDestroyArray(out); out=0;
    mlfEigmatrix(1, &out, in1);
    printf("The Eigen value of the first matrix is:\n");
    display(out);
```

```
    mxDestroyArray(out); out=0;

    /* Call the library termination routine */
    libmatrixTerminate();

    /* Free the memory created */
    mxDestroyArray(in1); in1=0;
    mxDestroyArray(in2); in2 = 0;
    mclTerminateApplication();
    return 1;
}
```

Note `mclInitializeApplication` can only be called *once* per application. Calling it a second time is an error, and will cause the function to return `false`. This function must be called before calling any C MX-function or MAT-file API function.

Steps to Use a Shared Library

To use a MATLAB Compiler-generated shared library in your application, you must perform the following steps:

1. Include the generated header file for each library in your application. Each MATLAB Compiler-generated shared library has an associated header file named `<lib-name>.h`, where `<lib-name>` is the library's name that was passed in on the command line when the library was compiled.

2. Initialize the MATLAB libraries by calling the `mclInitializeApplication` API function. You must call this function once per application, and it must be called before calling any other MATLAB API functions, such as C MX-functions or C MAT-file functions. `mclInitializeApplication` must be called before calling any functions in a MATLAB Compiler-generated shared library. You may optionally pass in application-level options to this function. `mclInitializeApplication` returns a Boolean status code. A return value of `true` indicates successful initialization, and `false` indicates failure.

3. For each MATLAB Compiler-generated shared library that you include in your application, call the library's initialization function. This function

performs several library-local initializations, such as unpacking the CTF archive, and starting an MCR instance with the necessary information to execute the code in that archive. The library initialization function will be named `<lib-name>Initialize()`, where `<lib-name>` is the library's name that was passed in on the command line when the library was compiled. This function returns a Boolean status code. A return value of `true` indicates successful initialization, and `false` indicates failure.

4. Call the exported functions of each library as needed. Use the C MX API to process input and output arguments for these functions.

5. When your application no longer needs a given library, call the library's termination function. This function frees the resources associated with its MCR instance. The library termination function will be named `<lib-name>Terminate()`, where `<lib-name>` is the library's name that was passed in on the command line when the library was compiled. Once a library has been terminated, that library's exported functions should not be called again in the application.

6. When your application no longer needs to call any MATLAB Compiler-generated libraries, call the `mclTerminateApplication` API function. This function frees application-level resources used by the MCR. Once you call this function, no further calls can be made to MATLAB Compiler-generated libraries in the application.

C++ Shared Library Target

C++ Shared Library Wrapper

The C++ library wrapper option allows you to create a shared library from an arbitrary set of M-files. The MATLAB Compiler generates a wrapper file and a header file. The header file contains all of the entry points for all of the compiled M-functions.

Note Even if you are not producing a shared library, you must use `-W lib` or `-W cpplib` when including any Compiler-generated code into a larger application. For more information, refer to "Mixing M-Files and C or C++" on page 5-11.

C++ Shared Library Example

This example rewrites the previous C shared library example using C++. The procedure for creating a C++ shared library from M-files is identical to the procedure for creating a C shared library, except you use the `cpplib` wrapper.

```
mcc  W cpplib:libmatrix  T link:lib addmatrix.m multiplymatrix.m
eigmatrix.m -v
```

The `W cpplib:<libname>` option tells the MATLAB 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 Compiler puts the shared library because you will need it later on.

Writing the Driver Application

This example uses a C++ version of the `matrixdriver` application, `matrixdriver.cpp`.

```
/*================================================================
 *
 * MATRIXDRIVER.CPP
 * Sample driver code that calls a C++ shared library created using
 * the MATLAB Compiler. Refer to the MATLAB Compiler documentation
 * for more information on this
 *
```

```
 * This is the wrapper CPP code to call a shared library created
 * using the MATLAB Compiler.
 *
 * Copyright 1984-2004 The MathWorks, Inc.
 *
 *============================================================*/

// Include the library specific header file as generated by the
// MATLAB Compiler
#include "libmatrix.h"

int main(){


    // Call application and library initialization. Perform this
    // initialization before calling any API functions or
    // Compiler-generated libraries.
    if (!mclInitializeApplication(NULL,0) ||
        !libmatrixInitialize())
    {
        std::cerr << "could not initialize the library properly"
                  << std::endl;
        return -1;
    }

    try
    {
        // Create input data
        double data[] = {1,2,3,4,5,6,7,8,9};
        mwArray in1(3, 3, mxDOUBLE_CLASS, mxREAL);
        mwArray in2(3, 3, mxDOUBLE_CLASS, mxREAL);
        in1.SetData(data, 9);
        in2.SetData(data, 9);

        // Create output array
        mwArray out;

        // Call the library function
        addmatrix(1, out, in1, in2);
```

```
        // Display the return value of the library function
       std::cout << "The value of added matrix is:" << std::endl;
        std::cout << out << std::endl;

        multiplymatrix(1, out, in1, in2);
        std::cout << "The value of the multiplied matrix is:"
                  << std::endl;
        std::cout << out << std::endl;

        eigmatrix(1, out, in1);
        std::cout << "The eigenvalues of the first matrix are:"
                  << std::endl;
        std::cout << out << std::endl;
    }
    catch (const mwException& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;
    }
    catch (...)
    {
        std::cerr << "Unexpected error thrown" << std::endl;
        return -1;
    }

    // Call the application and library termination routine
    libmatrixTerminate();
    mclTerminateApplication();

    return 0;
}
```

Compiling the Driver Application

To compile the `matrixdriver.cpp` driver code, you use your C++ compiler. By executing the following `mbuild` command that corresponds to you development platform, you will use your C++ compiler to compile the code.

```
mbuild matrixdriver.cpp libmatrix.lib          (Windows)
mbuild matrixdriver.cpp  L.  lmatrix  I.       (Linux)
```

Incorporating a C++ Shared Library Into an Application

To incorporate a C++ shared library into your application, you will, in general, follow the steps listed in "Steps to Use a Shared Library" on page 6-12. There are two main differences to note when using a C++ shared library.

- Interface functions use the `mwArray` type to pass arguments, rather than the `mxArray` type used with C shared libraries.
- C++ exceptions are used to report errors to the caller. Therefore, all calls must be wrapped in a `try-catch` block.

Exported Function Signature

The C++ shared library target generates two sets of interfaces for each M-function. The first set of exported interfaces is identical to the `mlx` signatures that are generated in C shared libraries. The second set of interfaces is the C++ function interfaces. The generic signature of the exported C++ functions is

M-functions with no return values.

```
void <function-name>(<list_of_input_variables>);
```

M-functions with at least one return value.

```
void <function-name>(int number_of_return_values,
     <list_of_return_variables>, <list_of_input_variables>);
```

In this case, `<list_of_input_variables>` represents a comma-separated list of type `const mwArray&` and `<list_of_return_variables>` represents a comma-separated list of type `mwArray&`. For example, in the `libmatrix` library, the C++ interfaces to the `addmatrix` M-function is generated as

```
void addmatrix(int nargout, mwArray& a , const mwArray& a1,
               const mwArray& a2);
```

Error Handling

C++ interface functions handle errors during execution by throwing a C++ exception. Use the `mwException` class for this purpose. Your application can catch `mwExceptions` and query the `what()` method to get the error message. To correctly handle errors when calling the C++ interface functions, wrap each call inside a `try-catch` block.

```
   try
{
        .
        (call function)
        .
}
catch (const mwException& e)
{
        .
        (handle error)
        .
}
```

The `matrixdriver.cpp` application illustrates the typical way to handle errors when calling the C++ interface functions.

MATLAB Compiler-Generated Interface Functions

A shared library generated by the MATLAB Compiler contains at least seven functions. There are three generated functions to manage library initialization and termination, one each for printed output and error messages, and two generated functions for each M-file compiled into the library.

To generate the functions described in this section, first copy `sierpinski.m` and `triangle.c` from `<matlabroot>/extern/examples/compiler` into your directory, and then execute the appropriate Compiler command.

For a C application

On Windows.

```
mcc -W lib:libtriangle -T link:lib sierpinski.m
mbuild triangle.c libtriangle.lib
```

On Linux.

```
mcc -W lib:libtriangle -T link:lib sierpinski.m
mbuild triangle.c -L. -ltriangle -I.
```

For a C++ application

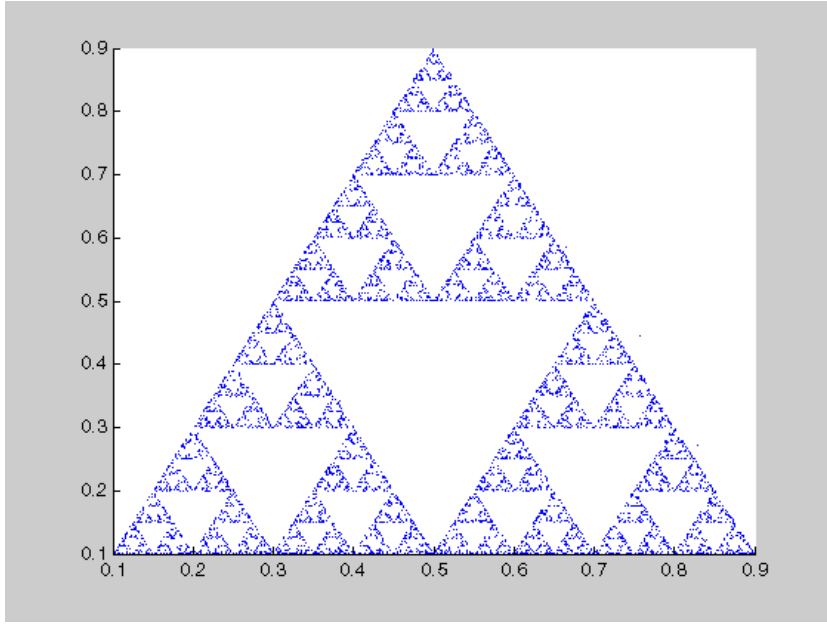
On Windows.

```
mcc -W cpplib:libtriangle -T link:lib sierpinski.m
mbuild triangle.cpp libtriangle.lib
```

On Linux.

```
mcc -W cpplib:libtriangle -T link:lib sierpinski.m
mbuild triangle.cpp -L. -ltriangle -I.
```

These commands create a main program named `triangle`, and a shared library named `libtriangle`. The library exports a single function that uses a simple iterative algorithm (contained in `sierpinski.m`) to generate the fractal known as Sierpinski's Triangle. The main program in `triangle.c` or `triangle.cpp` can optionally take a single numeric argument, which, if present, specifies the number of points used to generate the fractal. For example, `triangle 8000` generates a diagram with 8,000 points.



In this example the MATLAB Compiler places all of the generated functions discussed below into the generated file `libtriangle.c` or `libtriangle.cpp`.

Structure of Programs that Call Shared Libraries

All programs that call MATLAB Compiler-generated shared libraries have roughly the same structure:

1. Declare variables and process/validate input arguments.
2. Call `mclInitializeApplication`, and test for success. This function sets up the global MCR state and enables the construction of MCR instances.
3. Call, once for each library, `<libraryname>Initialize`, to create the MCR instance required by the library.
4. Invoke functions in the library, and process the results. (This is the main body of the program.)

5 Call, once for each library, `<libraryname>Terminate`, to destroy the associated MCR.

6 Call `mclTerminateApplication` to free resources associated with the global MCR state.

7 Clean up variables, close files, etc., and exit.

To see these steps in an actual example, review the main program in this example, `triangle.c`.

Library Initialization and Termination Functions

The library initialization and termination functions create and destroy, respectively, the MCR instance required by the shared library. You must call the initialization function before you invoke any of the other functions in the shared library, and you should call the termination function after you are finished making calls into the shared library (or you risk leaking memory).

There are two forms of the initialization function and one type of termination function. The simpler of the two initialization functions takes no arguments; most likely this is the version your application will call. In this example, this form of the initialization function is called `libtriangleInitialize`.

```
bool libtriangleInitialize(void)
```

This function creates an MCR instance using the default print and error handlers, and other information generated during the compilation process.

However, if you want more control over how printed output and error messages are handled, you may call the second form of the function, which takes two arguments.

```
bool libtriangleInitializeWithHandlers(
    mclOutputHandlerFcn error_handler,
    mclOutputHandlerFcn print_handler
)
```

By calling this function, you can provide your own versions of the print and error handling routines called by the MCR. Each of these routines has the same signature (for complete details, see “Print and Error Handling Functions” on page 6-22). By overriding the defaults, you can control how output is displayed and, for example, whether or not it goes into a log file.

Note Before calling either form of the library initialization routine, you must first call `mclInitializeApplication` to set up the global MCR state. See "Calling a Shared Library" on page 6-9 for more information.

On Microsoft Windows platforms, the Compiler generates an additional initialization function, the standard Microsoft DLL initialization function `DllMain`.

```
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason,
                    void *pv)
```

The generated `DllMain` performs a very important service; it locates the directory in which the shared library is stored on disk. This information is used to find the CTF archive, without which the application will not run. If you modify the generated `DllMain` (which we do not recommend you do), make sure you preserve this part of its functionality.

Library termination is simple.

