MATLAB® Compiler

The Language of Technical Computing

Computation

Visualization

Programming

User's Guide

Version 2

How to Contact The MathWorks:

508-647-7000 Phone

508-647-7001 Fax

The MathWorks, Inc.
24 Prime Park Way
Natick, MA 01760-1500 Mail

`http://www.mathworks.com` Web
`ftp.mathworks.com` Anonymous FTP server
`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
`subscribe@mathworks.com` Subscribing user registration
`service@mathworks.com` Order status, license renewals, passcodes
`info@mathworks.com` Sales, pricing, and general information

MATLAB Compiler User's Guide

Printing History:

September 1995	First printing	
March 1997	Second printing	
January 1998	Third printing	Revised for Version 1.2
January 1999	Fourth printing	Revised for Version 2.0 (Release 11)

Contents

1 Introducing the MATLAB Compiler

2 Installation and Configuration

3 Getting Started with MEX-Files

4 Stand-Alone Applications

5 Controlling Code Generation

6 Reference

A MATLAB Compiler Quick Reference

B Error and Warning Messages

C Directory Organization

D

1

Introducing the MATLAB Compiler

Introduction

This book describes version 2.0 of the MATLAB® Compiler. The MATLAB Compiler takes M-files as input and generates C or C++ source code as output. The MATLAB Compiler can generate these kinds of source code:

- C source code for building MEX-files.
- C or C++ source code for combining with other modules to form stand-alone applications. Stand-alone applications do not require MATLAB at runtime; they can run even if MATLAB is not installed on the system. The MATLAB Compiler *does* require the MATLAB C/C++ Math Library to create stand-alone applications that rely on the core math and data analysis capabilities of MATLAB. The MATLAB Compiler also requires the MATLAB C/C++ Graphics Library in order to create stand-alone applications that make use of Handle Graphics® functions.
- C code S-functions for use with Simulink®.
- C shared libraries (dynamically linked libraries, or DLLs, on Microsoft Windows NT) and C++ static libraries. These can be used without MATLAB on the system, but they do require the MATLAB C/C++ Math Library.

Note Version 2.0 of the MATLAB Compiler provides support for many of the MATLAB 5 features. Compiler 2.0 does *not* support `eval`, `input`, objects, C++ MEX-files, or numeric, nondouble data types such as `uint8`.

This chapter takes a closer look at these categories of C and C++ source code and explains the value of compiled code.

Before You Begin

Before reading this book, you should already be comfortable writing M-files. If you are not, see *Using MATLAB*.

Note The phrase *MATLAB interpreter* refers to the application that accepts MATLAB commands, executes M-files and MEX-files, and behaves as described in *Using MATLAB*. When you use MATLAB, you are using the MATLAB interpreter. The phrase *MATLAB Compiler* refers to this product

that translates M-files into C or C++ source code. 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.

New Features

MATLAB Compiler 2.0 supports much of the functionality of MATLAB 5. The new features of the Compiler are:

- Data Constructs
  - Multidimensional arrays
  - Cell arrays
  - Structure arrays
  - Sparse arrays
- Programming Tools
  - Variable input and output argument lists (`varargin`/`varargout`)
  - `try … catch … end`
  - `switch … end`
- Language Enhancements
  - Persistent variables
  - `load` and `save` commands
- Improved Compiler Options
- Macro Options
- Error/Warning Messages
- Improved `mex` and `mbuild` Scripts
- Stand-Alone Compiler

Data Constructs

Multidimensional Arrays. Multidimensional arrays in MATLAB are an extension of the two-dimensional matrix. You access a two-dimensional matrix element with two subscripts: the first represents the row index and the second represents the column index. In multidimensional arrays, use additional

subscripts for indexing. For example, a three-dimensional array has three subscripts and a four-dimensional array has four subscripts.

Cell Arrays. Cell arrays are a special class of MATLAB arrays where elements, or *cells*, contain MATLAB arrays. Cell arrays allow you to store dissimilar classes of arrays in the same array.

Structure Arrays. Structures are a class of MATLAB arrays that can store dissimilar arrays together. Structures differ from cell arrays in that you reference them by named fields.

Sparse Arrays. Sparse arrays provide an efficient representation of arrays that contain a significant number of zero-valued elements.

Programming Tools

Variable Input Arguments. The special argument `varargin` can be used to pass any number of input arguments to a function.

Variable Output Arguments. The special argument `varargout` can be used to return any number of output arguments from a function.

try ... catch ... end. Execution of the `try…catch` construct begins by executing the statements between `try` and `catch`. If this completes without an error, execution is complete: the statements between `catch` and `end` are *not* executed. If an error does occur, execution resumes with the statements between `catch` and `end`.

switch ... end. The `switch` statement lets you conditionally execute code depending on the value of a variable or expression.

Language Enhancements

Persistent Variables. Variables that are defined as persistent do not change value from one call to another. Persistent variables can be used within a function only and they remain in memory until the program is unloaded.

load and save Commands. The support for `load` and `save` has been enhanced to include loading into a structure.

Note For more information about these features and MATLAB 5 in general, see *Using MATLAB*.

Improved Compiler Options

The collection of new and improved options provides you with greater flexibility to control Compiler 2.0. You also have full access to Compiler 1.2 and its set of existing options. For more information, see the "mcc (Compiler 2.0)" and "mcc (Compiler 1.2)" reference pages in Chapter 6.

Macro Options

These options (`–m`, `–p`, `–x`, and `–S`) let you quickly and easily generate C and C++ stand-alone applications, and MATLAB and Simulink C MEX-files, respectively. Each macro replaces a sequence of several Compiler options making it much easier to generate your output.

Error/Warning Messages

The MATLAB Compiler 2.0 contains a comprehensive set of error and warning messages that help you isolate problems with your code. For more information, see Appendix B, "Error and Warning Messages."

Improved mex and mbuild Scripts

The `mex` script, which allows you to compile MEX-functions, and the `mbuild` script, which allows you to customize the building and linking of your code, have been enhanced to automatically search your system for supported third-party compilers.

Stand-Alone Compiler

You can run the MATLAB Compiler 2.0 from the DOS or UNIX command line, making it unnecessary to have MATLAB running on your system. You can call the Compiler directly from a makefile. This stand-alone MATLAB Compiler is faster than previous versions of the Compiler because it does not have to start MATLAB each time you invoke a compilation.

Compiler Licensing Changes

Starting with Compiler 1.2.1, a new licensing scheme has been employed that enables the product to be simpler and more user friendly.

The new licensing model:

- 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

Running MATLAB Not Required

In versions prior to 1.2.1, you could not run the MATLAB Compiler unless you were running MATLAB. On networked systems, this meant that one user would be holding the license for one copy of MATLAB and the Compiler, simultaneously. In effect, one user required both products and tied up both licenses until the user exited MATLAB. Although you can still run the Compiler from within MATLAB, it is not required. One user could be running the Compiler while another user could be using MATLAB.

Dedicated Compiler License

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.

MATLAB Compiler 1.2 Users

Differences Between Compiler 1.2 and 2.0

The earlier section, “New Features,” provides a comprehensive list of the features included in Compiler 2.0 that were not in Compiler 1.2.

Optimization

If you require optimized code, use the `–V1.2` option, which uses Compiler 1.2, along with the appropriate Compiler 1.2 options. See Appendix D, “Using Compiler 1.2,” for complete information on how to improve the performance of code generated by the MATLAB Compiler.

Overview

The MATLAB Compiler (`mcc`) can translate M-files into C files. The resultant C files can be used in any of the supported executable types including MEX, executable, or library by generating an appropriate *wrapper* file. A wrapper file contains the required interface between the Compiler-generated code and a supported executable type. For example, a MEX wrapper contains the MEX gateway routine that sets up the left- and right-hand arguments for invoking the Compiler-generated code.

The code produced by the MATLAB Compiler is independent of the final target type — MEX, executable, or library. The wrapper file provides the necessary interface to the target type.

Note MEX-files generated by the MATLAB Compiler 2.0 are not backward compatible. They require MATLAB 5.3/Release 11 or greater.

Creating MEX-Files

The MATLAB Compiler, when invoked with the –x macro option, produces a MEX-file from M-files. The Compiler:

1. Translates your M code to C code.
2. Generates a MEX wrapper.
3. Invokes the `mex` utility which builds the C MEX-file source into a MEX-file by linking the MEX-file with the MEX version of the math libraries (`libmatlbmx`).

Figure 1-1 illustrates the process of producing a MEX-file. The MATLAB interpreter dynamically loads MEX-files as they are needed. Some MEX-files run significantly faster than their M-file equivalents; in the end of this chapter we explain why this is so.

MATLAB users who do not have the MATLAB Compiler must write the source code for MEX-files in either Fortran or C. The *Application Program Interface Guide* explains the fundamentals of this process. To write MEX-files, you have

to know how MATLAB represents its supported data types and the MATLAB external interface (i.e., the application program interface, or API.)

If you are comfortable writing M-files and have the MATLAB Compiler, then you do not have to learn all the details involved in writing MEX-file source code.

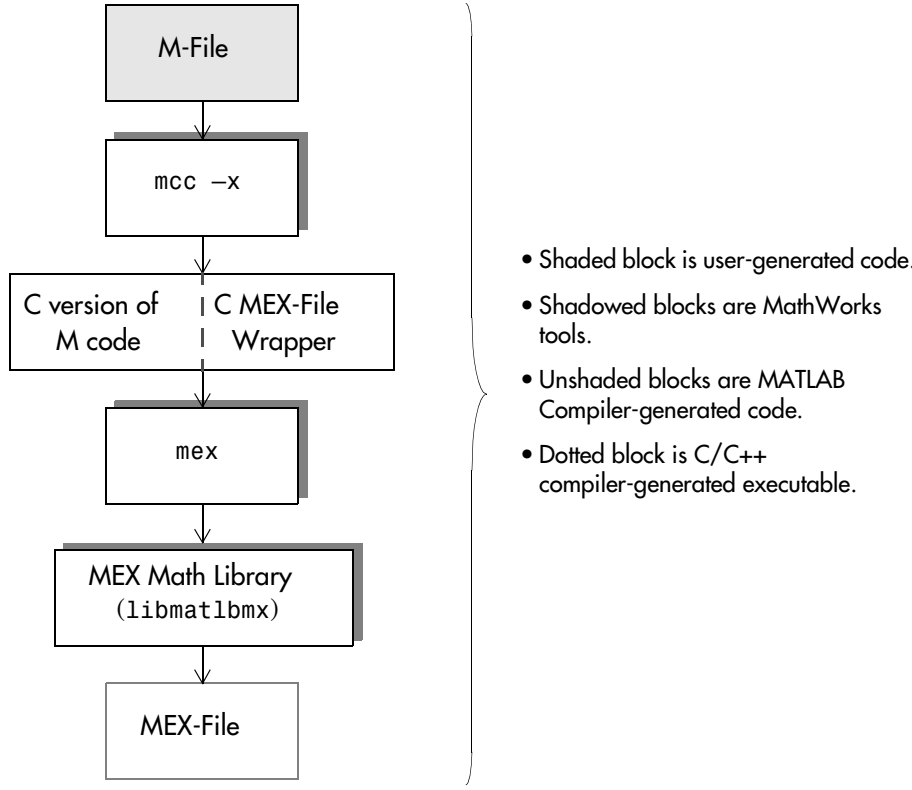


Figure 1-1: Developing MEX-Files

Creating Stand-Alone Applications

C Stand-Alone Applications

The MATLAB Compiler, when invoked with the appropriate option flag (`–m`), translates input M-files into C source code that is usable in any of the

supported executable types. The Compiler also produces the required wrapper file suitable for a stand-alone application. Then, your ANSI C compiler compiles these C source code files and the resulting object files are linked with the following libraries:

- The MATLAB M-File Math Library (`libmmfile`), which contains compiled versions of most MATLAB M-file math routines.
- The MATLAB Math Built-In Library (`libmatlb`), which contains compiled versions of most MATLAB built-in math routines.
- The MATLAB Array Access and Creation Library (`libmx`), which contains the array access routines.
- The MATLAB Utility Library (`libut`), which contains the utility routines used by various components in the background.
- The ANSI C Math Library.
- The MATLAB C/C++ Graphics Library (`libsgl`), if applicable.