```
void libtriangleTerminate(void)
```

Call this function (once for each library) before calling `mclTerminateApplication`.

Print and Error Handling Functions

By default, MATLAB Compiler-generated applications and shared libraries send printed output to standard output and error messages to standard error. The Compiler generates a default print handler and a default error handler that implement this policy. If you'd like to change this behavior, you must write your own error and print handlers and pass them in to the appropriate generated initialization function.

You may replace either, both, or neither of these two functions. Note that the MCR sends all regular output through the print handler and all error output through the error handler. Therefore, if you redefine either of these functions, the MCR will use your version of the function for all the output that falls into class for which it invokes that handler.

The default print handler takes the following form.

```
static int mclDefaultPrintHandler(const char *s)
```

The implementation is straightforward; it takes a string, prints it on standard output, and returns the number of characters printed. If you override or replace this function, your version must also take a string and return the number of characters "handled." The MCR calls the print handler when an executing M-file makes a request for printed output, e.g., via the MATLAB function `disp`. The print handler does not terminate the output with a carriage return or line feed.

The default error handler has the same form as the print handler.

```
static int mclDefaultErrorHandler(const char *s)
```

However, the default implementation of the print handler is slightly different. It sends the output to the standard error output stream, but if the string does not end with carriage return, the error handler adds one. If you replace the default error handler with one of your own, you should perform this check as well, or some of the error messages printed by the MCR will not be properly formatted.

Note The error handler, despite its name, does not handle the actual errors, but rather the message produced after the errors have been caught and handled inside the MCR. You cannot use this function to modify the error handling behavior of the MCR — use the `try` and `catch` statements in your M-files if you want to control how a MATLAB Compiler-generated application responds to an error condition.

Functions Generated from M-Files

For each M-file specified on the MATLAB Compiler command line, the Compiler generates two functions, the `mlx` function and the `mlf` function. Each of these generated functions performs the same action (calls your M-file function). The two functions have different names and present different interfaces. The name of each function is based on the name of the first function in the M-file (`sierpinski`, in this example); each function begins with a different three-letter prefix.

mlx Interface Function

The function that begins with the prefix `mlx` takes the same type and number of arguments as a MATLAB MEX-function. (See the External Interfaces documentation for more details on MEX-functions). The first argument, `nlhs`, is the number of output arguments, and the second argument, `plhs`, is a pointer to an array that the function will fill with the requested number of return values. (The "`lhs`" in these argument names is short for "left-hand side" — the output variables in a MATLAB expression are those on the left-hand side of the assignment operator.) The third and forth parameters are the number of inputs and an array containing the input variables.

```
void mlxSierpinski(int nlhs, mxArray *plhs[], int nrhs,
                   mxArray *prhs[])
```

mlf Interface Function

The second of the generated functions begins with the prefix `mlf`. This function expects its input and output arguments to be passed in as individual variables rather than packed into arrays. If the function is capable of producing one or more outputs, the first argument is the number of outputs requested by the caller.

```
void mlfSierpinski(int nargout, mxArray** x, mxArray** y,
                   mxArray* iterations, mxArray* draw)
```

Note that in both cases, the generated functions allocate memory for their return values. If you do not delete this memory (via `mxDestroyArray`) when you are done with the output variables, your program will leak memory.

Your program may call whichever of these functions is more convenient, as they both invoke your M-file function in an identical fashion. Most programs will likely call the `mlf` form of the function to avoid managing the extra arrays required by the `mlx` form. The example program in `triangle.c` calls `mlfSierpinski`.

```
mlfSierpinski(2, &x, &y, iterations, draw);
```

In this call, the caller requests two output arguments, `x` and `y`, and provides two inputs, `iterations` and `draw`.

If the output variables you pass in to an `mlf` function are nonNULL, the `mlf` function will attempt to free them using `mxDestroyArray`. This means that you can reuse output variables in consecutive calls to `mlf` functions without

worrying about memory leaks. It also implies that you must pass either `NULL` or a valid MATLAB array for all output variables or your program will fail because the memory manager cannot distinguish between a noninitialized (invalid) array pointer and a valid array. It will try to free a pointer that is not NULL — freeing an invalid pointer usually causes a segmentation fault or similar fatal error).

Using varargin and varargout in an M-Function Interface

If your M-function interface uses `varargin` or `varargout`, you must pass them as cell arrays. For example, if you have `N` `varargins`, you need to create one cell array of size `1-by-N`. Similarly, `varargouts` are returned back as one cell array. The length of the `varargout` is equal to the number of return values specified function call minus the number of actual variables passed. As in MATLAB, the cell array representing `varagout` has to be the last return variable (the variable preceding the first input variable) and the cell array representing `varargins` has to be the last formal parameter to the function call.

For information on creating cell arrays, refer to the C MX-function interface in the External Interfaces documentation.

For example, consider this M-file interface.

```
[a,b,varargout] = myfun(x,y,z,varargin)
```

The corresponding C interface for this is

```
void mlfMyfun(int numOfRetVars, mxArray **a, mxArray **b,
              mxArray **varargout, mxArray *x, mxArray *y,
              mxArray *z, mxArray *varargin)
```

In this example, the number of elements in `varargout` is `(numOfRetVars - 2)`, where 2 represents the two actual variables, `a` and `b`, being returned. Both `varargin` and `varargout` are single row, multiple column cell arrays.

7

COM and Excel Components

This chapter describes how to use the MATLAB Compiler to generate COM and Excel components.

Introduction

This chapter focuses on using two optional MATLAB Compiler-based products:

- MATLAB Builder for COM
- MATLAB Builder for Excel

These products add capabilities to the base MATLAB Compiler; they require the MATLAB Compiler and must be purchased separately.

Generating COM and Excel Components

To create COM objects or Excel add-ins, you use the MATLAB Builder for COM or MATLAB Builder for Excel products, respectively, along with the MATLAB Compiler.

MATLAB Builder for COM

To use the MATLAB Compiler to build COM objects, you must have MATLAB Builder for COM installed on your development machine. MATLAB Builder for COM lets you convert MATLAB algorithms to Common Object Model (COM) objects that are accessible from Visual Basic, C/C++, Microsoft Excel, or any other COM-based application.

For more information on MATLAB Builder for COM, see the MathWorks Web site (`http://www.mathworks.com/products/combuilder/`).

MATLAB Builder for Excel

To use the MATLAB Compiler to build Excel add-ins, you must have MATLAB Builder for Excel installed on your development machine. MATLAB Builder for Excel lets you convert MATLAB algorithms into independent Excel add-ins. MATLAB Builder for Excel generates a Visual Basic Application file (`.bas`) from your MATLAB model that you can import into Excel as a stand-alone function. Users can then call or use their MATLAB based algorithms the same way as other Excel add-ins.

For more information on MATLAB Builder for Excel, see the MathWorks Web site (`http://www.mathworks.com/products/matlabxl/`).

COM Object Target

Note To create COM components using the MATLAB Compiler, you must have MATLAB Builder for COM installed on your system. For more information, see the MATLAB Builder for COM documentation.

With the optional MATLAB Builder for COM product, you can create COM components that can be used in any application that works with COM objects.

You can use the MATLAB Compiler to create Component Object Model (COM) objects from MATLAB M-files. The collection of M-files is translated into a single COM class. MATLAB Builder for COM supports multiple classes per component.

COM Component Wrapper

The interface to the COM class is the same set of functions that are exported from a C shared library, but the MATLAB Compiler supports both C and C++ code generation in producing COM objects.

`mcc` automatically

- Invokes the Microsoft Interface Definition Language (MIDL) Compiler
- Invokes the resource compiler
- Specifies the `.DEF` files

Using `mcc` options you can enable auto registration of the COM-compatible DLL.

Note MATLAB Builder for COM is available on Windows only. The only compilers that support the building of COM objects with the MATLAB Compiler are Borland C++Builder (versions 3.0, 4.0, and 5.0) and Microsoft Visual C/C++ (versions 6.0, 7.0, and 7.1). The Borland C++Builder products require you to have the MIDL Compiler provided by Microsoft to create COM objects.

For example, to compile `plus1.m` into a COM object, use

```
mcc -B 'ccom:addin,addin,1.0' plus1.m
```

COM Components

The COM wrapper file allows you to create COM components from MATLAB M-files. The Compiler options that generate the COM wrappers are

```
-W com:<component_name>[,<class_name>[,<major>.<minor>]]
-W excel:<component_name>[,<class_name>[,<major>.<minor>]]
```

The COM wrapper options create a superset of the files created when producing a C or C++ library wrapper. In addition to the C or C++ library files, the COM wrapper creates the files shown in the following table.

File	Description
`<component_name>_idl.idl`	Interface description file for COM
`<component_name>_com.hpp`	C++ header file for the COM class
`<component_name>_com.cpp`	C++ source file for the COM class
`<component_name>_dll.cpp`	DLL interface for the COM object
`<component_name>.def`	Definition file for the COM DLL
`<component_name>.rc`	Resource file for the COM DLL

If the `<class_name>` is not specified, it defaults to `<component_name>`. If the version number is not specified, it defaults to the latest version built or 1.0, if there is no previous version.

The COM wrapper option generates all the required code and files to create a single COM object, which contains all of the compiler-generated interfaces. It creates a single COM class with the same name as the specified `<class_name>` and a corresponding interface class called `I<class_name>`. It uses the major and minor version numbers to control the major and minor version numbers of the generated COM interface.

MATLAB Builder for COM generates a COM interface using C++ rather than C. This is a requirement of COM and not particular to the MATLAB Compiler.

The MATLAB Builder for COM automatically adds all the generated C++ source files to the `mcc` command line. The details of how the new file types (`.def`, `.rc`, and `.idl`) are processed are specified in "How mbuild Processes the File Types" on page 7-5.

If the major and minor version numbers are specified, the Compiler replaces any existing type library with the specified new version number. If no version numbers are specified and there is an existing type library, the Compiler replaces the current version.

The MATLAB Compiler derives the name of the generated COM component DLL from the component name and version numbers: `<component_name>_<major>_<minor>.dll`. This prevents new versions from conflicting with each other. The user never uses the DLL name. It is not necessary to specify this name to the system because COM locates component DLLs using the Window's registry.

For details on how the MATLAB Compiler processes the `ccom` bundle file, see "Using Bundle Files" on page 4-7.

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

How mbuild Processes the File Types

The `mbuild` option, `-regsvr`, uses the `mwregsvr32` program to register the resulting shared library at the end of compilation. The Compiler uses this option whenever it produces a COM wrapper file.

<filename>.idl. You can specify IDL source files on the `mbuild` command line. These files are compiled using the MIDL Compiler. The compiler adds any generated `.idl` files to the `mbuild` command line.

<filename>.def. You can specify DEF files on the `mbuild` command line to indicate the symbols exported from a given shared library. It is an error to have more than one `.def` file specified on the command line.

<filename>.rc. You can specify an RC file on the `mbuild` command line and it is added into the DLL as required. It is an error to have more than one `.rc` file specified on the command line.

COM Signature

For each M-file specified on the `mcc` command line, MATLAB Builder for COM generates a COM-compatible interface function and a generic C interface function.

M-Function Signature.

```
[Y1, Y2,  , Varargout] = f(X1, X2,  , Varargin)
```

C Signature.

```
void mlxF(int nlhs, mxArray* plhs[],
    int nrhs, const mxArray* prhs[]);

   mxArray *mlfNF( int nargout,
                   mxArray ** y1,
                   mxArray **y2,
                   .
                   .
                   mxArray *x1,
                   mxArray *x2,
                   .
                   .
                    ... );
```

COM/IDL Signature.