The first two libraries (`libmmfile` and `libmatlb`) come with the MATLAB C/C++ Math Library product, the next two (`libmx` and `libut`) come with MATLAB. The ANSI C Math Library comes with your ANSI C compiler. The MATLAB C/C++ Graphics Library (`libsgl`) is available as a separate product from The MathWorks, Inc.

Note If you do *not* have the MATLAB C/C++ Graphics Library (`libsgl`), and your application calls a Handle Graphics function, a run-time error occurs.

To create C or C++ stand-alone applications, you must have the MATLAB C/C++ Math Library.

C++ Stand-Alone Applications

The MATLAB Compiler, when invoked with the appropriate option flag (`–p`), translates input M-files into C++ source code that is usable in any of the supported executable types. The Compiler also produces the required wrapper file suitable for a stand-alone application. Then, your C++ compiler compiles this C++ source code and the resulting object files are linked with the six C

object libraries listed above, as well as the MATLAB C++ Math Library, which contains C++ versions of MATLAB functions.

Note The MATLAB C++ Math Library must be linked first, then link the C object libraries listed above.

Developing a Stand-Alone Application

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. Figure 1-2 outlines this development process.

See Chapter 4 for complete details regarding stand-alone applications.

Figure 1-2 illustrates the process of developing a typical stand-alone C application. Use the same basic process for developing stand-alone C++ applications, but use the –p option instead of the –m option with the MATLAB Compiler, a C++ compiler instead of a C compiler, and the MATLAB C/C++ Math Library.

Note The MATLAB Compiler contains a tool, `mbuild`, which simplifies much of this process. Chapter 4, “Stand-Alone Applications,” describes the `mbuild` tool.

–p and –m are examples of options that you use to control how the Compiler works. Chapter 6 includes a complete description of the Compiler 2.0 options in the section, “mcc (Compiler 2.0).” Throughout this book you will see numerous examples of how these options are used with the Compiler to perform various tasks.

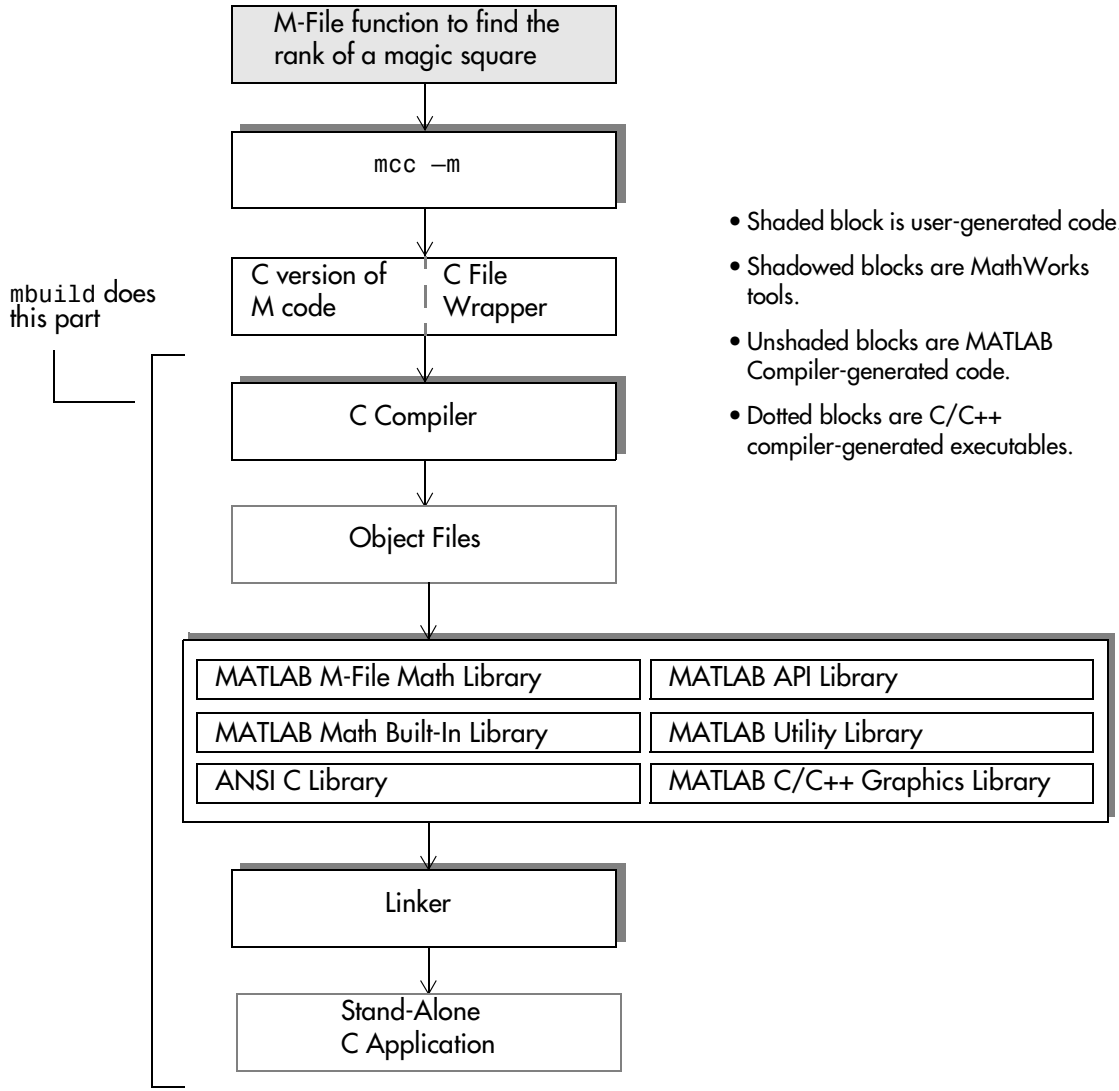


Figure 1-2: Developing a Typical Stand-Alone C Application

The MATLAB Compiler Family

Figure 1-3 illustrates the various ways you can use the MATLAB Compiler. The shaded blocks represent user-generated code; the unshaded blocks represent Compiler-generated code; the remaining blocks (drop shadow) represent MathWorks or other vendor tools.

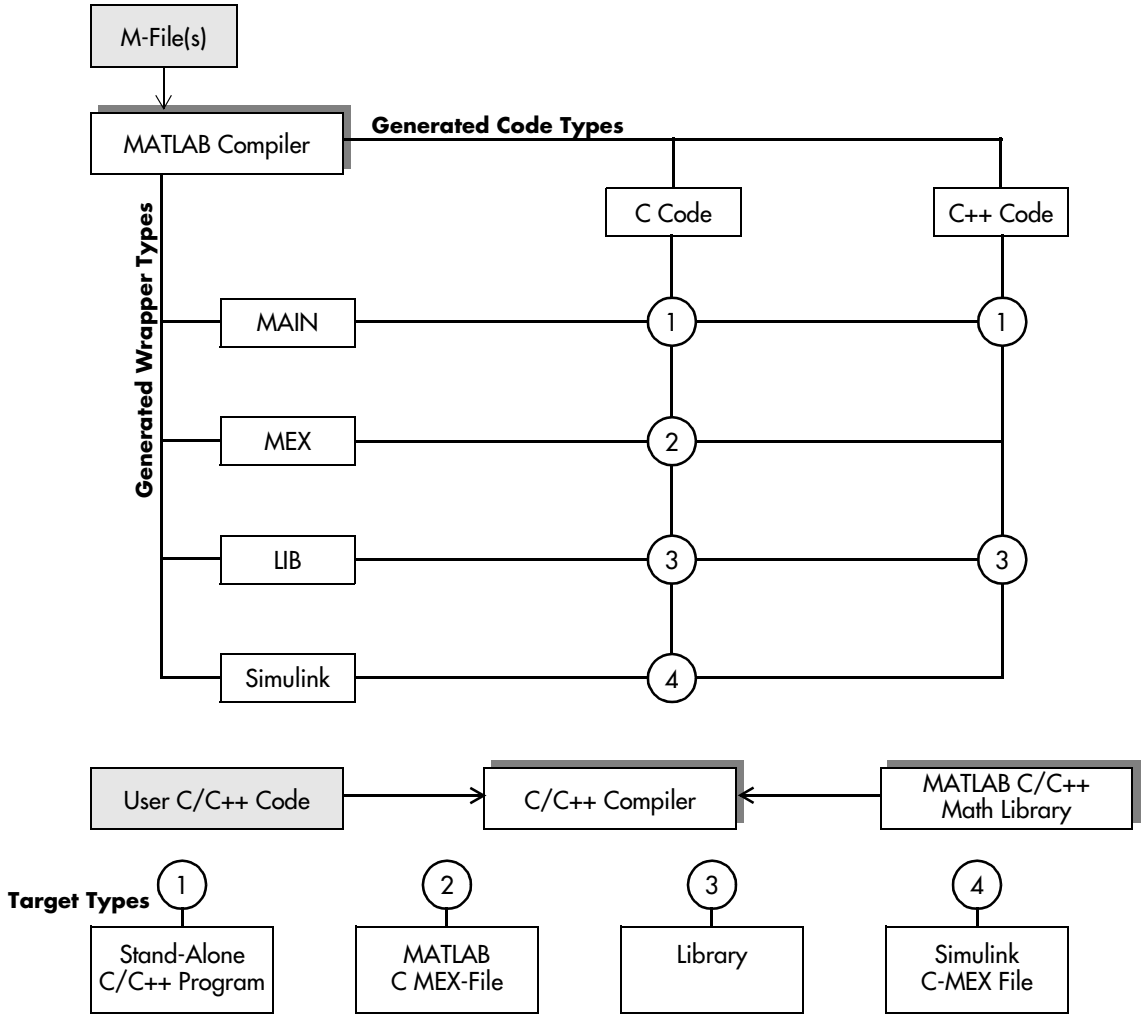


Figure 1-3: MATLAB Compiler Uses

The Compiler takes your M-file(s) and can generate C or C++ code. It can also generate one of four wrapper files depending on your specified target. This table shows the wrapper files the Compiler can generate, their associated targets, and the corresponding –W option (wrapper).

Table 1-1: Compiler 2.0 Wrappers and Targets

Wrapper File	Target	–W Setting
Main	Stand-alone C or C++ program	`–W main`
MEX	MATLAB C MEX-file	`–W mex`
Library	C shared library or C++ static library	`–W lib:`*`filename`*
Simulink S-function	Simulink C MEX-file	`–W simulink`

Each numbered node in Figure 1-3 indicates a combination of C/C++ code and a wrapper that generates a specific target type. The file(s) formed by combining the C/C++ code (denoted by User C/C++ Code) and the wrapper are then passed to the C/C++ compiler, which combines them with any user-defined C/C++ programs, and eventually links them against the appropriate libraries. The end result of this sequence is the target as described in Table 1-1.

Why Compile M-Files?

There are three main reasons to compile M-files:

- To speed them up
- To hide proprietary algorithms
- To create stand-alone applications or C shared libraries (DLLs on Windows) or C++ static libraries

Faster Execution

Compiled C or C++ code typically runs faster than its M-file equivalents because:

- Compiled code usually runs faster than interpreted code.
- C or C++ can avoid unnecessary memory allocation overhead that the MATLAB interpreter performs.

Cases When Performance Does Not Improve

Compilation is not likely to speed up M-file functions that:

- Are heavily vectorized
- Spend most of their time in MATLAB's built-in indexing, math, or graphics functions

Cases When Performance Does Improve

Compilation is most likely to speed up M-file functions that contain loops.

Compiler 1.2 Functionality

The previous release of the MATLAB Compiler, Version 1.2, incorporated several techniques that optimized the performance of the generated code. These optimizations are type imputation (`–r` and `–q`) and array boundary checking (`–i`).