```
HRESULT f([in] long nargout,
          [in,out] VARIANT* Y1,
          [in,out] VARIANT* Y2,
          .
          .
          [in,out] VARIANT* varargout,
          [in] VARIANT X1,
          [in] VARIANT X2,
          .
          .
          [in] VARIANT varargin);
```

The COM run-time performs all of the conversion between the COM types and MATLAB arrays. For details on this conversion, see the MATLAB Builder for Excel or MATLAB Builder for COM documentation.

Excel Plug-In Target

Note To create Excel plug-ins using the MATLAB Compiler, you must have MATLAB Builder for Excel installed on your system. For more information, see the MATLAB Builder for Excel documentation.

With the optional MATLAB Builder for Excel product, you can automatically generate a Visual Basic Application file (`.bas`) and a plug-in DLL from your MATLAB based application that can be imported into Excel as a stand-alone function.

You can use `mcc` to create a COM object from MATLAB M-files that can be used as an Excel plug-in. The collection of M-files is translated into a single Excel plug-in. MATLAB Builder for Excel supports one class per component.

Excel Plug-in Wrapper

The interface to the COM class is the same set of functions that are exported from a C shared library, but the MATLAB Compiler supports both C and C++ code generation in producing COM objects.

`mcc` automatically

- Invokes the Microsoft Interface Definition Language (MIDL) Compiler
- Invokes the resource compiler
- Specifies the `.DEF` files

Using `mcc` options you can enable auto registration of the COM-compatible DLL.

Note MATLAB Builder for Excel is available on Windows only. The only compilers that support the building of Excel plug-ins with the MATLAB Compiler are Borland C++Builder (versions 3.0, 4.0, and 5.0) and Microsoft Visual C/C++ (versions 6.0, 7.0, and 7.1). The Borland C++Builder products require you to have the MIDL Compiler provided by Microsoft to create COM objects.

For example, to compile `plus1.m` into an Excel plug-in, use

```
mcc -B 'cexcel:addin,addin,1.0' plus1.m
```

The COM class generated by MATLAB Builder for Excel has the same structure and obeys the same rules as the COM classes generated by MATLAB Builder for COM. For more details, see “COM Object Target” on page 7-3 and the MATLAB Builder for Excel documentation.

8

Reference

Functions — Categorical List

Pragmas

`%#external`	Pragma to call arbitrary C/C++ functions from your M-code.
`%#function`	`feval` pragma.

Command Line Tools

`buildmcr`	Generate the MCRInstaller archive.
`isdeployed`	Determine if code is running in deployed mode or MATLAB mode.
`mbuild`	Compile and link source files into a stand-alone executable or shared library.
`mcc`	Invoke MATLAB Compiler.

Purpose

Pragma to call arbitrary C/C++ functions from your M-code

Syntax

```
%#external
```

Description

The `%#external` pragma informs the MATLAB Compiler that the implementation version of the function (`Mlx`*f*) will be hand written and will not be generated from the M-code. This pragma affects only the single function in which it appears, and any M-function may contain this pragma (local, global, private, or method).

When using this pragma, the Compiler will generate an additional header file called `fcn_external.h`, where `fcn` is the name of the initial M-function containing the `%#external` pragma. This header file will contain the `extern` declaration of the function that the user must provide. This function must conform to the same interface as the Compiler-generated code. For more information on the `%#external` pragma, see “Interfacing M-Code to C/C++ Code” on page 4-13.

%#function

Purpose

`feval` pragma

Syntax

```
%#function <function_name-list>
```

Description

This pragma informs the MATLAB Compiler that the specified function(s) will be called through an `feval`, `eval`, or Handle Graphics callback. You need to specify this pragma only to assist the Compiler in locating and automatically compiling the set of functions when using the `-h` option.

If you are using the `%#function` pragma to define functions that are not available in M-code, you should use the `%#external` pragma to define the function. For example:

```
%#function myfunctionwritteninc
```

This implies that `myfunctionwritteninc` is an M-function that will be called using `feval`. The Compiler will look up this function to determine the correct number of input and output variables. Therefore, you need to provide a dummy M-function that contains a function line and a `%#external` pragma, such as

```
function y = myfunctionwritteninc( a, b, c );
%#external
```

The `function` statement indicates that the function takes three inputs (`a`, `b`, `c`) and returns a single output variable (`y`). No additional lines need to be present in the M-file.

Purpose

Generate MCRInstaller archive

Syntax

```
buildmcr;
filename = buildmcr;
filename = buildmcr(dirname);
filename = buildmcr(dirname,filename);
```

Description

The `buildmcr` function builds the MCRInstaller archive, `MCRInstaller.zip`, in the default directory. The archive is a ZIP file of the files required for the MCR. Directories are created as needed.

Examples

Example 1

```
buildmcr;
```

This example builds `MCRInstaller.zip` in the default directory. It returns nothing (unless it encounters an error or it is not the first time).

Example 2

```
mcr_zipfile = buildmcr;
```

This example builds `MCRInstaller.zip` in the default directory and returns the path of the generated ZIP file in `mcr_zipfile` as

```
mcr_zipfile = ...
              fullfile(matlabroot,'toolbox','compiler',
              'deploy',ARCH,'MCRInstaller.zip');
```

Example 3

```
mcr_zipfile = buildmcr(mcr_zipfile_dirname);
```

This example returns the ZIP file in

```
<mcr_zipfile_dirname>/MCRInstaller.zip
```

Example 4

```
mcr_zipfile =
buildmcr(mcr_zipfile_dirname,mcr_zipfile_filename);
```

returns the ZIP file in

```
<mcr_zipfile_dirname>/<mcr_zipfile_filename>
```

Example 5

```
mcr_zipfile = buildmcr('.');
```

This example builds `MCRInstaller.zip` in the current directory and returns

```
mcr_zipfile = fullfile(pwd,'MCRInstaller.zip')
```

Example 6

```
mcr_zipfile = buildmcr('.','my.zip');
```

This example builds `my.zip` in the current directory and returns

```
mcr_zipfile = fullfile(pwd,'my.zip')
```

Example 7

```
[mcr_zipfile,mcrlist] = buildmcr(...)
```

This example returns the list of the files in the ZIP file in `mcrlist`. It is a cell array of paths each relative to the MATLAB root directory. If the ZIP file already exists, nothing is done and a warning is produced. The required list is constructed from the installer files and pruned appropriately.

Purpose

Test if code is running in deployed mode or MATLAB mode

Syntax

```
x = isdeployed
```

Description

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

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

mbuild

Purpose

Compile and link source files into a stand-alone executable or shared library

Syntax

```
mbuild [option1 ... optionN] sourcefile1 [... sourcefileN]
       [objectfile1 ... objectfileN] [libraryfile1 ... libraryfileN]
        [exportfile1 ... exportfileN]
```

Note Supported types of source files are: `.c`, `.cpp`, `.idl`, `.rc`. To specify IDL source files to be compiled with the Microsoft Interface Definition Language (MIDL) Compiler, add `<filename>.idl` to the `mbuild` command line. To specify a DEF file, add `<filename>.def` to the command line. To specify an RC file, add `<filename>.rc` to the command line. Source files that are not one of the supported types are passed to the linker.

Description

`mbuild` is a script that supports various options that allow you to customize the building and linking of your code. Table 8-1, mbuild Options, lists the set of `mbuild` options. If no platform is listed, the option is available on both UNIX and Windows.

Table 8-1: mbuild Options

Option	Description
`-<arch>`	(UNIX) Assume local host has architecture `<arch>`. Possible values for `<arch>` include `sol2`, `hpux`, and `glnx86`.
`@<response_file>`	(Windows) Replace `@<response_file>` on the `mbuild` command line with the contents of the text file, `response_file`.
`-c`	Compile only. Do not link. Creates an object file but not an executable.
`-D<name>`	Define a symbol name to the C/C++ preprocessor. Equivalent to a `#define <name>` directive in the source.

Table 8-1: mbuild Options (Continued)

Option	Description
`-D<name>#<value>`	Define a symbol name and value to the C/C++ preprocessor. Equivalent to a `#define <name> <value>` directive in the source.
`-D<name>=<value>`	(UNIX) 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 a debuggable executable. If this option is specified, `mbuild` appends the value of options file variables ending in `DEBUGFLAGS` with their corresponding base variable. This option also disables the `mbuild` default behavior of optimizing built object code.
`-h[elp]`	Help; prints a description of `mbuild` and the list of options.
`-I<pathname>`	Add `<pathname>` to the list of directories to search for `#include` files.
`-inline`	Inline matrix accessor functions (`mx*`).
`-l<name>`	(UNIX) Link with object library `lib<name>`.
`-L<directory>`	(UNIX) Add `<directory>` to the list of directories containing object-library routines.

Table 8-1: mbuild Options (Continued)

Option	Description
`-lang <language>`	Specify compiler language. `<language>` can be `c` or `cpp`. By default, `mbuild` determines which compiler (C or C++) to use by inspection of the source file's extension. This option overrides that mechanism. This option is necessary when you use an unsupported file extension, or when you pass in all `.o` files and libraries.
`-n`	No execute mode. Print out any commands that `mbuild` would execute, but do not actually execute any of them.
`-O`	Optimize the object code by including the optimization flags listed in the options file. If this option is specified, `mbuild` appends the value of options file variables ending in `OPTIMFLAGS` with their corresponding base variable. Note that optimizations are enabled by default, are disabled by the `-g` option, but are reenabled by `-O`.
`-outdir <dirname>`	Place any generated object, resource, or executable files in the directory `<dirname>`. Do not combine this option with `-output` if the `-output` option gives a full pathname.
`-output <resultname>`	Create an executable named `<resultname>`. An appropriate executable extension is automatically appended. Overrides the `mbuild` default executable naming mechanism.
`-regsvr`	(Windows) Use the `regsvr32` program to register the resulting shared library at the end of compilation. The Compiler uses this option whenever it produces a COM wrapper file.

Table 8-1: mbuild Options (Continued)

Option	Description
`-setup`	Interactively specify the compiler options file to use as default for future invocations of `mbuild` by placing it in `<UserProfile>\Application Data\MathWorks\MATLAB\R14` (Windows) or `$HOME/.matlab/R14` (UNIX). When this option is specified, no other command line input is accepted.
`-U<name>`	Remove any initial definition of the C preprocessor symbol `<name>`. (Inverse of the `-D` option.)
`-v`	Verbose; Print the values for important internal variables after the options file is processed and all command line arguments are considered. Prints each compile step and final link step fully evaluated to see which options and files were used. Very useful for debugging.
`<name>=<value>`	(UNIX) Override an options file variable for variable `<name>`. If `<value>` contains spaces, enclose it in single quotes, e.g., `CFLAGS='opt1 opt2'`. The definition, `<def>`, can reference other variables defined in the options file. To reference a variable in the options file, prepend the variable name with a `$`, e.g., `CFLAGS='$CFLAGS opt2'`.
`<name>#<value>`	Override an options file variable for variable `<name>`. If `<def>` contains spaces, enclose it in single quotes, e.g., `CFLAGS='opt1 opt2'`. The definition, `<def>`, can reference other variables defined in the options file. To reference a variable in the options file, prepend the variable name with a `$`, e.g., `CFLAGS='$CFLAGS opt2'`.

Note 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.

Purpose

Invoke MATLAB Compiler

Syntax

```
mcc [-options] mfile1       [mfile2 ... mfileN]
                            [C/C++file1 ... C/C++fileN]
```

Description

`mcc` is the MATLAB command that invokes the MATLAB Compiler. You can issue the `mcc` command either from the MATLAB command prompt (MATLAB mode) or the DOS or UNIX command line (stand-alone mode).

Prepares M-file(s) for deployment outside of the MATLAB environment. Generates wrapper files in C or C++, and optionally builds stand-alone binary files. Writes any resulting files into the current directory, by default.

If more than one M-file is specified on the command line, the Compiler generates a C or C++ function for each M-file. If C or object files are specified, they are passed to `mbuild` along with any generated C files.

Options

-a Add to Archive. -Add a file to the CTF archive. Use

```
-a filename
```

to specify a file to be directly added to the CTF archive. Multiple `-a` options are permitted. The Compiler looks for these files on the MATLAB path, so specifying the full pathname is optional. These files are not passed to `mbuild`, so you can include files such as data files.

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

-B Specify bundle File. Replace the file on the `mcc` command line with the contents of the specified file. Use

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

The bundle file `filename` should contain only `mcc` command line options and corresponding arguments and/or other filenames. The file may contain other `-B` options. A bundle file can include replacement parameters for Compiler options

that accept names and version numbers. See “Using Bundle Files” on page 4-7 for a list of the bundle files included with the Compiler.

-c Generate C Code Only. When used with a macro option, generate C code but do not invoke `mbuild`, i.e., do not produce a stand-alone application. This option is equivalent to `-T codegen` placed at the end of the `mcc` command line.

-d Specified Directory for Output. Place output in a specified directory. Use

```
-d directory
```

to direct the output files from the compilation to the directory specified by the `-d` option.

-f Specified Options File. Override the default options file with the specified options file. Use

```
-f filename
```

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

Note It is recommended that you use `mbuild -setup`.

-g Generate Debugging Information. Include debugging symbol information for the wrapper.

-G Debug Only. Causes `mbuild` to invoke the C/C++ compiler with the appropriate C/C++ compiler options for debugging. You should specify `-G` if you want to debug the stand-alone application with a debugger.

-I Add Directory to Path. Add a new directory path to the list of included directories. Each `-I` option adds a directory to the *end* of the current search path. For example,

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

would set up the search path so that `directory1` is searched first for M-files, followed by `directory2`. This option is important for stand-alone compilation where the MATLAB path is not available.

-l Generate a Function Library. Macro to create a function library. This option generates a library wrapper function for each M-file on the command line and calls your C compiler to build a shared library, which exports these functions. The library name is the component name, which is derived from the name of the first M-file on the command line. This macro is equivalent to

```
-W lib -T link:lib
```

-m Generate a Stand-Alone Application. Macro to produce a stand-alone application. This macro is equivalent to

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

-M Direct Pass Through. Define compile-time options. Use

```
-M string
```

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

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

-N Clear Path. Passing `-N` effectively clears the path of all directories except the following core directories (this list is subject to change over time):

- `<matlabroot>/toolbox/matlab`
- `<matlabroot>/toolbox/local`
- `<matlabroot>/toolbox/compiler`

It also retains all subdirectories of the above list that appear on the MATLAB path at compile time. Including `-N` on the command line also allows you to replace directories from the original path, while retaining the relative ordering of the included directories. All subdirectories of the included directories that appear on the original path are also included.

-o Specify Executable. Specify the name and location of the final executable (stand-alone applications only). Use

```
-o outputfile
```

to name the final executable output of the Compiler. A suitable, possibly platform-dependent, extension is added to the specified name (e.g., `.exe` for Windows stand-alone applications).

-p Add Directory to Path. Add a directory to the compilation path in an order-sensitive context, i.e., the same order in which they are found on your MATLAB path.

```
-p directory
```

where `directory` is the directory to be included. If `directory` is not an absolute path, it is assumed to be under the current working directory. The rules for how these directories are included are

- If a directory is included with `-p` that is on the original MATLAB path, the directory and all its subdirectories that appear on the original path are added to the compilation path in an order-sensitive context.
- If a directory is included with `-p` that is not on the original MATLAB path, that directory is not included in the compilation. (You can use `-I` to add it.)

If a path is added with the `-I` option while this feature is active (`-N` has been passed) and it is already on the MATLAB path, it is added in the order-sensitive context as if it were included with `-p`. Otherwise, the directory is added to the head of the path, as it normally would be with `-I`.

Note Requires `-N` option.

-R Run-Time. Provide MCR run-time options. Use the syntax

```
-R option
```

to provide any of these run-time options.

Option	Description
`-nojvm`	Do not use the Java Virtual Machine (JVM).
`-nojit`	Do not use the MATLAB JIT (binary code generation used to accelerate M-file execution).

Note The `-R` option is only available for stand-alone applications. To override MCR options in the other MATLAB Compiler targets, use the `mclInitializeApplication` and `mclTerminateApplication` functions. For more information on these functions, see "Calling a Shared Library" on page 6-9.

-T Specify Target Stage. Specify the output target phase and type. Use the syntax

```
-T target
```

to define the output type. Valid *`target`* values are as follows:

Table 8-2: Output Stage Options

Target	Description
`codegen`	Generates a C/C++ wrapper file. The default is `codegen`.
`compile:exe`	Same as `codegen` plus compiles C/C++ files to object form suitable for linking into a stand-alone executable.
`compile:lib`	Same as `codegen` plus compiles C/C++ files to object form suitable for linking into a shared library/DLL.
`link:exe`	Same as `compile:exe` plus links object files into a stand-alone executable.

Table 8-2: Output Stage Options (Continued)

Target	Description
`link:lib`	Same as `compile:lib` plus links object files into a shared library/DLL.
Notes: `exe` uses the `mbuild` script to build an executable; `lib` uses `mbuild` to build a shared library.	

-v Verbose. Display the compilation steps, including

- The Compiler version number
- The source filenames 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 Warning Messages. Displays warning messages. Use the syntax

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

to control the display of warnings. This table lists the valid syntaxes.

Table 8-3: Warning Option

Syntax	Description
`-w list`	Generates a table that maps `<string>` to warning message for use with `enable`, `disable`, and `error`. Appendix B, "Error and Warning Messages" lists the same information.
`-w enable`	Enables complete warnings.

Table 8-3: Warning Option (Continued)

Syntax	Description
`-w disable[:<string>]`	Disables specific warning associated with `<string>`. Appendix B, "Error and Warning Messages" lists the valid `<string>` values. Leave off the optional `:<string>` to apply the `disable` action to all warnings.
`-w enable[:<string>]`	Enables specific warning associated with `<string>`. Appendix B, "Error and Warning Messages" lists the valid `<string>` values. Leave off the optional `:<string>` to apply the `enable` action to all warnings.
`-w error[:<string>]`	Treats specific warning associated with `<string>` as error. Leave off the optional `:<string>` to apply the `error` action to all warnings.

-W Wrapper Function. Controls the generation of function wrappers. Use the syntax

```
-W type
```

to control the generation of function wrappers for a collection of Compiler-generated M-files. You provide a list of functions and the Compiler generates the wrapper functions and any appropriate global variable definitions. This table shows the valid options.

Table 8-4: Function Wrapper Types

Type	Description
`main`	Produces a POSIX shell `main()` function.
`lib:<string>`	Produces an initialization and termination function for use when compiling this Compiler-generated code into a larger application. This option also produces a header file containing prototypes for all public functions in all M-files specified. `<string>` becomes the base (file) name for the generated C/C++ and header file. Creates a `.exports` file that contains all nonstatic function names.
`com:<component_name>,<class_name>,<version>`	Produces a COM object from MATLAB M-files.
`none`	Does not produce a wrapper file. The default is `none`.

-Y License File. Use

```
-Y license.dat_file
```

to override default `license.dat` file with specified argument.

-z Specify Path. Specify `path` for library and include files. Use

```
-z path
```

to specify `path` to use for the compiler libraries and include files instead of the path returned by `matlabroot`.

-? Help Message. Display MATLAB Compiler help at the command prompt.

Examples

Make a stand-alone executable for `myfun.m`.

```
mcc -m myfun
```

Make a stand-alone executable for `myfun.m`. Look for `myfun.m` in the `/files/source` directory, and put the resulting C files and executable in the `/files/target` directory.

```
mcc -m -I /files/source -d /files/target myfun
```

Make a stand-alone executable from `myfun1.m` and `myfun2.m` (using one `mcc` call).

```
mcc -m myfun1 myfun2
```

Make a shared/dynamically linked library called `liba` from `a0.m` and `a1.m`.

```
mcc -W lib:liba -T link:lib a0 a1
```

A

MATLAB Compiler Quick Reference

This appendix summarizes the MATLAB Compiler options and provides brief descriptions of how to perform common tasks.

Common Uses of the Compiler

This section summarizes how to use the MATLAB Compiler to generate some of its more standard results.

Create a Stand-Alone Application

Example 1

To create a stand-alone executable for `mymfile.m`, use

```
mcc -m mymfile
```

Example 2

To create a stand-alone application for `mymfile.m`, look for `mymfile.m` in the directory `/files/source`, and put the resulting C files and executable in `/files/target`, use

```
mcc -m -I /files/source -d /files/target mymfile
```

Example 3

To create a stand-alone application from `mymfile1.m` and `mymfile2.m` using a single `mcc` call, use

```
mcc -m mymfile1 mymfile2
```

Create a Library

Example 1

To create a C shared library from `foo.m`, use

```
mcc -l foo.m
```

Example 2

To create a C shared library called `library_one` from `foo1.m` and `foo2.m`, use

```
mcc -W lib:library_one -T link:lib foo1 foo2
```

Note You can add the `-g` option to any of these for debugging purposes.

mcc

Bold entries in the Comment/Options column indicate default values.

Table A-1: mcc Quick Reference

Option	Description	Comment/Options
`a filename`	Add `filename` to the CTF archive	None
`b`	Generate Excel-compatible formula function	Requires MATLAB Builder for Excel
`B filename[:arg[,arg]]`	Replace `-B filename` on the `mcc` command line with the contents of `filename`	The file should contain only `mcc` command line options. These are MathWorks included options files: `-B csharedlib:foo` C shared library `-B cpplib:foo` C++ library
`c`	Generate C wrapper code	Equivalent to `-T codegen`
`d directory`	Place output in specified `directory`	None
`f filename`	Use the specified options file, `filename`, when calling `mbuild`	`mbuild -setup` is recommended.
`g`	Generate debugging information	None
`G`	Debug only. Simply turn debugging on, so debugging symbol information is included.	None

Table A-1: mcc Quick Reference (Continued)

Option	Description	Comment/Options
`I directory`	Add `directory` to search path for M-files	MATLAB path is automatically included when running from MATLAB, but not when running from DOS/UNIX shell.
`l`	Macro to create a function library	Equivalent to `-W lib -T link:lib`
`m`	Macro to generate a C stand-alone application	Equivalent to `-W main -T link:exe`
`M string`	Pass `string` to `mbuild`	Use to define compile-time options.
`N`	Clear the path of all but a minimal, required set of directories	None
`o outputfile`	Specify name/location of final executable	Adds appropriate extension
`P directory`	Add `directory` to compilation path in an order-sensitive context	Requires `-N` option
`R` *`option`*	Specify run-time options for MCR	*`option`* = `-nojvm` `-nojit`
`T` *`target`*	Specify output stage	*`target`* = **`codegen`** `compile:`*`bin`* `link:`*`bin`* where *`bin`* = `exe` `lib`
`v`	Verbose; Display compilation steps	None

Table A-1: mcc Quick Reference (Continued)

Option	Description	Comment/Options

`w` *option*	Display warning messages	*option* = `list`
		level
		level`:string`
		where *level* = `disable`
		`enable`
		`error`

`W` *type*	Control the generation of function wrappers	*type* = `main`
		`lib:<string>`
		`none`
		`com:compname,clname,version`

`Y licensefile`	Use `licensefile` when checking out a Compiler license	None
`z path`	Specify `path` for library and include files	None
`?`	Display help message	None

B

Error and Warning Messages

This appendix lists and describes error messages and warnings generated by the MATLAB Compiler. Compile-time messages are generated during the compile or link phase. It is useful to note that most of these compile-time error messages should not occur if MATLAB can successfully execute the corresponding M-file. Run-time messages are generated when the executable program runs.

Use this reference to

- Confirm that an error has been reported
- Determine possible causes for an error
- Determine possible ways to correct an error

When using the MATLAB Compiler, if you receive an internal error message, record the specific message and report it to Technical Support at The MathWorks at `support@mathworks.com`.

Compile-Time Errors

Error: An error occurred while shelling out to *mex/mbuild* (error code = *errorno*). Unable to build executable (specify the -v option for more information). The Compiler reports this error if `mbuild` or `mex` generates an error.

Error: An error occurred writing to file "*filename*": *reason*. The file could not be written. The reason is provided by the operating system. For example, you may not have sufficient disk space available to write the file.

Error: Cannot write file "*filename*" because MCC has already created a file with that name, or a file with that name was specified as a command line argument. The Compiler has been instructed to generate two files with the same name. For example:

```
mcc -W lib:liba liba -t % Incorrect
```

Error: Could not check out a Compiler license. No additional Compiler licenses are available for your workgroup.

Error: Could not find license file "*filename*". (*Windows only*) The `license.dat` file could not be found in `<matlabroot>\bin`.

Error: File: "*filename*" not found. A specified file could not be found on the path. Verify that the file exists and that the path includes the file's location. You can use the `-I` option to add a directory to the search path

Error: File: "*filename*" is a script M-file and cannot be compiled with the current Compiler. The MATLAB Compiler cannot compile script M-files. To learn how to convert script M-files to function M-files, see "Converting Script M-Files to Function M-Files" on page 4-17.

Error: File: *filename* Line: # Column: # A variable cannot be made *storageclass1* after being used as a *storageclass2*. You cannot change a variable's storage class (global/local/persistent). Even though MATLAB allows this type of change in scope, the Compiler does not.

Error: Found illegal whitespace character in command line option: "*string*". The strings on the left and right side of the space should be separate arguments to MCC. For example:

```
mcc('-m', '-v', 'hello')   % Correct
mcc('-m -v', 'hello')      % Incorrect
```

Error: Improper usage of option *-optionname*. Type "mcc -?" for usage information. You have incorrectly used a Compiler option. For more information about Compiler options, see Chapter 8, "Reference" or type `mcc -?` at the command prompt.

Error: *libraryname* library not found. MATLAB has been installed incorrectly.

Error: No source files were specified (-? for help). You must provide the Compiler with the name of the source file(s) to compile.

Error: "*optionname*" is not a valid *-option* option argument. You must use an argument that corresponds to the option. For example:

```
mcc -W main  % Correct
mcc -W mex  % Incorrect
```

Error: Out of memory. Typically, this message occurs because the Compiler requests a larger segment of memory from the operating system than is currently available. Adding additional memory to your system could alleviate this problem.

Error: Previous warning treated as error. When you use the `-w error` option, this error displays immediately after a warning message.

Error: The argument after the *-option* option must contain a colon. The format for this argument requires a colon. For more information, see Chapter 8, "Reference" or type `mcc -?` at the command prompt.

Error: The environment variable MATLAB must be set to the MATLAB root directory. On UNIX, the `MATLAB` and `LM_LICENSE_FILE` variables must be set. The `mcc` shell script does this automatically when it is called the first time.

Error: The license manager failed to initialize (error code is *errornumber*). You do not have a valid Compiler license or no additional Compiler licenses are available.

Error: The option *-option* is invalid in *modename* mode (specify -? for help). The specified option is not available.

Error: The specified file "*filename*" cannot be read. There is a problem with your specified file. For example, the file is not readable because there is no read permission.

Error: The *-optionname* option requires an argument (e.g. "*proper_example_usage*"). You have incorrectly used a Compiler option. For more information about Compiler options, see Chapter 8, "Reference" or type `mcc -?` at the command prompt.

Error: -x is no longer supported. The MATLAB Compiler no longer generates MEX-files because there is no longer any performance advantage to doing so. The MATLAB JIT Accelerator makes compilation for speed obsolete.

Error: Unable to open file "*filename*":<string>. There is a problem with your specified file. For example, there is no write permission to the output directory, or the disk is full.

Error: Unable to set license linger interval (error code is *errornumber*). A license manager failure has occurred. Contact Technical Support at The MathWorks with the full text of the error message.

Error: Unknown warning enable/disable string: *warningstring*. `-w enable:`, `-w disable:`, and `-w error:` require you to use one of the warning string identifiers listed in the "Warning Messages" on page B-5.

Error: Unrecognized option: *-option*. The option is not one of the valid options for this version of the Compiler. See Chapter 8, "Reference" for a complete list of valid options for MATLAB Compiler 3.0 or type `mcc -?` at the command prompt.

Warning Messages

This section lists the warning messages that the MATLAB Compiler can generate. Using the `-w` option for `mcc`, you can control which messages are displayed. Each warning message contains a description and the warning message identifier string (in parentheses) that you can enable or disable with the `-w` option. For example, to enable the display of warnings related to undefined variables, you can use

```
mcc -w enable:undefined_variable
```

To enable all warnings except those generated by the `save` command, use

```
mcc -w enable -w disable:save_options
```

To display a list of all the warning message identifier strings, use

```
mcc -w list -m mfilename
```

For additional information about the `-w` option, see Chapter 8, “Reference”.

Warning: File: *filename* Line: # Column: # The call to function "*functionname*" on this line could not be bound to a function that is known at compile time. A run-time error will occur if this code is executed. *(no_matching_function)* The called function was not found on the search path. You can add the function to the MATLAB path or use the MATLAB Compiler `-a` option to add the missing function at compile time.

Warning: File: *filename* Line: # Column: # The #function pragma expects a list of function names. *(pragma_function_missing_names)* This pragma informs the MATLAB Compiler that the specified function(s) provided in the list of function names will be called through an `feval` call. This will automatically compile the selected functions.

Warning: M-file "*filename*" was specified on the command line with full path of "*pathname*", but was found on the search path in directory "*directoryname*" first. *(specified_file_mismatch)* The Compiler detected an inconsistency between the location of the M-file as given on the command line and in the search path. The Compiler uses the location in the search path. This warning occurs when you specify a full pathname on the `mcc` command line and a file with the same base name (filename) is found earlier on the search path. This warning is issued in the following example if the file `afile.m` exists in both `dir1` and `dir2`.