- C or C++ code can contain simpler data types than M-files. The MATLAB interpreter assumes that all variables in M-files are matrices. By contrast, the MATLAB Compiler declares some C or C++ variables as simpler data types, such as scalar integers; a C or C++ compiler can take advantage of these simpler data types to produce faster code. For instance, the code to add

two scalar integers executes much faster than the code to add two matrices. The process that the Compiler uses to determine which variables can be declared as simpler data types is called *type imputation*.

M-files are likely to speed up after compilation if they include variables that the MATLAB Compiler views as integer or real scalars, or the files operate on real data only.

- C can avoid unnecessary array boundary checking. The MATLAB interpreter always checks array boundaries whenever an M-file assigns a new value to an array. By contrast, you can tell the MATLAB Compiler not to generate this array-boundary checking code in the C code. (Note that the `–i` option, which controls this, is not available in C++.)

Note Neither of these optimization schemes is available in the current release, 2.0, of the MATLAB Compiler. Subsequent releases of the Compiler will have increasing optimization capability, soon matching and ultimately surpassing that of Compiler 1.2. You can still use these optimizations if you use the `–V1.2` option. Appendix D, "Using Compiler 1.2," contains complete information about these optimizations.

Hiding Proprietary Algorithms

MATLAB M-files are ASCII text files that anyone can view and modify. MEX-files are binary files. Shipping MEX-files or stand-alone applications instead of M-files hides proprietary algorithms and prevents modification of your M-files.

Stand-Alone Applications and Libraries

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

You can develop an algorithm in MATLAB to perform specialized calculations and use the Compiler to create a C shared library (DLL on Windows) or a C++ static library. You can then integrate the algorithm into a C/C++ application.

After you compile the C/C++ application, you can use the MATLAB algorithm to perform specialized calculations from your program.

Upgrading from Previous Versions

MATLAB Compiler 1.2

Compatibility

The MATLAB Compiler 2.0 is fully compatible with previous releases of the Compiler. If you have M-files that were compiled with a previous version of the Compiler and compile them with the new version, you will get the same results.

Installation

The MATLAB 5.3 (Release 11) installer automatically installs Compiler 2.0. Once you install and configure Compiler 2.0, you can compile your M-files from either the MATLAB prompt or the DOS or UNIX command line.

Note If you require Compiler 1.2 functionality, you must use the `–V1.2` option. If you use the `–V1.2` option, you can *not* run the Compiler from the DOS or UNIX command line.

MATLAB Compiler 1.0/1.1

In many cases, M-code that was written and compiled in MATLAB 4.2 will work as is in the MATLAB 5 series (5.0, 5.1, and so on). There are, however, certain changes that could impact your work, especially if you integrated Compiler-generated code into a larger application.

Changed Library Name

Beginning with MATLAB 5.0, the name of the shared library that contains compiled versions of most MATLAB M-file math routines, `libtbx`, has changed. The new library is now called `libmmfile`.

Changed Data Type Names

In C, beginning with MATLAB 5.0 the name of the basic MATLAB data type, `Matrix`, has changed. The new name for the data type is `mxArray`.

In C++, beginning with MATLAB 5.0 the name of the basic MATLAB data type, `mwMatrix`, has changed. The new name for the data type is `mwArray`.

Note To learn more about the language changes that occurred between MATLAB 4.2 and MATLAB 5, see the online document, *Upgrading from MATLAB 4 to MATLAB 5.0*, available from the Help Desk.

2

Installation and Configuration

Getting Started

This chapter explains:

- The system requirements you need to use the MATLAB Compiler
- How to install the MATLAB Compiler
- How to configure the MATLAB Compiler after you have installed it

This chapter includes information for both MATLAB Compiler platforms — UNIX and Microsoft Windows.

For information about the MATLAB Compiler not available at print time, see the *Known Software and Documentation Problems* book.

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. The sections, "Things to Be Aware of," provide this information for each platform.

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.

Overview

This section outlines the steps necessary to configure your system to create MEX-files.

Note You must configure your system to create MEX-files even if you only want to create stand-alone applications or libraries.

The sequence of steps to install and configure the MATLAB Compiler so that it can generate MEX-files is:

1. Install the MATLAB Compiler.
2. Install an ANSI C or C++ compiler, if you don't already have one installed.
3. Verify that `mex` can generate MEX-files.
4. Verify that the MATLAB Compiler can generate MEX-files from the MATLAB command line and from the UNIX or DOS command line.

Figure 2-1 shows the sequence on both platforms. The sections following the flowchart provide more specific details for the individual platforms. Additional steps may be necessary if you plan to create stand-alone applications or libraries, however, you still must perform the steps given in this chapter first. Chapter 4, "Stand-Alone Applications," provides the details about the additional installation and configuration steps necessary for creating stand-alone applications and libraries.

Note This flowchart assumes that MATLAB is properly installed on your system.

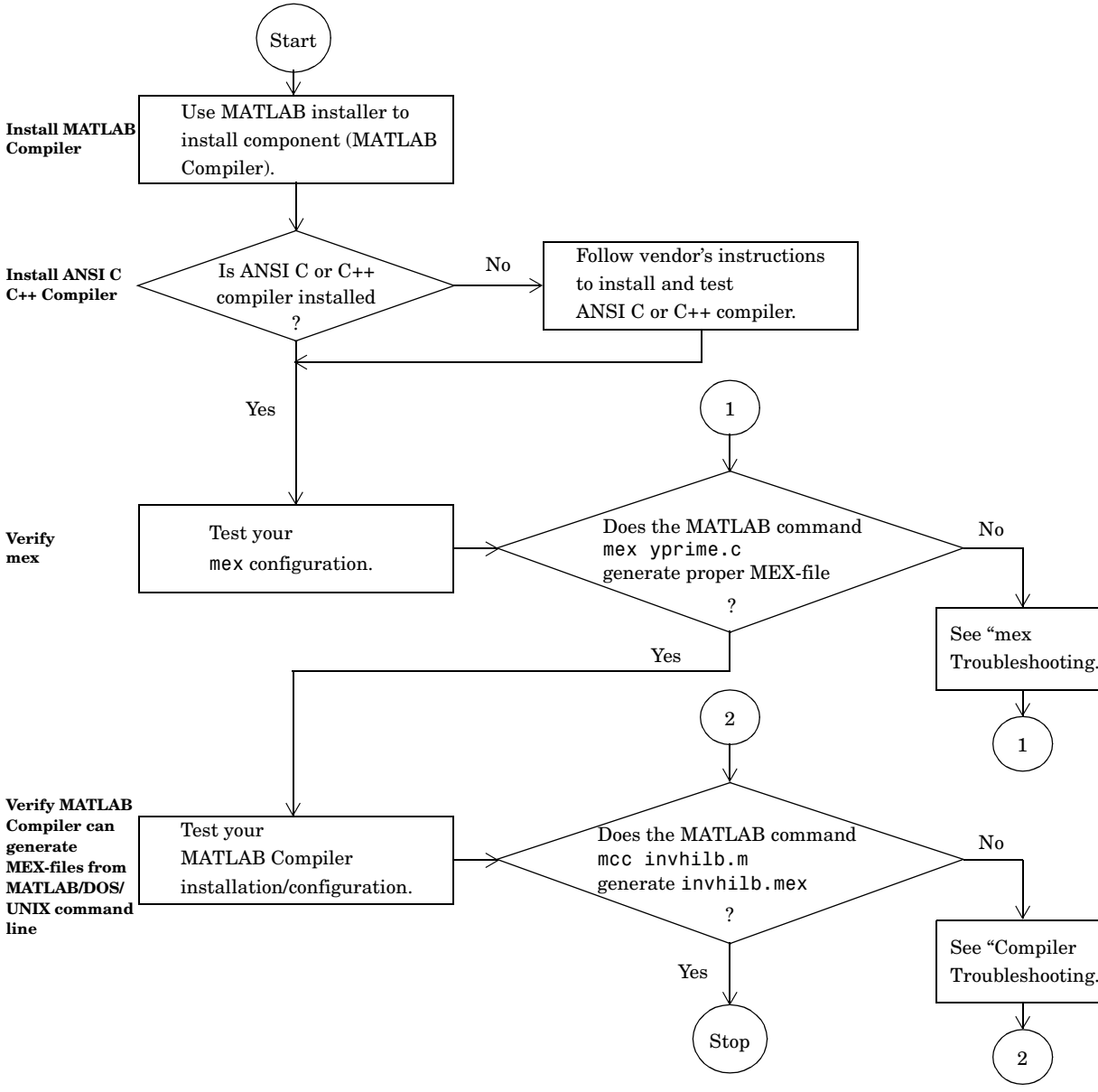


Figure 2-1: MATLAB Compiler Installation Sequence for Creating MEX-Files

UNIX Workstations

This section examines the system requirements, installation procedures, and configuration procedures for the MATLAB Compiler on UNIX systems.

System Requirements

You cannot install the MATLAB Compiler unless MATLAB 5.3/Release 11 or a later version is already installed on the 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.

Table 2-1 shows the requirements for creating UNIX applications with the MATLAB Compiler.

Table 2-1: Requirements for Creating UNIX Applications

To create...	You need...
MEX-files	ANSI C compiler MATLAB Compiler
Stand-alone C applications	ANSI C compiler MATLAB Compiler MATLAB C/C++ Math Library
Stand-alone C++ applications	C++ compiler MATLAB Compiler MATLAB C/C++ Math Library

The MATLAB C/C++ Math Library is a separately sold product.

Note Although the MATLAB Compiler supports the creation of stand-alone C++ applications, it does not support the creation of C++ MEX-files.

Supported ANSI C and C++ UNIX Compilers

The MATLAB Compiler supports the GNU C compiler, `gcc`, (except on HP and SGI64), the system's native ANSI C compiler on all UNIX platforms, the system's native C++ compiler on all UNIX platforms (except Linux), and the GNU C++ compiler, `g++`, on Linux.

Note For a list of all the compilers supported by MATLAB, see the MathWorks Technical Support Department's Technical Notes at:

```
http://www.mathworks.com/support/tech-notes/v5/1600/1601.shtml
```

Known Compiler Limitations. There are several known restrictions regarding the use of supported compilers.

- The SGI C compiler does not handle denormalized floating-point values correctly. Denormalized floating-point numbers are numbers that are less than the value of `DBL_MIN` in the compiler's `float.h` file.
- Due to a limitation of the GNU C++ compiler (`g++`) on Linux, `try…catch…end` blocks do not work.
- The `–A debugline:on` option does not work on the GNU C++ compiler (`g++`) on Linux because it uses `try…catch…end`.

Compiler Options Files

The MathWorks provides options files for every supported C or C++ compiler. These files contain the necessary flags and settings for the compiler. This table shows the preconfigured options files that are included with MATLAB for UNIX.

Compiler	Options File
System native ANSI compiler	`mexopts.sh`
`gcc` (GNU C compiler)	`gccopts.sh`

We provide information on the options files 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.

Locating Options Files

To locate your options file, the `mex` script searches the following:

- The current directory
- `$HOME/matlab`
- `<matlab>/bin`

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

Installation

MATLAB Compiler

To install the MATLAB Compiler on UNIX systems, follow the instructions in the *MATLAB Installation Guide for UNIX*. 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 MATLAB Compiler does not appear as one of the installation choices, contact The MathWorks to get an updated license file (`license.dat`):

- Via the Web at `www.mathworks.com`. On the MathWorks home page, click on the MATLAB Access option, log in to the Access home page, and follow the instructions. MATLAB Access membership is free of charge and available to all customers.
- Via e-mail at `service@mathworks.com`
- Via telephone at 508-647-7000; ask for Customer Service
- Via fax at 508-647-7001

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. The following section, "Things to Be

Aware of," contains several UNIX-specific details regarding the installation and configuration of your ANSI C or C++ compiler.

Note On some UNIX platforms, a C or C++ compiler may already be installed. Check with your system administrator for more information.

Things to Be Aware of

This table provides information regarding the installation and configuration of a C or C++ compiler on your system.

Description	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.

mex Verification

Choosing a Compiler

Using the System Compiler. If the MATLAB Compiler and your supported C or C++ compiler are installed on your system, you are ready to create C MEX-files. To create a MEX-file, you can simply enter

```
mex filename.c
```

This simple method of creating MEX-files works for the majority of users. It uses the system's compiler as your default compiler for creating C MEX-files.