```
mcc -m -I /dir1 /dir2/afile.m
```

Warning: The file *filename* was repeated on the Compiler command line. *(repeated_file)* This warning occurs when the same filename appears more than once on the compiler command line. For example:

```
mcc -m sample.m sample.m % Will generate the warning
```

Warning: The name of a shared library should begin with the letters "lib". "*libraryname*" doesn't. *(missing_lib_sentinel)* This warning is generated if the name of the specified library does not begin with the letters “lib”. This warning is specific to UNIX and does not occur on Windows. For example:

```
mcc -t -W lib:liba -T link:lib a0 a1 % No warning
mcc -t -W lib:a -T link:lib a0 a1    % Will generate a warning
```

Warning: The specified private directory is not unique. Both "*directoryname1*" and "*directoryname2*" are found on the path for this private directory. *(duplicate_private_directories)* The Compiler cannot distinguish which private function to use.

Run-Time Errors

Note The error messages described in this section are generated by the MATLAB Compiler into the code exactly as they are written, but are not the only source of run-time errors.

Run-time Error: File: *filename* Line: # Column: # The call to function "*functionname*" on this line passed *quantity1* inputs and the function is declared with *quantity2*. There is an inconsistency between the formal and actual number of inputs to a function.

Run-time Error: File: *filename* Line: # Column: # The call to function "*functionname*" on this line requested *quantity1* outputs and the function is declared with *quantity2*. There is an inconsistency between the formal and actual number of outputs from a function.

Run-time Error: File: *filename* Line: # Column: # The function "*functionname*" was called with more than the declared number of inputs (*quantity1*). There is an inconsistency between the declared number of formal inputs and the actual number of inputs.

Run-time Error: File: *filename* Line: # Column: # The function "*functionname*" was called with more than the declared number of outputs (*quantity1*). There is an inconsistency between the declared number of formal outputs and the actual number of outputs.

Depfun Errors

The MATLAB Compiler uses a dependency analysis (`Depfun`) to determine the list of necessary files to include in the CTF package. If this analysis encounters a problem, `Depfun` displays an error.

These error messages take the form

```
Depfun Error: <message>
```

There are three causes of these messages:

- MCR/Dispatcher errors
- XML parser errors
- `Depfun`-produced errors

MCR/Dispatcher Errors

These errors originate directly from the MCR/dispatcher. If one of these error occurs, report it to Technical Support at The MathWorks at `support@mathworks.com`.

XML Parser Errors

These errors appear as

```
Depfun Error: XML error: <message>
```

Where `<message>` is a message returned by the XML parser. If this error occurs, report it to Technical Support at The MathWorks at `support@mathworks.com`.

Depfun-Produced Errors

These errors originate directly from `depfun`.

Depfun Error: Internal error. This error occurs if an internal error is encountered that is unexpected, for example, a memory allocation error or a system error of some kind. This error is never user generated. If this error occurs, report it to Technical Support at The MathWorks at `support@mathworks.com`.

Depfun Error: Unexpected error thrown. This error is similar to the previous one. If this error occurs, report it to Technical Support at The MathWorks at `support@mathworks.com`.

Depfun Error: Invalid file name: <filename>. An invalid filename was passed to `Depfun`.

Depfun Error: Invalid directory: <dirname>. An invalid directory was passed to `Depfun`.

C

Troubleshooting

This appendix identifies some of the more common problems that may occur when using the MATLAB Compiler.

mbuild

This section identifies some of the more common problems that might occur when configuring `mbuild` to create stand-alone applications.

Options File Not Writeable. When you run `mbuild -setup`, `mbuild` makes a copy of the appropriate options file and writes some information to it. If the options file is not writeable, you are asked if you want to overwrite the existing options file. If you choose to do so, the existing options file is copied to a new location and a new options file is created.

Directory or File Not Writeable. If a destination directory or file is not writeable, ensure that the permissions are properly set. In certain cases, make sure that the file is not in use.

mbuild Generates Errors. If you run `mbuild filename` and get errors, it may be because you are not using the proper options file. Run `mbuild -setup` to ensure proper compiler and linker settings.

Compiler and/or Linker Not Found. On Windows, if you get errors such as `unrecognized command` or `file not found`, make sure the command line tools are installed and the path and other environment variables are set correctly in the options file. For MS Visual Studio, for example, make sure to run `vcvars32.bat` (MSVC 6.x and earlier) or `vsvars32.bat` (MSVC 7.x).

mbuild Not a Recognized Command. If `mbuild` is not recognized, verify that `<matlabroot>\bin` is on your path. On UNIX, it may be necessary to rehash.

mbuild Works from Shell but Not from MATLAB (UNIX). If the command

```
mcc -m hello
```

works from the UNIX command prompt but does not work from the MATLAB prompt, you may have a problem with your `.cshrc` file. When MATLAB launches a new C shell to perform compilations, it executes the `.cshrc` script. If this script causes unexpected changes to the `PATH` environment variable, an error may occur. You can test this by performing a

```
set SHELL=/bin/sh
```

prior to launching MATLAB. If this works correctly, then you should check your `.cshrc` file for problems setting the `PATH` environment variable.

Cannot Locate Your Compiler (Windows). If `mbuild` has difficulty locating your installed compilers, it is useful to know how it goes about finding compilers. `mbuild` automatically detects your installed compilers by first searching for locations specified in the following environment variables:

- `BORLAND` for Borland C/C++, Version 5.3
- `MSVCDIR` for Microsoft Visual C/C++, Version 6.0, 7.0, or 7.1

Next, `mbuild` searches the Windows registry for compiler entries.

Internal Error When Using mbuild -setup (Windows). Some antivirus software packages such as Cheyenne AntiVirus and Dr. Solomon may conflict with the `mbuild -setup` process. If you get an error message during `mbuild -setup` of the following form

```
mex.bat: internal error in sub get_compiler_info(): don't
recognize <string>
```

then you need to disable your antivirus software temporarily and rerun `mbuild -setup`. After you have successfully run the `setup` option, you can reenable your antivirus software.

Verification of mbuild Fails. If none of the previous solutions addresses your difficulty with `mbuild`, contact Technical Support at The MathWorks at `support@mathworks.com`.

MATLAB Compiler

Typically, problems that occur when building stand-alone C and C++ applications involve `mbuild`. However, it is possible that you may run into some difficulty with the MATLAB Compiler. One problem that might occur when you try to generate a stand-alone application involves licensing.

Licensing Problem. If you do not have a valid license for the MATLAB Compiler, you will get an error message similar to the following when you try to access the Compiler.

```
Error: Could not check out a Compiler License:
No such feature exists.
```

If you have a licensing problem, contact The MathWorks. A list of contacts at The MathWorks is provided at the beginning of this manual.

MATLAB Compiler Does Not Generate Application. If you experience other problems with the MATLAB Compiler, contact Technical Support at The MathWorks at `support@mathworks.com`.

Missing Functions In Callbacks. If your application includes a call to a function in a callback string or in a string passed as an argument to the `feval` function or an ODE solver, and this is the only place in your M-file this function is called, the Compiler will not compile the function. The Compiler does not look in these text strings for the names of functions to compile. See “Fixing Callback Problems: Missing Functions” on page 1-22 for more information.

Borland Compiler Does Not Work with the Builder Products. The only compilers that support the building of COM objects are Borland C++Builder (versions 3.0, 4.0, 5.0, and 6.0) and Microsoft Visual C/C++ (versions 6.0, 7.0, and 7.1). The Borland C++Builder products require you to have the MIDL compiler provided by Microsoft to create COM objects.

D

C++ Utility Library Reference

This appendix describes the C++ utility library provided with the MATLAB Compiler.

Primitive Types

The `mwArray` API supports all primitive types that can be stored in a MATLAB array. This table lists all the types.

Type	Description	mxClassID
`mxChar`	Character type	`mxCHAR_CLASS`
`mxLogical`	Logical or Boolean type	`mxLOGICAL_CLASS`
`mxDouble`	Double-precision floating-point type	`mxDOUBLE_CLASS`
`mxSingle`	Single-precision floating-point type	`mxSINGLE_CLASS`
`mxInt8`	1-byte signed integer	`mxINT8_CLASS`
`mxUint8`	1-byte unsigned integer	`mxUINT8_CLASS`
`mxInt16`	2-byte singed integer	`mxUINT16_CLASS`
`mxUint16`	2-byte unsigned integer	`mxUINT16_CLASS`
`mxInt32`	4-byte signed integer	`mxINT32_CLASS`
`mxUint32`	4-byte unsigned integer	`mxUINT32_CLASS`
`mxInt64`	8-byte signed integer	`mxINT64_CLASS`
`mxUint64`	8-byte unsigned integer	`mxUINT64_CLASS`

Utility Classes

mwString Class

The `mwString` class is a simple string class used by the `mwArray` API to pass string data as output from some certain methods.

Constructors

Methods

Operators

mwString()

Construct an empty string

C++ syntax

```
#include "mclcppclass.h"
mwString str;
```

Arguments

None

Return value

None

Description

Use this constructor to create an empty string.

mwString(const char* str)

Construct a new string and initialize the strings data with the supplied char buffer

C++ syntax

```
#include "mclcppclass.h"
mwString str("This is a string");
```

Arguments

str. NULL-terminated char buffer to initialize the string.

Return value

None

Description

Use this constructor to create a string from a NULL-terminated char buffer.

mwString(const mwString& str)

Copy constructor for `mwString`

Constructs a new string and initialize its data with the supplied `mwString`.

C++ syntax

```
#include "mclcppclass.h"
mwString str("This is a string");
mwString new_str(str);     // new_str contains a copy of the
                           // characters in str.
```

Arguments

str. `mwString` to be copied

Return value

None

Description

Use this constructor to create an `mwString` that is a copy of an existing one.

int Length() const

Return the number of characters in the string

C++ syntax

```
#include "mclcppclass.h"
mwString str("This is a string");
int len = str.Length();    // len should be 16.
```

Arguments

None

Return value

The number of characters in the string.

Description

Use this method to get the length of an `mwString`. The value returned does not include the terminating NULL character.

operator const char* () const

Return a pointer to the internal buffer of the string

C++ syntax

```
#include "mclcppclass.h"
mwString str("This is a string");
const char* pstr = (const char*)str;
```

Arguments

None

Return value

A pointer to the internal buffer of the string.

Description

Use this operator to get direct read-only access to the string's data buffer.

mwString& operator=(const mwString& str)

`mwString` assignment operator

C++ syntax

```
#include "mclcppclass.h"
mwString str("This is a string");
mwString new_str = str;     // new_str contains a copy of the data
                            // in str.
```

Arguments

str. String to make a copy of.

Return value

A reference to the invoking `mwString` object.

Description

Use this operator to copy the contents of one string into another.

mwString& operator=(const char* str)

`mwString` assignment operator

C++ syntax

```
#include "mclcppclass.h"
const char* pstr = "This is a string";
mwString str = pstr;   // str contains a copy of the data in pstr.
```

Arguments

str. `char` buffer to make copy of.

Return value

A reference to the invoking `mwString` object.

Description

Use this operator to copy the contents of a NULL-terminated

buffer into an `mwString`.

bool operator==(const mwString& str) const

Test two `mwStrings` for equality

C++ syntax

```
#include "mclcppclass.h"
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str == str2);  // ret should have a value of false.
```

Arguments

str. String to compare.

Return value

The result of the comparison.

Description

Use this operator to test two strings for equality.

bool operator!=(const mwString& str) const

Test two `mwStrings` for inequality

C++ syntax

```
#include "mclcppclass.h"
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str != str2);      // ret should have a value of
                               // true.
```

Arguments

str. String to compare.

Return value

The result of the comparison.

Description

Use this operator to test two strings for inequality.

bool operator<(const mwString& str) const

Compare the input string with this string and return `true` if this string is lexicographically less than the input string

C++ syntax

```
#include "mclcppclass.h"
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str < str2);       // ret should have a value of
                               // true.
```

Arguments

str. String to compare.

Return value

The result of the comparison.

Description

Use this operator to test two strings for order.

bool operator<=(const mwString& str) const

Compare the input string with this string and return `true` if this string is lexicographically less than or equal to the input string

C++ syntax

```
#include "mclcppclass.h"
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str <= str2);      // ret should have a value of
                               // true.
```

Arguments

str. String to compare.

Return value

The result of the comparison.

Description

Use this operator to test two strings for order.

bool operator>(const mwString& str) const

Compare the input string with this string and return `true` if this string is lexicographically greater than the input string

C++ syntax

```
#include "mclcppclass.h"
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str > str2);       // ret should have a value of
                               // false.
```

Arguments

str. String to compare.

Return value

The result of the comparison.

Description

Use this operator to test two strings for order.

bool operator>=(const mwString& str) const

Compare the input string with this string and return `true` if this string is lexicographically greater than or equal to the input string

C++ syntax

```
#include "mclcppclass.h"
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str >= str2);// ret should have a value of false.
```

Arguments

str. String to compare.

Return value

The result of the comparison.

Description

Use this operator to test two strings for order.

friend std::ostream& operator<<(std::ostream& os, const mwString& str)

Copy the contents of the input string to the specified `ostream`

C++ syntax

```
#include "mclcppclass.h"
#include <ostream>
mwString str("This is a string");
std::cout << str << std::endl;     // should print "This is a
                                   // string" to standard out.
```

Arguments

os. `ostream` to copy string to.

str. String to copy.

Return value

The input `ostream`.

Description

Use this operator to print the contents of an `mwString` to an `ostream`.

mwException Class

The `mwException` class is the basic exception type used by the `mwArray` API and the C++ interface functions. All errors created during calls to the `mwArray` API and to MATLAB Compiler-generated C++ interface functions are thrown as `mwExceptions`.

Constructors

Methods

Operators

mwException()

Construct a new `mwException` with the default error message

C++ syntax

```
#include "mclcppclass.h"
throw mwException();
```

Arguments

None

Return value

None

Description

Use this constructor to create an `mwException` without specifying an error message.

mwException(const char* msg)

Construct a new `mwException` with a specified error message

C++ syntax

```
#include "mclcppclass.h"
try
{
   throw mwException("This is an error");
}
catch (const mwException& e)
{
       std::cout << e.what() << std::endl; // Displays "This is
                                           // an error" to
                                           // standard out.
}
```

Arguments

msg. Error message.

Return value

None

Description

Use this constructor to create an `mwException` with a specified error message.

mwException(const mwException& e)

Copy constructor for `mwException` class

C++ syntax

```
#include "mclcppclass.h"
try
{
       throw mwException("This is an error");
}
catch (const mwException& e)
{
       throw mwException(e);       // Rethrows same error.
}
```

Arguments

e. `mwException` to create copy of.

Return value

None

Description

Use this constructor to create a copy of an `mwException`. The copy will have the same error message as the original.

mwException(const std::exception& e)

Create a new `mwException` from an existing `std::exception`

C++ syntax

```
#include "mclcppclass.h"
try
{
    .
}
catch (const std::exception& e)
{
   throw mwException(e);   // Rethrows same error.
}
```

Arguments

e. `std::exception` to create copy of.

Return value

None

Description

Use this constructor to create a new `mwException` and initialize the error message with the error message from the given `std::exception`.

const char *what() const throw()

Return the error message contained in this exception

C++ syntax

```
#include "mclcppclass.h"
try
{
    .
}
catch (const std::exception& e)
{
   std::cout << e.what() << std::endl; // Displays the error
                                       // message to
                                       // standard out.
}
```

Arguments

None

Return value

A pointer to a NULL-terminated character buffer containing the error message.

Description

Use this method to retrieve the error message from an `mwException`.

mwException& operator=(const mwException& e)

Assignment operator for `mwException` class

C++ syntax

```
#include "mclcppclass.h"
try
{

}
catch (const mwException& e)
{
   mwException e2 = e;
   throw e2;
}
```

Arguments

e. `mwException` to create copy of.

Return value

A reference to the invoking `mwException`.

Description

Use this operator to create a copy of an `mwException`. The copy will have the same error message as the original.

mwException& operator=(const std::exception& e)

Assignment operator for `mwException` class

C++ syntax

```
#include "mclcppclass.h"
try
{

}
catch (const std::exception& e)
{
   mwException e2 = e;
   throw e2;
}
```

Arguments

e. `std::exception` to initialize copy with.

Return value

A reference to the invoking `mwException`.

Description

Use this operator to create a copy of an `std::exception`. The copy will have the same error message as the original.

mwArray Class

Use the `mwArray` class to pass input/output arguments to MATLAB Compiler-generated C++ interface functions. This class consists of a thin wrapper around a MATLAB array. The `mwArray` class provides the necessary constructors, methods, and operators for array creation and initialization, as well as simple indexing.

Constructors

Methods

Operators

Static Methods

mwArray()

Construct an empty array of type `mxUNKNOWN_CLASS`

C++ syntax

```
#include "mclcppclass.h"
mwArray a;
```

Return value

None

Description

Use this constructor to create an empty array of unknown type.

mwArray(mxClassID mxID)

Construct an empty array of a specified type

C++ syntax

```
#include "mclcppclass.h"
mwArray a(mxDOUBLE_CLASS);
```

Return value

None

Description

Use this constructor to create an empty array of the specified type. Any valid `mxClassID` can be used. See the External Interfaces documentation for more information on `mxClassID`.

mwArray(int num_rows, int num_cols, mxClassID mxID, mxComplexity cmplx = mxREAL)

Construct a matrix of the specified type and dimensions

For numeric types, the matrix can be either real or complex.

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray b(3, 3, mxSINGLE_CLASS, mxCOMPLEX);
mwArray c(2, 3, mxCELL_CLASS);
```

Arguments

num_rows. The number of rows

num_cols. The number of columns

mxID. The datatype `type` of the matrix

cmplx. The complexity of the matrix (numeric types only)

Return value

None

Description

Use this constructor to create a matrix of the specified type and complexity. You can use any valid `mxClassID`. Consult the External Interfaces documentation for more information on `mxClassID`. For numeric types, pass `mxCOMPLEX` for the last argument to create a complex matrix. All elements are initialized to zero. For cell matrices, all elements are initialized to empty cells.

mwArray(int num_dims, const int* dims, mxClassID mxID, mxComplexity cmplx = mxREAL)

Construct an n-dimensional array of the specified type and dimensions

For numeric types, the array can be either real or complex.

C++ syntax

```
#include "mclcppclass.h"
int dims[3] = {2,3,4};
mwArray a(3, dims, mxDOUBLE_CLASS);
mwArray b(3, dims, mxSINGLE_CLASS, mxCOMPLEX);
mwArray c(3, dims, mxCELL_CLASS);
```

Arguments

num_dims. Size of the `dims` array

dims. Dimensions of the array

mxID. The datatype `type` of the matrix.

cmplx. The complexity of the matrix (numeric types only)

Return value

None

Description

Use this constructor to create an n-dimensional array of the specified type and complexity. You can use any valid `mxClassID`. Consult the External Interfaces documentation for more information on `mxClassID`. For numeric types, pass `mxCOMPLEX` for the last argument to create a complex matrix. All elements are initialized to zero. For cell arrays, all elements are initialized to empty cells.

mwArray(const char* str)

Construct a character array from the supplied string

C++ syntax

```
#include "mclcppclass.h"
mwArray a("This is a string");
```

Arguments

str. NULL-terminated string

Return value

None

Description

Use this constructor to create a 1-by-n array of type `mxCHAR_CLASS`, with `n = strlen(str)`, and initialize the array's data with the characters in the supplied string.

mwArray(int num_strings, const char** str)

Construct a character matrix from a list of strings

C++ syntax

```
#include "mclcppclass.h"
const char** str = {"String1", "String2", "String3"};
mwArray a(3, str);
```

Arguments

num_strings. Number of strings in the input array

str. Array of NULL-terminated strings

Return value

None

Description

Use this constructor to create a matrix of type `mxCHAR_CLASS`, and initialize the array's data with the characters in the supplied strings. The created array has dimensions `m`-by-`max`, where `max` is the length of the longest string in `str`.

mwArray(int num_rows, int num_cols, int num_fields, const char** fieldnames)

Construct a `struct` matrix of the specified dimensions and fieldnames

C++ syntax

```
#include "mclcppclass.h"
const char** fields = {"a", "b", "c"};
mwArray a(2, 2, 3, fields);
```

Arguments

num_rows. Number of rows in the `struct` matrix

num_cols. Number of columns in the `struct` matrix

num_fields. Number of fields in the `struct` matrix

fieldnames. Array of NULL-terminated strings representing the fieldnames.

Return value

None

Description

Use this constructor to create a matrix of type `mxSTRUCT_CLASS`, with the specified fieldnames. All elements are initialized with empty cells.

mwArray(int num_dims, const int* dims, int num_fields, const char** fieldnames)

Construct an n-dimensional `struct` array of the specified dimensions and fieldnames

C++ syntax

```
#include "mclcppclass.h"
const char** fields = {"a", "b", "c"};
int dims[3] = {2, 3, 4}
mwArray a(3, dims, 3, fields);
```

Arguments

num_dims. Size of the `dims` array

dims. Dimensions of the `struct` array

num_fields. Number of fields in the `struct` array

fieldnames. Array of NULL-terminated strings representing the fieldnames

Return value

None

Description

Use this constructor to create an n-dimensional array of type `mxSTRUCT_CLASS`, with the specified fieldnames. All elements are initialized with empty cells.

mwArray(const mwArray& arr)

`mwArray` copy constructor

Constructs a new array from an existing one.

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray b(a);
```

Arguments

arr. `mwArray` to copy

Return value

None

Description

Use this constructor to create a copy of an existing array. The new array contains a deep copy of the input array.

mwArray(<type> re)

Construct a real scalar array of the type of the input argument and initialize the data with the input argument's value.

C++ syntax

```
#include "mclcppclass.h"
double x = 5.0;
mwArray a(x);      // Creates a 1X1 double array with value 5.0
```

Arguments

re. Scalar value to initialize array with

Return value

None

Description

Use this constructor to create a real scalar array. `<type>` can be any of the following: `mxDouble`, `mxSingle`, `mxInt8`, `mxUint8`, `mxInt16`, `mxUint16`, `mxInt32`, `mxUint32`, `mxInt64`, `mxUint64`, or `mxLogical`. The scalar array is created with the type of the input argument.

mwArray(<type> re, <type> im)

Construct a complex scalar array of the type of the input arguments and initialize the real and imaginary parts of the data with the input argument's values.

C++ syntax

```
#include "mclcppclass.h"
double re = 5.0;
double im = 10.0;
mwArray a(re, im);// Creates a 1X1 complex array with value 5+10i
```

Arguments

re. Scalar value to initialize real part with

im. Scalar value to initialize imaginary part with

Return value

None

Description

Use this constructor to create a complex scalar array. The first input argument initializes the real part and the second argument initializes the imaginary part. `<type>` can be any of the following: `mxDouble`, `mxSingle`, `mxInt8`, `mxUint8`, `mxInt16`, `mxUint16`, `mxInt32`, `mxUint32`, `mxInt64`, or `mxUint64`. The scalar array is created with the type of the input arguments.

mwArray Clone() const

Return a new array representing a deep copy of this array

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray b = a.Clone();
```

Arguments

None

Return value

New `mwArray` representing a deep copy of the original.

Description

Use this method to create a copy of an existing array. The new array contains a deep copy of the input array.

mwArray SharedCopy() const

Return a new array representing a shared copy of this array

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray b = a.SharedCopy();
```

Arguments

None

Return value

New `mwArray` representing a reference counted version of the original.

Description

Use this method to create a shared copy of an existing array. The new array and the original array both point to the same data.

mwArray Serialize() const

Serialize the underlying array into a byte array, and return this data in a new array of type `mxUINT8_CLASS`

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray s = a.Serialize();
```

Arguments

None

Return value

New `mwArray` of type `mxUINT8_CLASS` containing the serialized data.

Description

Use this method to serialize an array into bytes. A 1-by-`n` numeric matrix of type `mxUINT8_CLASS` is returned containing the serialized data. The data can be deserialized back into the original representation by calling `maArray::Deserialize()`.

mxClassID ClassID() const

Return the type of this array

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
mxClassID id = a.ClassID();// Should return mxDOUBLE_CLASS
```

Arguments

None

Return value

The `mxClassID` of the array.

Description

Use this method to determine the type of the array. Consult the External Interfaces documentation for more information on `mxClassID`.

int ElementSize() const

Return the size in bytes of an element of this array

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
int size = a.ElementSize();// Should return sizeof(double)
```

Arguments

None

Return value

The size in bytes of an element of this type of array.