If you do not need to change C or C++ compilers, or you do not need to modify your compiler options files, you can skip ahead in this section to "Creating MEX-Files." If you need to know how to change the options file, continue with this section.

Changing Compilers

Changing the Default Compiler. To change your default C or C++ compiler, you select a different options file. You can do this at anytime by using the command

```
mex –setup

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

    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 'mex –f' command
    (see 'mex –help' for more information).

The options files available for mex are:

  1: <matlab>/bin/gccopts.sh :
         Template Options file for building gcc MEX-files
  2: <matlab>/bin/mexopts.sh :
         Template Options file for building MEX-files using the
         system ANSI compiler

Enter the number of the options file to use as your default options
file:
```

Select the proper options file for your system by entering its number and pressing **Return**. If an options file doesn't exist in your MATLAB directory, the system displays a message stating that the options file is being copied to your

user-specific matlab directory. If an options file already exists in your matlab directory, the system prompts you to overwrite it.

Note 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.

Using the setup option resets your default compiler so that the new compiler is used every time you use the mex script.

Modifying the Options File. Another use of the setup option is if you want to change your options file settings. 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.

As the previous note says, setup copies the appropriate options file to your individual directory. To make your user-specific changes to the options file, you then 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.

Temporarily Changing the Compiler. To temporarily change your C or C++ compiler, use the –f option, as in

```
mex –f <file> …
```

The –f option tells the mex script to use the options file, <file>. If <file> is not in the current directory, then <file> must be the full pathname to the desired options file. Using the –f option tells the mex script to use the specified options file for the current execution of mex only; it does not reset the default compiler.

Creating MEX-Files

To create MEX-files on UNIX, first copy the source file(s) to a local directory, and then change directory (cd) to that local directory.

On UNIX, MEX-files are created with platform-specific extensions, as shown in Table 2-2.

Table 2-2: MEX-File Extensions for UNIX

Platform	MEX-File Extension
DEC Alpha	`mexaxp`
HP 9000 PA-RISC	`mexhp7`
IBM RS/6000	`mexrs6`
Linux	`mexlx`
SGI	`mexsg`
SGI 64	`mexsg64`
Solaris	`mexsol`

The `<matlab>/extern/examples/mex` directory contains C source code for the example `yprime.c`. After you copy the source file (`yprime.c`) to a local directory and `cd` to that directory, enter at the MATLAB prompt:

```
mex yprime.c
```

This should create the MEX-file called `yprime` with the appropriate extension corresponding to your UNIX platform. For example, if you create the MEX-file on Solaris, its name is `yprime.mexsol`.

You can now call `yprime` as if it were an M-function. For example,

```
yprime(1,1:4)
ans =
     2.0000    8.9685    4.0000    -1.0947
```

If you encounter problems generating the MEX-file or getting the correct results, refer to the *Application Program Interface Guide* for additional information about MEX-files.

MATLAB Compiler Verification

Verifying from MATLAB

Once you have verified that you can generate MEX-files on your system, you are ready to verify that the MATLAB Compiler is correctly installed. Type the following at the MATLAB prompt:

```
mcc –x invhilb
```

After a short delay, this command should complete and display the MATLAB prompt. Next, at the MATLAB prompt, type:

```
which invhilb
```

The `which` command should indicate that `invhilb` is now a MEX-file by listing the filename followed by the appropriate UNIX MEX-file extension. For example, if you run the Compiler on Solaris, the Compiler creates the file `invhilb.mexsol`. Finally, at the MATLAB prompt, type:

```
invhilb(10)
```

Note that this tests only the compiler's ability to make MEX-files. If you want to create stand-alone applications, refer to Chapter 4, "Stand-Alone Applications," for additional details.

Verifying from UNIX Command Prompt

To verify that the Compiler can generate MEX-files from the UNIX command prompt, you follow a similar procedure as that used in the previous section.

Note Before you test to see if the Compiler can generate MEX-files from the UNIX command prompt, you may want to delete the MEX-file you created in the previous section, `invhilb.mexsol`, or whatever the extension is on your system. That way, you can be sure your newly generated MEX-file is the result of using the Compiler from the UNIX prompt.

Copy `invhilb.m` from the `<matlab>/toolbox/matlab/elmat` directory to a local directory and then type the following at the UNIX prompt:

```
mcc –x invhilb
```

Next, verify that `invhilb` is now a MEX-file by listing the `invhilb` files:

```
ls invhilb.*
```

You will see a list similar to this:

```
invhilb.c        invhilb.m        invhilb_mex.c
invhilb.h        invhilb.mexsol*
```

These are the various files that the Compiler generates from the M-file. The Compiler-generated MEX-file appears in the list as the filename followed by the appropriate UNIX MEX-file extension. In this example, the Compiler was executed on Solaris, so the Compiler creates the file `invhilb.mexsol`. For more information on which files the Compiler creates for a compilation, see Chapter 5, " Controlling Code Generation."

To test the newly created MEX-file, start MATLAB and, at the MATLAB prompt, type:

```
invhilb(10)
```

Microsoft Windows on PCs

This section examines the system requirements, installation procedures, and configuration procedures for the MATLAB Compiler on PCs running Windows 95/98 or Windows NT.

System Requirements

You cannot install the MATLAB Compiler unless MATLAB 5.3/Release 11 or a later version is already installed on the system. The MATLAB Compiler imposes no operating system or memory requirements beyond what is necessary to run MATLAB. The MATLAB Compiler consumes a small amount of disk space.

Table 2-3 shows the requirements for creating PC applications with the MATLAB Compiler.

Table 2-3: Requirements for Creating PC Applications

To create...	You need...
MEX-files	ANSI C compiler MATLAB Compiler
Stand-alone C applications	ANSI C compiler MATLAB Compiler MATLAB C/C++ Math Library
Stand-alone C++ applications	C++ compiler MATLAB Compiler MATLAB C/C++ Math Library

The MATLAB C/C++ Math Library is a separately sold product.

Note Although the MATLAB Compiler supports the creation of stand-alone C++ applications, it does not support the creation of C++ MEX-files.

Supported ANSI C and C++ PC Compilers

To create C MEX-files, stand-alone C/C++ applications, or dynamically linked libraries (DLLs) with the MATLAB Compiler, you must install and configure a supported C/C++ compiler. Use one of the following 32-bit C/C++ compilers that create 32-bit Windows dynamically linked libraries (DLLs) or Windows NT applications:

- Watcom C/C++ versions 10.6 & 11.0
- Borland C++ versions 5.0, 5.2, & 5.3
- Microsoft Visual C++ (MSVC) versions 4.2, 5.0, & 6.0

Note For a list of all the compilers supported by MATLAB, see the MathWorks Technical Support Department's Technical Notes at:

```
http://www.mathworks.com/support/tech-notes/v5/1600/1601.shtml
```

To create stand-alone applications or DLLs, you also need the MATLAB C/C++ Math Library, which is sold separately. Also, if your applications use Handle Graphics, you will need the MATLAB C/C++ Graphics Library, which is sold separately.

Applications generated by the MATLAB Compiler are 32-bit applications and only run on Windows 95/98 and Windows NT systems.

Known Compiler Limitations. There are several known restrictions regarding the use of supported compilers.

- Some compilers, e.g., Watcom, do not handle denormalized floating-point values correctly. Denormalized floating-point numbers are numbers that are less than the value of `DBL_MIN` in your compiler's `float.h` file.
- The MATLAB Compiler 2.0 sometimes will generate `goto` statements for complicated `if` conditions. The Borland C++ Compiler prohibits the `goto` statement within a `try…catch` block. This error can occur if you use the `–A debugline:on` option, because its implementation uses `try…catch`. To work around this limitation, simplify the `if` conditions.
- 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 this an illegal octal integer

```
009.0
```

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 = [008 09 10];
```

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

```
x = [8 9 10];
```

Compiler Options Files

The MathWorks provides options files for every supported C or C++ compiler. These files contain the necessary flags and settings for the compiler. This table shows the preconfigured PC options files that are included with MATLAB.

Compiler	Options File

Microsoft C/C++, Version 4.2
Microsoft C/C++, Version 5.0
Microsoft C/C++, Version 6.0
	`msvcopts.bat`
	`msvc50opts.bat`
	`msvc60opts.bat`

Watcom C/C++, Version 10.6 Watcom C/C++, Version 11.0	`watcopts.bat` `wat11copts.bat`

Borland C++, Version 5.0
Borland C++, Version 5.2
Borland C++, Version 5.3
	`bccopts.bat`
	`bccopts.bat`
	`bcc53opts.bat`

Note We provide information on the options files 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.

Locating Options Files

To locate your options file, the `mex` script searches the following:

- The current directory
- The `user profile` directory (see the following section, "The User Profile Directory Under Windows," for more information about this directory)
- `<matlab>\bin`

`mex` uses the first occurrence of the options file it finds. If no options file is found, `mex` 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 `mex` and `mbuild` utilities store their respective options files, `mexopts.bat` and `compopts.bat`, which are created during the `–setup` process, in a subdirectory of your `user profile` directory, named `Application Data\MathWorks\MATLAB`. Under Windows NT and Windows 95/98 with user profiles enabled, your `user profile` directory is `%windir%\Profiles\username`. Under Windows 95/98 with user profiles disabled, your `user profile` directory is `%windir%`. Under Windows 95/98, you can determine whether or not user profiles are enabled by using the **Passwords** control panel.

Installation

MATLAB Compiler

To install the MATLAB Compiler on a PC, follow the instructions in the *MATLAB Installation Guide for PC*. 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:

- Via the Web at `www.mathworks.com`. On the MathWorks home page, click on the MATLAB Access option, log in to the Access home page, and follow the instructions. MATLAB Access membership is free of charge and available to all customers.
- Via e-mail at `service@mathworks.com`
- Via telephone at 508-647-7000, ask for Customer Service
- Via fax at 508-647-7001

ANSI C or C++ Compiler

To install your ANSI C or C++ compiler, follow the vendor's instructions that accompany your compiler. Be sure to test the C/C++ compiler to make sure it is installed and configured properly. The following section, "Things to Be Aware of," contains some Windows-specific details regarding the installation and configuration of your C/C++ compiler.

Things to Be Aware of

This table provides information regarding the installation and configuration of a C/C++ compiler on your system.

Description	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	Microsoft Foundation Classes (MFC) are not required.

Description	Comment
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.
Updating the registry	If your installer gives you the option of updating the registry, you should do it.
Installing Microsoft Visual 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.

mex Verification

Choosing a Compiler

Systems with Exactly One C/C++ Compiler. If you have properly installed the MATLAB Compiler and your supported C or C++ compiler, you can now create C MEX-files. On systems where there is exactly one C or C++ compiler available to you, the `mex` utility automatically configures itself for the appropriate compiler. So, for many users, to create a C MEX-file, you can simply enter

```
mex filename.c
```

This simple method of creating MEX-files works for the majority of users. It uses your installed C or C++ compiler as your default compiler for creating your MEX-files.

If you are a user who does not need to change compilers, or you do not need to modify your compiler options files, you can skip ahead in this section to "Creating MEX-Files."

Systems with More than One C/++ Compiler. On systems where there is more than one C or C++ compiler, the `mex` utility lets you select which of the compilers you

want to use. Once you choose your C or C++ compiler, that compiler becomes your default compiler and you no longer have to select one when you compile MEX-files.

For example, if your system has both the Borland and Watcom compilers, when you enter for the first time

```
mex filename.c
```

you are asked to select which compiler to use.

```
mex has detected the following compilers on your machine:

[1] : Borland compiler in T:\Borland\BC.500
[2] : WATCOM compiler in T:\watcom\c.106

[0] : None

Please select a compiler. This compiler will become the default:
```

Select the desired compiler by entering its number and pressing **Return**. You are then asked to verify the information.

Changing Compilers

Changing the Default Compiler. To change your default C or C++ compiler, you select a different options file. You can do this at any time by using the `mex –setup` option.

This example shows the process of changing your default compiler to the Microsoft Visual C++ Version 6.0 compiler.