Description

Use this method to determine the size in bytes of an element of this array type.

int NumberOfElements() const

Return the number of elements in this array

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
int n = a.NumberOfElements();// Should return 4
```

Arguments

None

Return value

Number of elements in this array.

Description

Use this method to determine the total size of the array.

int NumberOfNonZeros() const

Return the number of nonzero elements for a sparse array

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
int n = a.NumberOfNonZeros();// Should return 4
```

Arguments

None

Return value

Number of nonzero elements in this array.

Description

Use this method to determine the size of the of the array's data. If the underlying array is not sparse, this returns the same value as `NumberOfElements()`.

Note This method does not analyze the actual values of the array elements.

---

int MaximumNonZeros() const

Return the maximum number of nonzero elements for a sparse array

If the underlying array is not sparse, returns the same value as `NumberOfElements()`.

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
int n = a.MaximumNonZeros();// Should return 4
```

Arguments

None

Return value

Number of allocated nonzero elements in this array.

Description

Use this method to determine the allocated size of the of the array's data. If the underlying array is not sparse, this returns the same value as `NumberOfElements()`.

Note This method does not analyze the actual values of the array elements.

---

int NumberOfDimensions() const

Return the number of dimensions in this array

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
int n = a.NumberOfDimensions();// Should return 4
```

Arguments

None

Return value

Number of dimensions in this array.

Description

Use this method to determine the dimensionality of the array.

int NumberOfFields() const

Return the number of fields in a `struct` array

C++ syntax

```
#include "mclcppclass.h"
const char** fields = {"a", "b", "c"};
mwArray a(2, 2, 3, fields);
int n = a.NumberOfFields(); // Should return 3
```

Arguments

None

Return value

Number of fields in the array.

Description

Use this method to determine the number of fields in a `struct` array. If the underlying array is not of type `struct`, zero is returned.

mwString GetFieldName(int index)

Return a string representing the name of the (zero-based) ith field in a `struct` array

C++ syntax

```
#include "mclcppclass.h"
const char** fields = {"a", "b", "c"};
mwArray a(2, 2, 3, fields);
const char* name = (const char*)a.GetFieldName(1); // Should
                                                   // return "b"
```

Arguments

Index. zero-based field number.

Return value

`mwString` containing the fieldname.

Description

Use this method to determine the name of a given field in a `struct` array. If the underlying array is not of type `struct`, an exception is thrown.

mwArray GetDimensions() const

Return an array of type `mxINT32_CLASS` representing the dimensions of this array

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray dims = a.GetDimensions();
```

Arguments

None

Return value

`mwArray` type `mxINT32_CLASS` containing the dimensions of the array.

Description

Use this method to determine the size of each dimension in the array. The size of the returned array is 1-by-`NumberOfDimensions()`.

bool IsEmpty() const

Return `true` if the underlying array is empty

C++ syntax

```
#include "mclcppclass.h"
mwArray a;
bool b = a.IsEmpty(); // Should return true
```

Arguments

None

Return value

Boolean indicating if the array is empty.

Description

Use this method to determine if an array is empty.

bool IsSparse() const

Return `true` if the underlying array is sparse

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
bool b = a.IsSparse(); // Should return false
```

Arguments

None

Return value

Boolean indicating if the array is sparse.

Description

Use this method to determine if an array is sparse.

bool IsNumeric() const

Return `true` if the underlying array is numeric

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
bool b = a.IsNumeric(); // Should return true.
```

Arguments

None

Return value

Boolean indicating if the array is numeric.

Description

Use this method to determine if an array is numeric.

bool IsComplex() const

Return `true` if the underlying array is complex

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS, mxCOMPLEX);
bool b = a.IsComplex(); // Should return true.
```

Arguments

None

Return value

Boolean indicating if the array is complex.

Description

Use this method to determine if an array is complex.

bool Equals(const mwArray& arr) const

Test two arrays for equality

C++ syntax

```
#include "mclcppclass.h"
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray b(1, 1, mxDOUBLE_CLASS);
a = 1.0;
b = 1.0;
bool b = a.Equals(b); // Should return true.
```

Arguments

arr. Array to compare to this array.

Return value

Boolean value indicating the equality of the two arrays.

Description

Returns `true` if the input array is byte-wise equal to this array. This method makes a byte-wise comparison of the underlying arrays. Therefore, arrays of the same type should be compared. Arrays of different types will not in general be equal, even if they are initialized with the same data.

int CompareTo(const mwArray& arr) const

Compare two arrays for order

C++ syntax

```
#include "mclcppclass.h"
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray b(1, 1, mxDOUBLE_CLASS);
a = 1.0;
b = 1.0;
int n = a.CompareTo(b); // Should return 0
```

Arguments

arr. Array to compare to this array.

Return value

Returns a negative integer, zero, or a positive integer if this array is less than, equal to, or greater than the specified array.

Description

Compares this array with the specified array for order. This method makes a byte-wise comparison of the underlying arrays. Therefore, arrays of the same type should be compared. Arrays of different types will, in general, not be ordered equivalently, even if they are initialized with the same data.

int HashCode() const

Return a hash code for this array

C++ syntax

```
#include "mclcppclass.h"
mwArray a(1, 1, mxDOUBLE_CLASS);
int n = a.HashCode();
```

Arguments

None

Return value

An integer value representing a unique hash code for the array.

Description

This method constructs a unique hash value form the underlying bytes in the array. Therefore, arrays of different types will have different hash codes, even if they are initialized with the same data.

mwString ToString() const

Return a string representation of the underlying array

C++ syntax

```
#include <stdio.h>
#include "mclcppclass.h"
mwArray a(1, 1, mxDOUBLE_CLASS, mxCOMPLEX);
a.Real() = 1.0;
a.Imag() = 2.0;
printf("%s\n", (const char*)(a.ToString()));   // Should print
                                               // "1 + 2i" on the
                                               // screen.
```

Arguments

None

Return value

An `mwString` containing the string representation of the array.

Description

This method returns a string representation of the underlying array. The string returned is the same string that is returned by typing a variable's name at the MATLAB command prompt.

mwArray RowIndex() const

Return an array containing the row indices of each element in this array

C++ syntax

```
#include <stdio.h>
#include "mclcppclass.h"
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray rows = a.RowIndex();
```

Arguments

None

Return value

An `mwArray` containing the row indices.

Description

Returns an array of type `mxINT32_CLASS` representing the row indices (first dimension) of this array. For sparse arrays, the indices are returned for just the non-zero elements and the size of the array returned is 1-by-`NumberOfNonZeros()`. For nonsparse arrays, the size of the array returned is 1-by-`NumberOfElements()`, and the row indices of all of the elements are returned.

mwArray ColumnIndex() const

Return an array containing the column indices of each element in this array

C++ syntax

```
#include "mclcppclass.h"
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray rows = a.ColumnIndex();
```

Arguments

None

Return value

An `mwArray` containing the column indices.

Description

Returns an array of type `mxINT32_CLASS` representing the column indices (second dimension) of this array. For sparse arrays, the indices are returned for just the non-zero elements and the size of the array returned is 1-by-`NumberOfNonZeros()`. For nonsparse arrays, the size of the array returned is 1-by-`NumberOfElements()`, and the column indices of all of the elements are returned.

void MakeComplex()

Convert a real numeric array to complex

C++ syntax

```
#include "mclcppclass.h"
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double idata[4] = {10.0, 20.0, 30.0, 40.0};
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(rdata, 4);
a.MakeComplex();
a.Imag().SetData(idata, 4);
```

Arguments

None

Return value

None

Description

Use this method to convert a numeric array that has been previously allocated as `real` to `complex`. If the underlying array is of a nonnumeric type, an `mwException` is thrown.

mwArray Get(int num_indices, ...)

Return the single element at the specified 1-based index

The index is passed by first passing the number of indices followed by a comma separated list of 1-based indices.

C++ syntax

```
#include "mclcppclass.h"
double data[4] = {1.0, 2.0, 3.0, 4.0};
double x;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(data, 4);
x = a.Get(1,1);         // x = 1.0
x = a.Get(2, 1, 2);     // x = 3.0
x = a.Get(2, 2, 2);     // x = 4.0
```

Arguments

num_indices. Number of indices passed in.

... Comma-separated list of input indices. Number of items must equal `num_indices`.

Return value

An `mwArray` containing the value at the specified index.

Description

Use this method to fetch a single element at a specified index. The index is passed by first passing the number of indices followed by a comma-separated list of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing), in which case the element at the specified 1-based offset is returned, accessing data in column-wise order, or `NumberOfDimensions()` (multiple subscript indexing), in which case, the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the ith index has the valid range: `1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

mwArray Get(const char* name, int num_indices, ...)

Return the single element at the specified field name and 1-based index in a struct array

The index is passed by first passing the number of indices followed by a comma-separated list of 1-based indices.

C++ syntax

```
#include "mclcppclass.h"
const char** fields = {"a", "b", "c"};

mwArray a(1, 1, 3, fields);
mwArray b = a.Get("a", 1, 1);
mwArray b = a.Get("b", 2, 1, 1);
```

Arguments

name. NULL-terminated string containing the field name to get.

num_indices. Number of indices passed in.

... Comma-separated list of input indices. Number of items must equal `num_indices`.

Return value

An `mwArray` containing the value at the specified field name and index.

Description

Use this method to fetch a single element at a specified field name and index. This method may only be called on an array that is of type `mxSTRUCT_CLASS`. An `mwException` is thrown if the underlying array is not a `struct` array. The field name passed must be a valid field name in the `struct` array. The index is passed by first passing the number of indices followed by a comma-separated list of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing), in which case the element at the specified 1-based offset is returned, accessing data in column-wise order, or `NumberOfDimensions()` (multiple subscript indexing), in which case, the index list is used to access the specified element. The valid range for indices is

`1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the ith index has the valid range: `1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

mwArray GetA(int num_indices, const int* index)

Return the single element at the specified 1-based index

The index is passed as an array of 1-based indices.

C++ syntax

```
#include "mclcppclass.h"
double data[4] = {1.0, 2.0, 3.0, 4.0};
int index[2] = {1, 1};
double x;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(data, 4);
x = a.GetA(1, index);      // x = 1.0
x = a.GetA(2, index);      // x = 1.0
index[0] = 2;
index[1] = 2;
x = a.Get(2, index);       // x = 4.0
```

Arguments

num_indices. Size of index array.

index. Array of at least size `num_indices` containing the indices.

Return value

An `mwArray` containing the value at the specified index.

Description

Use this method to fetch a single element at a specified index. The index is passed by first passing the number of indices, followed by an array of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing), in which case the element at the specified 1-based offset is returned, accessing data in column-wise order, or `NumberOfDimensions()` (multiple sub-script indexing), in which case, the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the ith index has the valid range:

`1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

mwArray GetA(const char* name, int num_indices, const int* index)

Return the single element at the specified field name and 1-based index in a `struct` array

The index is passed as an array of 1-based indices.

C++ syntax

```
#include "mclcppclass.h"
const char** fields = {"a", "b", "c"};
int index[2] = {1, 1};
mwArray a(1, 1, 3, fields);
mwArray b = a.Get("a", 1, index);
mwArray b = a.Get("b", 2, index);
```

Arguments

name. NULL-terminated string containing the field name to get.

num_indices. Number of indices passed in.

index. Array of at least size `num_indices` containing the indices.

Return value

An `mwArray` containing the value at the specified field name and index.

Description

Use this method to fetch a single element at a specified field name and index. This method may only be called on an array that is of type `mxSTRUCT_CLASS`. An `mwException` is thrown if the underlying array is not a `struct` array. The field name passed must be a valid field name in the `struct` array. The index is passed by first passing the number of indices followed by an array of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing), in which case the element at the specified 1-based offset is returned, accessing data in column-wise order, or `NumberOfDimensions()` (multiple subscript indexing), in which case, the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For

multiple subscript indexing, the ith index has the valid range:
`1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

mwArray Real()

Return an `mwArray` that references the real part of a complex array

C++ syntax

```
#include "mclcppclass.h"
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double idata[4] = {10.0, 20.0, 30.0, 40.0};
mwArray a(2, 2, mxDOUBLE_CLASS, mxCOMPLEX);
a.Real().SetData(rdata, 4);
a.Imag().SetData(idata, 4);
```

Arguments

None

Return value

An `mwArray` referencing the real part of the array.

Description

Use this method to access the real part of a complex array. The returned `mwArray` is considered real and has the same dimensionality and type as the original.

mwArray Imag()

Return an `mwArray` that references the imaginary part of a complex array

C++ syntax

```
#include "mclcppclass.h"
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double idata[4] = {10.0, 20.0, 30.0, 40.0};
mwArray a(2, 2, mxDOUBLE_CLASS, mxCOMPLEX);
a.Real().SetData(rdata, 4);
a.Imag().SetData(idata, 4);
```

Arguments

None

Return value

An `mwArray` referencing the imaginary part of the array.

Description

Use this method to access the imaginary part of a complex array. The returned `mwArray` is considered real and has the same dimensionality and type as the original.

void Set(const mwArray& arr)

Assign a shared copy of the input array to the currently referenced cell for arrays of type `mxCELL_CLASS` and `mxSTRUCT_CLASS`

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray b(2, 2, mxINT16_CLASS);
mwArray c(1, 2, mxCELL_CLASS);
c.Get(1,1).Set(a);     // Sets c(1) = a
a.Get(1,2).Set(b);     // Sets c(2) = b
```

Arguments

arr. `mwArray` to assign to currently referenced cell.

Return value

None

Description

Use this method to construct cell and struct arrays.

void GetData(<numeric-type>* buffer, int len) const

Copy the array's data into a supplied numeric buffer

C++ syntax

```
#include "mclcppclass.h"
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double data_copy[4] ;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(rdata, 4);
a.GetData(data_copy, 4);
```

Arguments

buffer. Buffer to receive copy.

len. Maximum length of buffer. A maximum of `len` elements will be copied.

Return value

None

Description

Valid types for `<numeric-type>` are `mxDOUBLE_CLASS`, `mxSINGLE_CLASS`, `mxINT8_CLASS`, `mxUINT8_CLASS`, `mxINT16_CLASS`, `mxUINT16_CLASS`, `mxINT32_CLASS`, `mxUINT32_CLASS`, `mxINT64_CLASS`, and `mxUINT64_CLASS`. The data is copied in column-major order. If the underlying array is not of the same type as the input buffer, the data is converted to this type as it is copied. If a conversion cannot be made, an `mwException` is thrown.

void GetLogicalData(mxLogical* buffer, int len) const

Copy the array's data into a supplied `mxLogical` buffer

C++ syntax

```
#include "mclcppclass.h"
mxLogical data[4] = {true, false, true, false};
mxLogical data_copy[4] ;
mwArray a(2, 2, mxLOGICAL_CLASS);
a.SetData(data, 4);
a.GetData(data_copy, 4);
```

Arguments

buffer. Buffer to receive copy.

len. Maximum length of buffer. A maximum of `len` elements will be copied.

Return value

None

Description

The data is copied in column-major order. If the underlying array is not of type `mxLOGICAL_CLASS`, the data is converted to this type as it is copied. If a conversion cannot be made, an `mwException` is thrown.

void GetCharData(mxChar* buffer, int len) const

Copy the array's data into a supplied `mxChar` buffer

C++ syntax

```
#include "mclcppclass.h"
mxChar data[6] = {'H', 'e' , `l' , 'l' , 'o' , '\0'};
mxChar data_copy[6] ;
mwArray a(1, 6, mxCHAR_CLASS);
a.SetData(data, 6);
a.GetData(data_copy, 6);
```

Arguments

buffer. Buffer to receive copy.

len. Maximum length of buffer. A maximum of `len` elements will be copied.

Return value

None

Description

The data is copied in column-major order. If the underlying array is not of type `mxCHAR_CLASS`, the data is converted to this type as it is copied. If a conversion cannot be made, an `mwException` is thrown.

void SetData(<numeric-type>* buffer, int len)

Copy the data from the supplied numeric buffer into the array

C++ syntax

```
#include "mclcppclass.h"
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double data_copy[4] ;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(rdata, 4);
a.GetData(data_copy, 4);
```

Arguments

buffer. Buffer containing data to copy.

len. Maximum length of buffer. A maximum of `len` elements will be copied.

Return value

None

Description

Valid types for `<numeric-type>` are `mxDOUBLE_CLASS`, `mxSINGLE_CLASS`, `mxINT8_CLASS`, `mxUINT8_CLASS`, `mxINT16_CLASS`, `mxUINT16_CLASS`, `mxINT32_CLASS`, `mxUINT32_CLASS`, `mxINT64_CLASS`, and `mxUINT64_CLASS`. The data is copied in column-major order. If the underlying array is not of the same type as the input buffer, the data is converted to this type as it is copied. If a conversion cannot be made, an `mwException` is thrown.

void SetLogicalData(mxLogical* buffer, int len)

Copy the data from the supplied `mxLogical` buffer into the array

C++ syntax

```
#include "mclcppclass.h"
mxLogical data[4] = {true, false, true, false};
mxLogical data_copy[4] ;
mwArray a(2, 2, mxLOGICAL_CLASS);
a.SetData(data, 4);
a.GetData(data_copy, 4);
```

Arguments

buffer. Buffer containing data to copy.

len.
Maximum length of buffer. A maximum of `len` elements will be copied.

Return value

None

Description

The data is copied in column-major order. If the underlying array is not of type `mxLOGICAL_CLASS`, the data is converted to this type as it is copied. If a conversion cannot be made, an `mwException` is thrown.

void SetCharData(mxChar* buffer, int len)

Copy the data from the supplied `mxChar` buffer into the array

C++ syntax

```
#include "mclcppclass.h"
mxChar data[6] = {'H', 'e' , `l' , 'l' , 'o' , '\0'};
mxChar data_copy[6] ;
mwArray a(1, 6, mxCHAR_CLASS);
a.SetData(data, 6);
a.GetData(data_copy, 6);
```

Arguments

buffer. Buffer containing data to copy.

len.
Maximum length of buffer. A maximum of `len` elements will be copied.

Return value

None

Description

The data is copied in column-major order. If the underlying array is not of type `mxCHAR_CLASS`, the data is converted to this type as it is copied. If a conversion cannot be made, an `mwException` is thrown.

mwArray operator()(int i1, int i2, int i3, ...,)

Return the single element at the specified 1-based index

The index is passed as a comma-separated list of 1-based indices. This operator is overloaded to support 1 through 32 indices.

C++ syntax

```
#include "mclcppclass.h"
double data[4] = {1.0, 2.0, 3.0, 4.0};
double x;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(data, 4);
x = a(1,1);    // x = 1.0
x = a(1,2);    // x = 3.0
x = a(2,2);    // x = 4.0
```

Arguments

i1, i2, i3, ..., Comma-separated list of input indices.

Return value

An `mwArray` containing the value at the specified index.

Description

Use this operator to fetch a single element at a specified index. The index is passed as a comma-separated list of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing), in which case the element at the specified 1-based offset is returned, accessing data in column-wise order, or `NumberOfDimensions()` (multiple subscript indexing), in which case, the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the ith index has the valid range: `1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

mwArray operator()(const char* name, int i1, int i2, int i3, ...,)

Return the single element at the specified field name and 1-based index in a `struct` array

The index is passed as a comma-separated list of 1-based indices. This operator is overloaded to support 1 through 32 indices.

C++ syntax

```
#include "mclcppclass.h"
const char** fields = {"a", "b", "c"};
int index[2] = {1, 1};
mwArray a(1, 1, 3, fields);
mwArray b = a("a", 1, 1);
mwArray b = a("b", 1, 1);
```

Arguments

name. NULL-terminated string containing the field name to get.

i1, i2, i3, ..., Comma-separated list of input indices.

Return value

An `mwArray` containing the value at the specified field name and index.

Description

Use this method to fetch a single element at a specified field name and index. This method may only be called on an array that is of type `mxSTRUCT_CLASS`. An `mwException` is thrown if the underlying array is not a `struct` array. The field name passed must be a valid field name in the `struct` array. The index is passed by first passing the number of indices, followed by an array of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing), in which case the element at the specified 1-based offset is returned, accessing data in column-wise order, or `NumberOfDimensions()` (multiple subscript indexing), in which case, the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the ith index has the valid range:

`1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

mwArray& operator=(const <type>& x)

Assign a single scalar value to the array

This operator is overloaded for all numeric and logical types.

C++ syntax

```
#include "mclcppclass.h"
mwArray a(2, 2, mxDOUBLE_CLASS);
a(1,1) = 1.0;       // assigns 1.0 to element (1,1)
a(1,2) = 2.0;       // assigns 2.0 to element (1,2)
a(2,1) = 3.0;       // assigns 3.0 to element (2,1)
a(2,2) = 4.0;       // assigns 4.0 to element (2,2)
```

Arguments

x. Value to assign.

Return value

A reference to the invoking `mwArray`.

Description

Use this operator to set a single scalar value.

operator <type>() const

Fetch a single scalar value from the array

This operator is overloaded for all numeric and logical types.

C++ syntax

```
#include "mclcppclass.h"
double data[4] = {1.0, 2.0, 3.0, 4.0};
double x;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(data, 4);
x = (double)a(1,1);    // x = 1.0
x = (double)a(1,2);    // x = 3.0
x = (double)a(2,1);    // x = 2.0
x = (double)a(2,2);    // x = 4.0
```

Arguments

None

Return value

A single scalar value from the array.

Description

Use this operator to fetch a single scalar value.

static mwArray Deserialize(const mwArray& arr)

Deserialize an array that has been serialized with `mwArray::Serialize`

C++ syntax

```
#include "mclcppclass.h"
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(rdata, 4);
mwArray b = a.Serialize();
a = mwArray::Deserialize(b);   // a should contain same data as
                                // original
```

Arguments

arr. `mwArray` that has been obtained by calling `mwArray::Serialize`

Return value

A new `mwArray` containing the deserialized array.

Description

Use this method to deserialize an array that has been serialized with `mwArray::Serialize()`. The input array must be of type `mxUINT8_CLASS` and contain the data from a serialized array. If the input data does not represent a serialized `mwArray`, the behavior of this method is undefined.

static double GetNaN()

Get the value of NaN (Not-a-Number)

C++ syntax

```
#include "mclcppclass.h"
double x = mwArray::GetNaN();
```

Arguments

None

Return value

The value of NaN (Not-a-Number) on your system.

Description

Call mwArray::GetNaN to return the value of NaN for your system. NaN is the IEEE arithmetic representation for Not-a-Number. Certain mathematical operations return NaN as a result, for example,

- 0.0/0.0
- Inf-Inf

The value of NaN is built in to the system; you cannot modify it.

static double GetEps()

Get the value of `eps`

C++ syntax

```
#include "mclcppclass.h"
double x = mwArray::GetEps();
```

Arguments

None

Return value

The value of the MATLAB `eps` variable.

Description

Call `mwArray::GetEps` to return the value of the MATLAB `eps` variable. This variable is the distance from 1.0 to the next largest floating-point number. Consequently, it is a measure of floating-point accuracy. The MATLAB `pinv` and `rank` functions use `eps` as a default tolerance.

static double GetInf()

Get the value of `Inf` (infinity)

C++ syntax

```
#include "mclcppclass.h"
double x = mwArray::GetInf();
```

Arguments

None

Return value

The value of `Inf` (infinity) on your system.

Description

Call `mwArray::GetInf` to return the value of the MATLAB internal `Inf` variable. `Inf` is a permanent variable representing IEEE arithmetic positive infinity. The value of `Inf` is built into the system; you cannot modify it.

Operations that return `Inf` include:

- Division by 0. For example, 5/0 returns `Inf`.
- Operations resulting in overflow. For example, `exp(10000)` returns `Inf` because the result is too large to be represented on your machine.

static bool IsFinite(double x)

Test if a value is finite. Returns `true` if the value is finite.

C++ syntax

```
#include "mclcppclass.h"
bool x = mwArray::IsFinite(1.0);   // Returns true
```

Arguments

Value to test for finiteness

Return value

Result of test.

Description

Call `mwArray::IsFinite` to determine whether or not a value is finite. A number is finite if it is greater than `-Inf` and less than `Inf`.

static bool IsInf(double x)

Test if a value is infinite. Returns `true` if the value is infinite.

C++ syntax

```
#include "mclcppclass.h"
bool x = mwArray::IsInf(1.0);      // Returns false
```

Arguments

Value to test for infinity

Return value

Result of test.

Description

Call `mwArray::IsInf` to determine whether or not a value is equal to infinity or minus infinity. MATLAB stores the value of infinity in a permanent variable named `Inf`, which represents IEEE arithmetic positive infinity. The value of the variable, `Inf`, is built into the system; you cannot modify it.

Operations that return infinity include:

- Division by 0. For example, 5/0 returns infinity.
- Operations resulting in overflow. For example, `exp(10000)` returns infinity because the result is too large to be represented on your machine. If the value equals `NaN` (Not-a-Number), then `mxIsInf` returns `false`. In other words, `NaN` is not equal to infinity.

static bool IsNaN(double x)

Test if a value is `NaN` (Not-a-Number). Returns `true` if the value is `NaN`.

C++ syntax

```
#include "mclcppclass.h"
bool x = mwArray::IsNaN(1.0);  // Returns false
```

Arguments

Value to test for `NaN`

Return value

Result of test.

Description

Call `mwArray::IsNaN` to determine whether or not the value is `NaN`. `NaN` is the IEEE arithmetic representation for Not-a-Number. `NaN` is obtained as a result of mathematically undefined operations such as

- 0.0/0.0
- `Inf-Inf`

The system understands a family of bit patterns as representing `NaN`. In other words, `NaN` is not a single value, rather it is a family of numbers that MATLAB (and other IEEE-compliant applications) use to represent an error condition or missing data.

Index

Symbols

A

B

C

D

E

F

N

O

P

V

W

Z