```
mex –setup

Please choose your compiler for building external interface (MEX)
files.

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

Choose your C compiler:
[1] Borland C          (version 5.0, 5.2, or 5.3)
[2] Microsoft Visual C (version 4.2, 5.0, or 6.0)
[3] Watcom C           (version 10.6 or 11)

Or choose a Fortran compiler:
[4] DIGITAL Visual Fortran (version 5.0)

[0] None

Compiler: 2

Choose the version of your C compiler:
[1] Microsoft Visual C 4.2
[2] Microsoft Visual C 5.0
[3] Microsoft Visual C 6.0

version: 3

Your machine has a Microsoft Visual C compiler located at
D:\DevStudio6.
Do you want to use this compiler [y]/n? y

Please verify your choices:

Compiler: Microsoft Visual C 6.0
Location: D:\DevStudio6

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

```
The default options file:
"C:\WINNT\Profiles\username
\Application Data\MathWorks\MATLAB\mexopts.bat" is being
updated...
```

If the specified compiler cannot be located, you are given the message:

```
The default location for compiler-name is directory-name,
but that directory does not exist on this machine.
Use directory-name anyway [y]/n?
```

Using the `setup` option sets your default compiler so that the new compiler is used everytime you use the `mex` script.

Modifying the Options File. Another use of the `setup` option is if you want to change your options file settings. 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.

The `setup` option copies the appropriate options file to your `user profile` directory. To make your user-specific changes to the options file, you 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 `mex` script will use the new options file everytime with your modified settings.

Temporarily Changing the Compiler. To temporarily change your C or C++ compiler, use the `–f` option, as in

```
mex –f <file> …
```

The `–f` option tells the `mex` script to use the options file, `<file>`. If `<file>` is not in the current directory, then `<file>` must be the full pathname to the desired options file. Using the `–f` option tells the `mex` script to use the specified options file for the current execution of `mex` only; it does not reset the default compiler.

Creating MEX-Files

The `<matlab>\extern\examples\mex` directory contains C source code for the example `yprime.c`. To verify that your system can create MEX-files, enter at the MATLAB prompt:

```
cd([matlabroot '\extern\examples\mex'])
mex yprime.c
```

This should create the `yprime.dll` MEX-file. MEX-files created on Windows 95/98 or NT always have the extension `dll`.

You can now call `yprime` as if it were an M-function. For example,

```
yprime(1,1:4)
ans =
     2.0000    8.9685    4.0000    -1.0947
```

If you encounter problems generating the MEX-file or getting the correct results, refer to the *Application Program Interface Guide* for additional information about MEX-files.

MATLAB Compiler Verification

Verifying from MATLAB

Once you have verified that you can generate MEX-files on your system, you are ready to verify that the MATLAB Compiler is correctly installed. Type the following at the MATLAB prompt:

```
mcc -x invhilb
```

After a short delay, this command should complete and display the MATLAB prompt. Next, at the MATLAB prompt, type:

```
which invhilb
```

The `which` command should indicate that `invhilb` is now a MEX-file; it should have created the file `invhilb.dll`. Finally, at the MATLAB prompt, type:

```
invhilb(10)
```

Note that this tests only the compiler's ability to make MEX-files. If you want to create stand-alone applications or DLLs, refer to Chapter 4, "Stand-Alone Applications," for additional details.

Verifying from DOS Command Prompt

To verify that the Compiler can generate C MEX-files from the DOS command prompt, you follow a similar procedure as that used in the previous section.

Note Before you test to see if the Compiler can generate MEX-files from the DOS command prompt, you may want to delete the MEX-file you created in the previous section, `invhilb.dll`. That way, you can be sure your newly generated MEX-file is the result of using the Compiler from the DOS prompt. To delete this file, you must clear the MEX-file or quit MATLAB; otherwise the deletion will fail.

Copy `invhilb.m` from the `<matlab>\toolbox\matlab\elmat` directory to a local directory and then type the following at the DOS prompt:

```
mcc –x invhilb
```

Next, verify that `invhilb` is now a MEX-file by listing the `invhilb` files:

```
dir invhilb*
```

You will see a list containing:

```
invhilb.c
invhilb.dll
invhilb.h
invhilb.m
invhilb_mex.c
```

These are the files that the Compiler generates from the M-file, in addition to the original M-file, `invhilb.m`. The Compiler-generated MEX-file appears in the list as the filename followed by the extension, `dll`. In this example, the Compiler creates the file `invhilb.dll`. For more information on which files the Compiler creates for a compilation, see Chapter 5, " Controlling Code Generation."

To test the newly created MEX-file, you would start MATLAB and, at the MATLAB prompt, you could type:

```
invhilb(10)
```

Troubleshooting

This section identifies some of the more common problems that can occur when installing and configuring the MATLAB Compiler.

mex Troubleshooting

Non-ANSI C Compiler on UNIX

A common configuration problem in creating C MEX-files on UNIX involves using a non-ANSI C compiler. You must use an ANSI C compiler.

DLLs Not on Path on Windows

MATLAB will fail to load MEX-files if it cannot find all DLLs referenced by the MEX-file; the DLLs must be on the DOS path or in the same directory as the MEX-file. This is also true for third-party DLLs.

Segmentation Violation or Bus Error

If your MEX-file causes a segmentation violation or bus error, there is most likely a problem with the MATLAB Compiler. Contact Technical Support at The MathWorks.

Generates Wrong Answers

If your program generates the wrong answer(s), there are several possible causes. There could be an error in the computational logic or there may be a defect in the MATLAB Compiler. Run your original M-file with a set of sample data and record the results. Then run the associated MEX-file with the sample data and compare the results with those from the original M-file. If the results are the same, there may be a logic problem in your original M-file. If the results differ, there may be a defect in the MATLAB Compiler. In this case, send the pertinent information via e-mail to `support@mathworks.com`.

mex Works from Shell But Not from MATLAB (UNIX)

If the command

```
mex –x yprime.c
```

works from the UNIX shell 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`, an error may occur. You can test whether this is true 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`.

Cannot Locate Your Compiler (PC)

If `mex` has difficulty locating your installed compilers, it is useful to know how it goes about finding compilers. `mex` automatically detects your installed compilers by first searching for locations specified in the following environment variables:

- `BORLAND` for Borland C++ Compiler, Version 5.0, 5.2, or 5.3
- `WATCOM` for the Watcom C/C++ Compiler
- `MSVCDIR` for Microsoft Visual C/C++, Version 5.0 or 6.0
- `MSDEVDIR` for Microsoft Visual C/C++, Version 4.2

Next, `mex` searches the Windows Registry for compiler entries. Note that Watcom does not add an entry to the registry. Digital Fortran does not use an environment variable; `mex` only looks for it in the registry.

Internal Error When Using mex –setup (PC)

Some antivirus software packages such as Cheyenne AntiVirus and Dr. Solomon may conflict with the `mex –setup` process. If you get an error message during `mex –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 `mex –setup`. After you have successfully run the `setup` option, you can re-enable your antivirus software.

Verification of mex Fails

If none of the previous solutions addresses your difficulty with `mex`, contact Technical Support at The MathWorks at `support@mathworks.com` or 508 647-7000.

Troubleshooting the Compiler

One problem that might occur when you try to use the Compiler 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 MEX-File

If you experience other problems with the MATLAB Compiler, contact Technical Support at The MathWorks at `support@mathworks.com` or 508 647-7000.

3

Getting Started with MEX-Files

This chapter gets you started compiling M-files with the MATLAB Compiler. By the end of this chapter, you should know how to:

- Compile M-files into MEX-files
- Invoke MEX-files
- Generate Simulink S-functions

This chapter also lists the limitations and restrictions of the MATLAB Compiler.

A Simple Example

Sierpinski Gasket Example

Consider an M-file function called `gasket.m`.

```
function theImage = gasket(numPoints)
%GASKET An image of a Sierpinski Gasket.
%   IM = GASKET(NUMPOINTS)
%
%   Example:
%   x = gasket(50000);
%   imagesc(x);colormap([1 1 1;0 0 0]);
%   axis equal tight

%   Copyright (c) 1984-98 by The MathWorks, Inc
%   $Revision: 1.1 $  $Date: 1998/09/11 20:05:06 $

theImage = zeros(1000,1000);

corners = [866 1;1 500;866 1000];
startPoint = [866 1];
theRand = rand(numPoints,1);
theRand = ceil(theRand*3);

for i=1:numPoints
   startPoint = floor((corners(theRand(i),:)+startPoint)/2);
   theImage(startPoint(1),startPoint(2)) = 1;
end
```

How the Function Works

This function determines the coordinates of a Sierpinski Gasket using an Iterated Function System algorithm. The function starts with three points that define a triangle, and starting at one of these points, chooses one of the remaining points at random. A dot is placed at the midpoint of these two points. From the new point, a dot is placed at the midpoint between the new point and a point randomly selected from the original points. This process continues and eventually leads to an approximation of a curve.

The curve can be graphed in many ways. Sierpinski's method is:

- Start with a triangle and from it remove a triangle that is one-half the height of the original and inverted. This leaves three triangles.
- From each of the remaining three triangles, remove a triangle that is one-fourth the height of these new triangles and inverted. This leaves nine triangles.
- The process continues and at infinity the surface area becomes zero and the length of the curve is infinite.

`gasket.m` is a good candidate for compilation because it contains a loop. The overhead of the `for` loop command is relatively high compared to the cost of the loop body. M-file programmers usually try to avoid loops containing scalar operations because loops run relatively slowly under the MATLAB interpreter.

To achieve a reasonable approximation of the Sierpinski Gasket, set the number of points to 50,000. To compute the coordinates and time the computation, you can use

```
tic; x = gasket(50000); toc
```

To display the figure, you can use

```
imagesc(x); colormap([1 1 1;0 0 0]);
axis equal tight
```

Invoking the M-File

To get a baseline reading, you can determine how long it takes the MATLAB interpreter to run `gasket.m`. The built-in MATLAB functions `tic` and `toc` are useful tools for measuring time.

```
tic; x = gasket(50000); toc
elapsed_time =
10.0740
```

On the Pentium Pro 200, the M-file took about 10 seconds of CPU time to calculate the first 50,000 points on the Sierpinski Gasket.

Note The timings listed in this book were recorded on a Pentium Pro 200 MHz PC running Microsoft Windows NT. In each case, the code was executed two times and the results of the second execution were captured for this book. All of the timings listed throughout this book are for reference purposes only. They are not absolute; if you execute the same example under the same conditions, your times will probably differ from these values. Use these values as a frame of reference only.

Compiling the M-File into a MEX-File

To create a MEX-file from this M-file, enter the `mcc` command at the MATLAB interpreter prompt:

```
mcc –x gasket
```

This `mcc` command generates:

- A file named `gasket.c` containing MEX-file C source code.
- A file named `gasket.h` containing the public information.
- A file named `gasket_mex.c` containing the MEX-function interface (MEX wrapper).
- A MEX-file named `gasket.mex`. (The actual filename extension of the executable MEX-file varies depending on your platform, e.g., on the PC the file is named `gasket.dll`.)

`mcc` automatically invokes `mex` to create `gasket.mex` from `gasket.c`. The `mex` utility encapsulates the appropriate C compiler and linker options for your system.

This example uses the –x macro option to create the MEX-file. For more information on this Compiler option as well as the other options, see "mcc (Compiler 2.0)" in Chapter 6. For more information on the files that the Compiler generates, see Chapter 5, "Controlling Code Generation."

Invoking the MEX-File

Invoke the MEX-file version of `gasket` from the MATLAB interpreter the same way you invoke the M-file version.

```
tic; x = gasket(50000); toc
```

MATLAB runs the MEX-file version (`gasket.mex`) rather than the M-file version (`gasket.m`). Given an M-file and a MEX-file with the same root name (`gasket`) in the same directory, the MEX-file takes precedence.

This produces:

```
elapsed_time =
     8.0410
```

The MEX-file runs about 20% faster than the M-file version. To display the Sierpinski Gasket, use

```
imagesc(x); colormap([1 1 1;0 0 0]);
axis equal tight
```

Figure 3-1 shows the results.

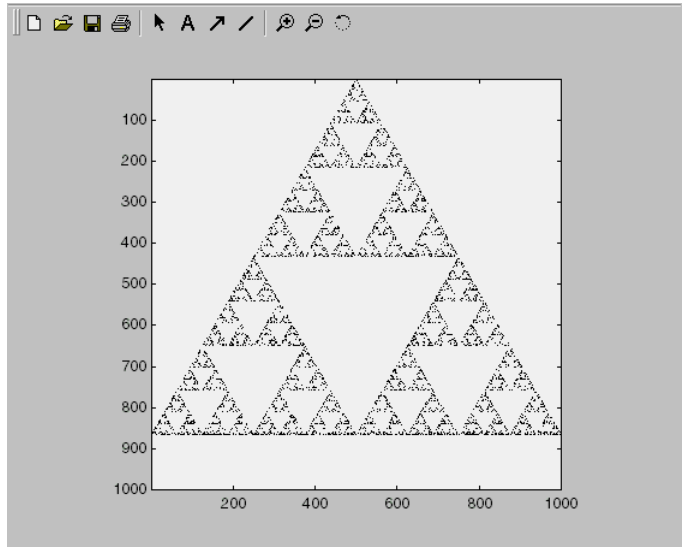


Figure 3-1: The Sierpinski Gasket for 50,000 Points

Compiler Options

The MATLAB Compiler uses a family of options, also called option flags, to control the functionality of the Compiler. Chapter 6 includes a complete description of the Compiler 2.0 options in the section, “mcc (Compiler 2.0).” Throughout this book you will see how these options are used with the Compiler to perform various tasks.

One particular set of Compiler options, macros, are particularly useful for performing straightforward compilations.

Macros

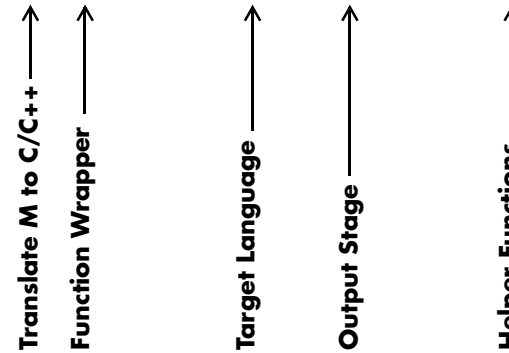
Macro options provide a simplified approach to compilation. Instead of manually grouping several options together to perform a particular type of compilation, you can use one simple option to quickly accomplish basic compilation tasks.

Note Macro options are intended to simplify the more common compilation tasks. You can always use individual options to customize the compilation process to satisfy your particular needs.

Table 3-1 shows the complete set of macro options available in Compiler 2.0.

Table 3-1: Macro Options

Macro Option	Creates	Option Equivalence
`-m`	Stand-alone C application	`-t -W main -L C -T link:exe -h`
`-p`	Stand-alone C++ application	`-t -W main -L Cpp -T link:exe -h`
`-x`	MEX-function	`-t -W mex -L C -T link:mex`
`-S`	Simulink S-function	`-t -W simulink -L C -T link:mex`



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:

```
-t -W main -L C -T link:exe -h
```

Table 3-2 shows the five options that compose the –m macro and the information that they provide to the Compiler.

Table 3-2: The –m Macro

Option	Function
–t	Translate M code to C/C++ code.
–W main	Produce a wrapper file suitable for a stand-alone application.
–L C	Generate C code as the target language.
–T link:exe	Create an executable as the output.
–h	Automatically, find and compile helper functions included in the source M-file.

Command Line Syntax

As you work with the Compiler, you will encounter situations where you must provide the Compiler with several options. You may specify one or more option flags to mcc at the command prompt.

Most option flags have a one-letter name. You can list options separately on the command line, for example:

```
mcc –m –g myfun
```

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 –m –A full myfun       % Options listed separately
mcc –mA full myfun         % Options combined, A option last
```

This format is *not* valid:

```
mcc –Am full myfun         % Options combined, A option not last
```

In cases where you have more than one option that take arguments, you can include only 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.

Note The `–V1.2` option cannot be combined with other options; it must stand by itself. For example, you cannot use

```
mcc –V1.2ir myfun
```

You would use

```
mcc –V1.2 –ir myfun
```

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,

```
mcc –m –W none test.m
```

is equivalent to

```
mcc –t –W main –L C –T link:exe –h –W none test.m
```

In the second case, 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 If you combine options on the command line (e.g., macros and regular options), they can both affect the same settings and can therefore override each other depending on their order in the command line. In these cases, the options are resolved from left to right as discussed in this section.

Limitations and Restrictions

MATLAB Code

This version of the MATLAB Compiler supports almost all of the functionality of MATLAB. However, there are some limitations and restrictions that you should be aware of. Although this version of the MATLAB Compiler cannot compile the following, a future version will be able to compile them.

- Script M-files (See page 3-17 for further details.)
- Functions that are only MEX functions
- M-files that use objects

The Compiler cannot compile:

- M-files containing `eval` or `input`. These functions create and use internal variables that only the MATLAB interpreter can handle.
- Inline functions because they are implemented using `eval`.
- Built-in MATLAB functions (functions such as `eig` have no M-file, so they can't be compiled). Note, however, that most of these functions are available to you because they are in the MATLAB Math Built-in Library (`libmatlb`).
- Calls to `load` or `save` that do not specify the names of the variables to load or save and do not specify a result variable for `load`. The `load` and `save` functions *are* supported in compiled code for lists of variables only. For example, this is acceptable:

```
load( filename, 'a', 'b', 'c' );      % This is OK and loads the
                                      % values of a, b, and c from
                                      % the file.
x = load(filename);
```

  However, this is *not* acceptable:

```
load( filename, var1, var2, var3 );  % This is not allowed.
```

  There is *no* support for the `load` and `save` options –`ascii` and –`mat`, and the variable wildcard (`*`).

Stand-Alone Applications

The restrictions and limitations noted in the previous section also apply to stand-alone applications. The functions in Table 3-3 are supported in MEX-mode, but are not supported in stand-alone mode.

Note You cannot call any Handle Graphics functions unless you have the optional Graphics Library installed.

Table 3-3: Unsupported Functions in Stand-Alone Mode

add_block	add_line	cd	cholinc
clc	close_system	dbclear	dbdown
dbquit	dbstack	dbstatus	dbstep
dbstop	dbtype	dbup	delete_block
delete_line	dir	echo	exist
get_param	inferiorto	inmem	int8
int16	int32	luinc	mfile2struct
new_system	open_system	pause	set_param
sim	simget	simset	single
sldebug	superiorto	system_dependent	type
uint8	uint16	uint32	which

In addition, stand-alone applications cannot access:

- Calls to MEX-file functions because the MEX-file functions require MATLAB to be running, which is not the case with stand-alone applications
- Simulink functions

Although the MATLAB Compiler *can* compile M-files that call these functions, the MATLAB C/C++ Math library does not support them. Therefore, unless you

write your own versions of the unsupported routines, when you run the executable, you will get a run-time error.

Generating Simulink S-Functions

You can use the MATLAB Compiler to generate Simulink C MEX S-functions. This allows you to speed up Simulink models that contain MATLAB M-code that is referenced from a MATLAB Fcn block.

For more information about Simulink in general, see the *Using Simulink* manual. For more information about Simulink S-functions, see the *Writing S-Functions* book.

Simulink-Specific Options

By using Simulink-specific options with the MATLAB Compiler, you can generate a complete S-function that is compatible with the S-Function block. The Simulink-specific options are –S, –u, and –y. Using any of these options with the MATLAB Compiler causes it to generate code that is compatible with Simulink.

Using the -S Option

The simplest S-function that the MATLAB Compiler can generate is one with a dynamically sized number of inputs and outputs. That is, you can pass any number of inputs and outputs in or out of the S-function. Both the MATLAB Fcn block and the S-Function block are single-input, single-output blocks. Only one line can be connected to the input or output of these blocks. However, each line may be a vector signal, essentially giving these blocks multi-input, multi-output capability. To generate a C language S-function of this type from an M-file, use the –S option:

```
mcc –S mfilename
```

Note The MATLAB Compiler option that generates a C language S-function is a capital S (–S).

The result is an S-function described in the following files:

```
mfilename.c
mfilename.h
mfilename_simulink.c
mfilename.ext        (where ext is the MEX-file extension for your
                      platform, e.g., dll for Windows)
```

Using the -u and -y Options

Using the –S option by itself will generate code suitable for most general applications. However, if you would like to exert more control over the number of valid inputs or outputs for your function, you should use the –u and/or –y options. These options specifically set the number of inputs (u) and the number of outputs (y) for your function. If either –u or –y is omitted, the respective input or output will be dynamically sized.

```
mcc –S –u 1 –y 2 mfilename
```

In the above line, the S-function will be generated with an input vector whose width is 1 and an output vector whose with is 2. If you were to connect the referencing S-function block to signals that do not correspond to the correct number of inputs or outputs, Simulink will generate an error when the simulation starts.

Specifying S-Function Characteristics

Sample Time

Similar to the MATLAB Fcn block, the automatically generated S-function has an inherited sample time.

Data Type

The input and output vectors for the Simulink S-function must be double-precision vectors or scalars. You must ensure that the variables you use in the M-code for input and output are also double-precision values.

Note Simulink S-functions that are generated via the –S option of the Compiler are not currently compatible with Real-Time Workshop®. They can, however, be used to rapidly prototype code in Simulink.

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. 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 nor can it compile a function M-file that calls a script.

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 matrix 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 t = houdini
m = magic(4); % Assign 4x4 matrix 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.mex` no longer creates variable `m`

in the MATLAB workspace. If it is important to have `m` accessible from the MATLAB workspace, you can change the beginning of the function to:

```
function [m,t] = houdini;
```

4

Stand-Alone Applications

Introduction

Note You must have the optional MATLAB C/C++ Math Library installed on your system if you want to create stand-alone applications.

This chapter explains how to use the MATLAB Compiler to code and build stand-alone applications. The first part of the chapter concentrates on using the `mbuild` script to build stand-alone applications and the second part concentrates on the coding of the applications. Stand-alone applications run without the help of the MATLAB interpreter. In fact, stand-alone applications *run* even if MATLAB is not installed on the system. However, stand-alone applications *do* require the run-time shared libraries. The specific shared libraries required for each platform are listed within the following sections.

Differences Between MEX-Files and Stand-Alone Applications

MEX-files and stand-alone applications differ in these respects:

- MEX-files run in the same process space as the MATLAB interpreter. When you invoke a MEX-file, the MATLAB interpreter dynamically links in the MEX-file.
- Stand-alone C or C++ applications run independently of MATLAB.

Stand-Alone C Applications

To build stand-alone C applications as described in this chapter, MATLAB, the MATLAB Compiler, a C compiler, and the MATLAB C/C++ Math Library must be installed on your system.

Note The MATLAB Compiler will compile calls to MEX-files in stand-alone mode, but you must provide suitable definitions for these functions so that the application can link properly.

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

The MATLAB Compiler translates input M-files into C source code suitable for your own stand-alone applications. After compiling this C source code, the resulting object file is linked with the object libraries:

- The MATLAB M-File Math Library (`libmmfile`), which contains compiled versions of most MATLAB M-file math routines.
- The MATLAB Math Built-In Library (`libmatlb`), which contains compiled versions of most MATLAB built-in math routines.
- The MATLAB Array Access and Creation Library (`libmx`), which contains the array access routines.
- The MATLAB Utilities Library (`libut`), which contains the utility routines used by various components in the background.
- The MATLAB Graphics Library (`libsgl`), if applicable.
- The ANSI C Math Library.

The `libmmfile` and `libmatlb` libraries come with the MATLAB C/C++ Math Library product, the `libmx` and `libut` libraries come with MATLAB, and the `libsgl` library comes with the MATLAB C/C++ Graphics Library product. The last library comes with your ANSI C compiler.

Note If you attempt to compile `.m` files to produce stand-alone applications and you do not have the MATLAB C/C++ Math Library installed, the system will not be able to find the appropriate libraries and the linking will fail. Also, if you do not have the MATLAB C/C++ Graphics Library installed, the MATLAB Compiler will generate run-time errors if the graphics functions are called.

Stand-Alone C++ Applications

To build stand-alone C++ applications, MATLAB, the MATLAB Compiler, a C++ compiler, and the MATLAB C/C++ Math Library must be installed on your system.

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

The MATLAB Compiler, when invoked with the appropriate option flag (`–p` or `–L Cpp`), translates input M-files into C++ source code suitable for your own stand-alone applications. After compiling this C++ source code, the resulting object file is linked with the above C object libraries and the MATLAB C++ Math Library (`libmatpp`), which contains C++ versions of MATLAB functions. The `mbuild` script links the MATLAB C++ Math Library first, then the C object libraries listed above.

Note On the PC, the MATLAB C++ Math Library is static because the different PC compiler vendors use different C++ name mangling algorithms.

Building Stand-Alone C/C++ Applications

This section explains how to build stand-alone C and C++ applications on UNIX systems and PCs running Microsoft Windows.

This section begins with a summary of the steps involved in building stand-alone C/C++ applications, including the `mbuild` script, which helps automate the build process, and then describes platform-specific issues for both supported platforms.

Note This chapter assumes that you have installed and configured the MATLAB Compiler.

Overview

On both operating systems, the steps you use to build stand-alone C and C++ applications are:

1. Verify that `mbuild` can create stand-alone applications.
2. Verify that the MATLAB Compiler can link object files with the proper libraries to form a stand-alone application.

Figure 4-1 shows the sequence on both platforms. The sections following the flowchart provide more specific details for the individual platforms.

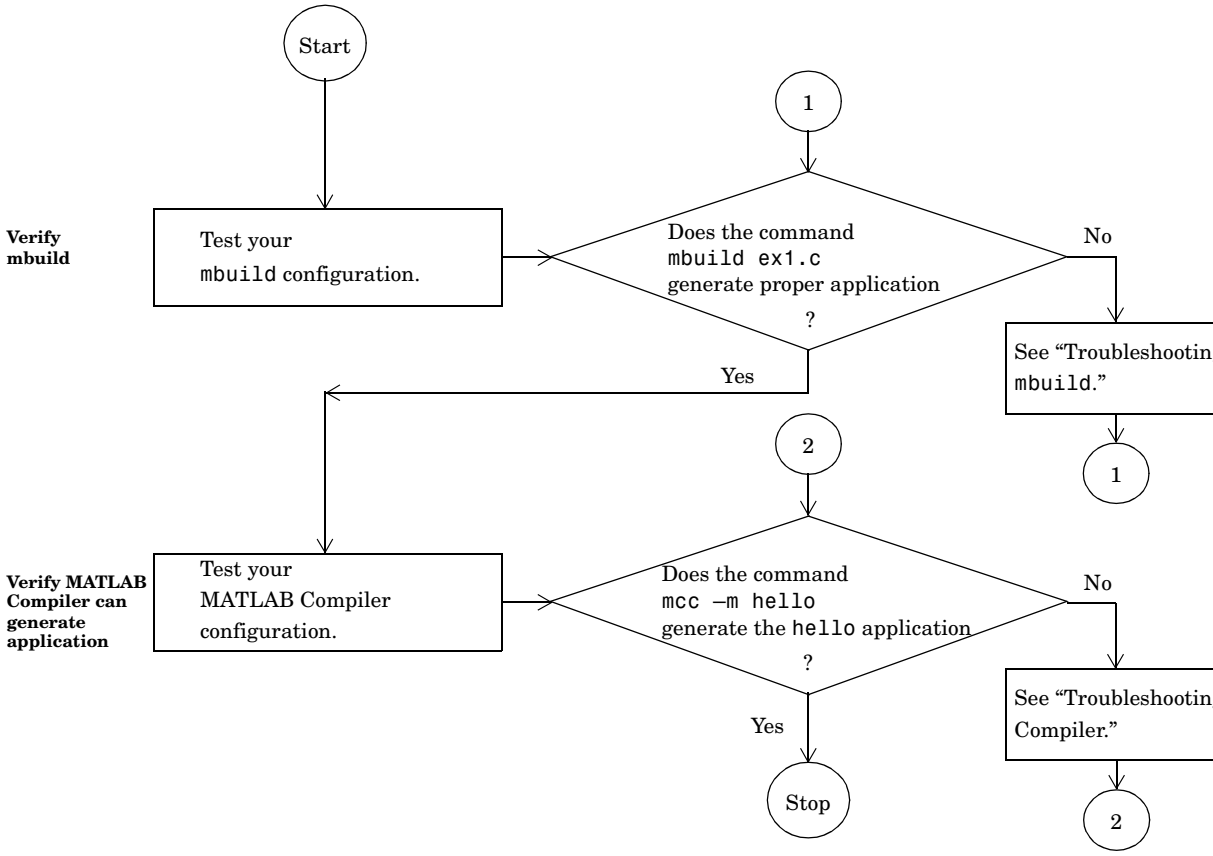


Figure 4-1: Sequence for Creating Stand-Alone C/C++ Applications

Packaging Stand-Alone Applications

To distribute a stand-alone application, you must include the application's executable as well as the shared libraries with which the application was linked. The necessary shared libraries vary by platform and are listed within the individual UNIX and Windows sections that follow.

Getting Started

Introducing mbuild

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
- Switch between C and C++ development
- Build your application

The MATLAB Compiler (`mcc`) automatically invokes `mbuild` under certain conditions. In particular, `mcc -m` or `mcc -p` invokes `mbuild` to perform compilation and linking. See the section, "mcc (Compiler 2.0)," in Chapter 6 for complete details on which Compiler options you should use in order to use the `mbuild` script.

If you do not want `mcc` to invoke `mbuild` automatically, you can use the `-c` option. For example, `mcc -mc filename`.

Compiler Options Files

Options files contain the required compiler and linker settings for your particular C or C++ compiler. The MathWorks provides options files for every supported C or C++ compiler. The options file for UNIX is `mbuildopts.sh`; Table 4-4 contains the options files for the PC.

Much of the information on options files in this chapter is provided for those users who may need to modify an options file to suit their specific needs. Many users never have to be concerned with how the options files work.

Note Before you can create stand-alone C or C++ applications, you must install the MATLAB C/C++ Math Library on your system. The MATLAB C/C++ Math Library is a separately sold product available from The MathWorks, Inc.

If you are developing C++ applications, make sure your C++ compiler supports the templates features of the C++ language. If it does not, you may be unable to use the MATLAB C/C++ Math Library.

Building Stand-Alone Applications on UNIX

This section explains how to compile and link C or C++ source code into a stand-alone UNIX application.

Configuring for C or C++

`mbuild` deduces the type of files you are compiling by the file extension. If you include both C and C++ files, `mbuild` uses the C++ compiler and the MATLAB C++ Math Library. If `mbuild` cannot deduce from the file extensions whether to compile C or C++, `mbuild` invokes the C compiler. The MATLAB Compiler generates only `.c` and `.cpp` files. Table 4-1 shows the supported file extensions.

Table 4-1: UNIX File Extensions for mbuild

Language	Extension(s)
C	`.c`

C++	`.cpp`
	`.C`
	`.cxx`
	`.cc`

Note You can override the language choice that is determined from the extension by using the `–lang` option of `mbuild`. For more information about this option, as well as all of the other `mbuild` options, see the `mbuild` reference page in Chapter 6.

Locating Options Files

`mbuild` locates your options file by searching the following:

- The current directory
- `$HOME/matlab`
- `<matlab>/bin`

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

Preparing to Compile

Note Refer to "Supported ANSI C and C++ UNIX Compilers" in Chapter 2 for information about supported compilers and important limitations.

Using the System Compiler

If the MATLAB Compiler and your supported C or C++ compiler are installed on your system, you are ready to create C or C++ stand-alone applications. To create a stand-alone C application, you can simply enter

```
mbuild filename.c
```

This simple method works for the majority of users. Assuming `filename.c` contains a `main` function, this example uses the system's compiler as your default compiler for creating your stand-alone application. If you are a user who does not need to change C or C++ compilers, or you do not need to modify your compiler options files, you can skip ahead in this section to "Verifying mbuild." If you need to know how to change the options file or select a different compiler, continue with this section.

Changing Compilers

Changing the Default Compiler. You need to use the `setup` option if you want to change any options or link against different libraries. At the UNIX prompt type:

```
mbuild –setup
```

The `setup` option creates a user-specific options file for your ANSI C or C++ compiler. Executing `mbuild –setup` presents a list of options files currently included in the `bin` subdirectory of MATLAB.

```
mbuild –setup

Using the 'mbuild –setup' command selects an options file that is
placed in ~/matlab 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.
```

```
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: /matlab/bin/mbuildopts.sh :
         Build and link with MATLAB C/C++ Math Library
```

If there is more than one options file, you can select the one you want by entering its number and pressing **Return**. If there is only one options file available, it is automatically copied to your MATLAB directory if you do not already have an `mbuild` options file. If you already have an `mbuild` options file, you are prompted to overwrite the existing one.

Note The options file is stored in the MATLAB subdirectory of your home directory. This allows each user to have a separate `mbuild` configuration.

Using the `setup` option sets your default compiler so that the new compiler is used everytime you use the `mbuild` script.

Modifying the Options File. Another use of the `setup` option is if you want to change your options file settings. 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.

If you need to change the options that `mbuild` passes to your compiler or linker, you must first run

```
mbuild –setup
```

which copies a master options file to your local MATLAB directory, typically `$HOME/matlab/mbuildopts.sh`.

If you need to see which options `mbuild` passes to your compiler and linker, use the verbose option, `–v`, as in

```
mbuild –v filename1 [filename2 …]
```

to generate a list of all the current compiler settings. To change the options, use an editor to make changes to your options file, which is in your local `matlab` directory. Your local `matlab` directory is a user-specific, MATLAB directory in your individual home directory that is used specifically for your individual options files. You can also embed the settings obtained from the verbose option of `mbuild` into an integrated development environment (IDE) or makefile that you need to maintain outside of MATLAB. Often, however, it is easier to call `mbuild` from your makefile. See your system documentation for information on writing makefiles.

Note Any changes made to the local options file will be overwritten if you execute `mbuild –setup`. To make the changes persist through repeated uses of `mbuild –setup`, you must edit the master file itself, `<matlab>/bin/mbuildopts.sh`.

Temporarily Changing the Compiler. To temporarily change your C or C++ compiler, use the `–f` option, as in

```
mbuild –f <file> …
```

The `–f` option tells the `mbuild` script to use the options file, `<file>`. If `<file>` is not in the current directory, then `<file>` must be the full pathname to the desired options file. Using the `–f` option tells the `mbuild` script to use the specified options file for the current execution of `mbuild` only; it does not reset the default compiler.

Verifying mbuild

There is C source code for an example `ex1.c` included in the `<matlab>/extern/examples/cmath` directory, where `<matlab>` represents the top-level directory where MATLAB is installed on your system. To verify that `mbuild` is properly configured on your system to create stand-alone applications, copy `ex1.c` to your local directory and type `cd` to change to that directory. Then, at the MATLAB prompt, enter:

```
mbuild ex1.c
```

This should create the file called `ex1`. Stand-alone applications created on UNIX systems do not have any extensions.

Locating Shared Libraries

Before you can run your stand-alone application, you must tell the system where the API and C shared libraries reside. This table provides the necessary UNIX commands depending on your system's architecture.

Architecture	Command
HP700	`setenv SHLIB_PATH <matlab>/extern/lib/hp700:$SHLIB_PATH`
IBM RS/6000	`setenv LIBPATH <matlab>/extern/lib/ibm_rs:$LIBPATH`
All others	`setenv LD_LIBRARY_PATH <matlab>/extern/lib/<arch>:$LD_LIBRARY_PATH`
	where: `<matlab>` is the MATLAB root directory `<arch>` is your architecture (i.e., `alpha`, `lnx86`, `sgi`, `sgi64`, or `sol2`)

It is convenient to place this command in a startup script such as `~/.cshrc`. Then the system will be able to locate these shared libraries automatically, and you will not have to re-issue the command at the start of each login session.

Note On all UNIX platforms, the Compiler library is shipped as a shared object (`.so`) file or shared library (`.sl`). Any Compiler-generated, stand-alone application must be able to locate the C/C++ libraries along the library path environment variable (`SHLIB_PATH`, `LIBPATH`, or `LD_LIBRARY_PATH`) in order to be found and loaded. Consequently, to share a Compiler-generated, stand-alone application with another user, you must provide all of the required shared libraries. For more information about the required shared libraries for UNIX, see "Distributing Stand-Alone UNIX Applications."

Running Your Application

To launch your application, enter its name on the command line. For example,

```
ex1
ans =

     1     3     5
     2     4     6


ans =

   1.0000 + 7.0000i   4.0000 +10.0000i
   2.0000 + 8.0000i   5.0000 +11.0000i
   3.0000 + 9.0000i   6.0000 +12.0000i
```

Verifying the MATLAB Compiler

There is MATLAB code for an example, `hello.m`, included in the `<matlab>/extern/examples/compiler` directory. To verify that the MATLAB Compiler can generate stand-alone applications on your system, type the following at the MATLAB prompt:

```
mcc –m hello.m
```

This command should complete without errors. To run the stand-alone application, `hello`, invoke it as you would any other UNIX application, typically by typing its name at the UNIX prompt. The application should run and display the message

```
Hello, World
```

When you execute the `mcc` command to link files and libraries, `mcc` actually calls the `mbuild` script to perform the functions.

Distributing Stand-Alone UNIX Applications

To distribute a stand-alone application, you must include the application's executable and the shared libraries against which the application was linked. This package of files includes:

- Application (executable)
- `libmmfile.`*`ext`*
- `libmatlb.`*`ext`*
- `libmat.`*`ext`*
- `libmx.`*`ext`*
- `libut.`*`ext`*
- `libsgl.`*`ext`* (if applicable)
- `libmatpp.`*`ext`* (This is necessary only if you are using the C++ Math Library.)

where `.`*`ext`* is

`.a` on IBM RS/6000; `.so` on Solaris, Alpha, Linux, and SGI; and `.sl` on HP 700.

For example, to distribute the `ex1` example for Solaris, you need to include `ex1`, `libmmfile.so`, `libmatlb.so`, `libmat.so`, `libmx.so`, `libut.so`, `libsgl.so` (if applicable), and `libmatpp.so`. Remember to locate the shared libraries along the `LD_LIBRARY_PATH` environment variable so that they can be found and loaded.

Installing C++ and Fortran Support

MATLAB users require access to both the C++ and Fortran run-time shared libraries. These are usually provided as part of the operating system installation. For Digital UNIX, however, the C++ shared libraries are part of the base installation package, but the Fortran shared libraries are on a separate disk called the "Associated Products CD." MATLAB users running under Digital UNIX should install both the C++ and Fortran run-time shared libraries.

Note If you distribute an application created with the Math Libraries on Digital UNIX, your users must have both the C++ and Fortran run-time shared libraries installed on their systems.

About the mbuild Script

The `mbuild` script supports various options that allow you to customize the building and linking of your code. Many users do not need to know any

additional details of the `mbuild` script; they use it in its simplest form. The information in Table 4-2 is provided for those users who require more flexibility with the tool. The `mbuild` syntax and options are:

```
mbuild [–options] filename1 [filename2 …]
```

Table 4-2: mbuild Options on UNIX

Option	Description
`–c`	Compile only; do not link.
`–D<name>[=<def>]`	Define C preprocessor macro `<name>` [as having value `<def>`.]
`–f <file>`	Use `<file>` to override the default options file; `<file>` is a full pathname if the options file is not in current directory.
`–g`	Build an executable with debugging symbols included.
`–h[elp]`	Help; prints a description of `mbuild` and the list of options.
`–I<pathname>`	Add `<pathname>` to the Compiler `include` search path.
`–l<file>`	Link against library `lib<file>`.
`–L<pathname>`	Include `<pathname>` in the list of directories to search for libraries.
`–lang <language>`	Override language choice implied by file extension. `<language>` = `c` for C (default) `cpp` for C++ This option is necessary when you use an unsupported file extension, or when you pass in all `.o` files and libraries.

Table 4-2: mbuild Options on UNIX (Continued)

Option	Description
`–link <target>`	Specify output type. `<target>` = `exe` for an executable (default) `shared` for shared library (See "Building Shared Libraries" later in this chapter.)
`<name>=<def>`	Override options file setting 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'`.
`–n`	No execute flag. This option causes the commands used to compile and link the target to display without executing them.
`–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 <name>`	Create an executable named `<name>`. (An appropriate executable extension is automatically appended.)
`–O`	Build an optimized executable.
`–setup`	Set up default options file. This option should be the only argument passed.

Table 4-2: mbuild Options on UNIX (Continued)

Option	Description
–U<name>	Undefine C preprocessor macro <name>.
–v	Verbose; print all compiler and linker settings.

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 "mcc (Compiler 2.0)" in Chapter 6.

Building Stand-Alone Applications on PCs

This section explains how to compile and link the C/C++ code generated from the MATLAB Compiler into a stand-alone Windows application.

Configuring for C or C++

`mbuild` determines whether to compile in C or C++ by examining the type of files you are compiling. Table 4-3 shows the file extensions that `mbuild` interprets as indicating C or C++ files.

Table 4-3: Windows File Extensions for mbuild

Language	Extension(s)
C	`.c`

C++	`.cpp`
	`.cxx`
	`.cc`

- If you include both C and C++ files, `mbuild` uses the C++ compiler and the MATLAB C++ Math Library.
- If `mbuild` cannot deduce from the file extensions whether to compile in C or C++, `mbuild` invokes the C compiler.

Note You can override the language choice that is determined from the extension by using the `–lang` option of `mbuild`. For more information about this option, as well as all of the other `mbuild` options, see Table 4-5.

Locating Options Files

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

- The current directory
- The `user profile` directory (For more information about this directory, see the section, “The User Profile Directory Under Windows,” in Chapter 2.)
- `<matlab>\bin`

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.

Preparing to Compile

Note Refer to "Supported ANSI C and C++ PC Compilers" in Chapter 2 for information about supported compilers and important limitations.

Choosing a Compiler

Systems with Exactly One C/C++ Compiler. If the MATLAB Compiler and your supported C or C++ compiler are installed on your system, you are ready to create C or C++ stand-alone applications. On systems where there is exactly one C or C++ compiler available to you, the mbuild utility automatically configures itself for the appropriate compiler. So, for many users, to create a C or C++ stand-alone applications, you can simply enter

```
mbuild filename.c
```

This simple method works for the majority of users. Assuming filename.c contains a main function, this example uses your installed C or C++ compiler as your default compiler for creating your stand-alone application. If you are a user who does not need to change compilers, or you do not need to modify your compiler options files, you can skip ahead in this section to "Verifying mbuild." If you need to know how to change the options file or select a different compiler, continue with this section.

Systems with More than One C/C++ Compiler. On systems where there is more than one C or C++ compiler, the mbuild utility lets you select which of the compilers you want to use. Once you choose your C or C++ compiler, that compiler becomes your default compiler and you no longer have to select one when you compile your stand-alone applications.

For example, if your system has both the Borland and Watcom compilers, when you enter for the first time

```
mbuild filename.c
```

you are asked to select which compiler to use.

```
mbuild has detected the following compilers on your machine:

[1] : Borland compiler in T:\Borland\BC.500
[2] : WATCOM compiler in T:\watcom\c.106

[0] : None

Please select a compiler. This compiler will become the default:
```

Select the desired compiler by entering its number and pressing **Return**. You are then asked to verify your information.

Changing Compilers

Changing the Default Compiler. To change your default C or C++ compiler, you select a different options file. You can do this at anytime by using the `setup` command.

This example shows the process of changing your default compiler to the Microsoft Visual C/C++ Version 6.0 compiler.

```
mbuild –setup

Please choose your compiler for building stand-alone MATLAB
applications.

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

Choose your C/C++ compiler:
[1] Borland C/C++           (version 5.0, 5.2, or 5.3)
[2] Microsoft Visual C/C++  (version 4.2, 5.0, or 6.0)
[3] Watcom C/C++            (version 10.6 or 11)

[0] None

Compiler: 2
```

```
Choose the version of your C/C++ compiler:
[1] Microsoft Visual C/C++ 4.2
[2] Microsoft Visual C/C++ 5.0
[3] Microsoft Visual C/C++ 6.0

version: 3

Your machine has a Microsoft Visual C/C++ compiler located at
D:\Program Files\DevStudio6.
Do you want to use this compiler [y]/n? y

Please verify your choices:

Compiler: Microsoft Visual C/C++ 6.0
Location: D:\Program Files\DevStudio6

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

The default options file:
"C:\WINNT\Profiles\username
\Application Data\MathWorks\MATLAB\compopts.bat" is being
updated...
```

If the specified compiler cannot be located, you are given the message:

```
The default location for compiler-name is directory-name,
but that directory does not exist on this machine.
Use directory-name anyway [y]/n?
```

Using the `setup` option sets your default compiler so that the new compiler is used everytime you use the `mbuild` script.

Modifying the Options File. Another use of the `setup` option is if you want to change your options file settings. 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.

The `setup` option copies the appropriate options file to your `user profile` directory. To make your user-specific changes to the options file, you edit your copy of the options file in your `user profile` directory to correspond to your specific needs and save the modified file. This sets your default compiler's

options file to your specific version. Table 4-4 lists the names of the PC options files included in this release of MATLAB.

If you need to see which options `mbuild` passes to your compiler and linker, use the verbose option, –v, as in

```
mbuild –v filename1 [filename2 …]
```

to generate a list of all the current compiler settings used by `mbuild`. To change the options, use an editor to make changes to your options file that corresponds to your compiler. You can also embed the settings obtained from the verbose option into an integrated development environment (IDE) or makefile that you need to maintain outside of MATLAB. Often, however, it is easier to call `mbuild` from your makefile. See your system documentation for information on writing makefiles.

Note Any changes that you make to the local options file `compopts.bat` will be overwritten the next time you run `mbuild –setup`. If you want to make your edits persist through repeated uses of `mbuild –setup`, you must edit the master file itself. The master options files are also located in `<matlab>\bin`.

Table 4-4: Compiler Options Files on the PC

Compiler	Master Options File
Borland C/C++, Version 5.0	`bcccompp.bat`
Borland C/C++, Version 5.2	`bcc52compp.bat`
Borland C/C++, Version 5.3	`bcc53compp.bat`
Microsoft Visual C/C++, Version 4.2	`msvccompp.bat`
Microsoft Visual C/C++, Version 5.0	`msvc50compp.bat`
Microsoft Visual C/C++, Version 6.0	`msvc60compp.bat`
Watcom C/C++, Version 10.6	`watccompp.bat`
Watcom C/C++, Version 11	`wat11ccompp.bat`

Combining Customized C and C++ Options Files. The options files for `mbuild` have changed as of MATLAB 5.3 (Release 11) so that the same options file can be used to create both C and C++ stand-alone applications. If you have modified your own separate options files to create C and C++ applications, you can combine them into one options file.

To combine your existing options files into one universal C and C++ options file:

1. Copy from the C++ options file to the C options file all lines that set the variables `COMPFLAGS`, `OPTIMFLAGS`, `DEBUGFLAGS`, and `LINKFLAGS`.

2. In the C options file, within just those copied lines from step 1, replace all occurrences of `COMPFLAGS` with `CPPCOMPFLAGS`, `OPTIMFLAGS` with `CPPOPTIMFLAGS`, `DEBUGFLAGS` with `CPPDEBUGFLAGS`, and `LINKFLAGS` with `CPPLINKFLAGS`.

This process modifies your C options file to be a universal C/C++ options file.

Temporarily Changing the Compiler. To temporarily change your C or C++ compiler, use the `–f` option, as in

```
mbuild –f <file> …
```

The `–f` option tells the `mbuild` script to use the options file, `<file>`. If `<file>` is not in the current directory, then `<file>` must be the full pathname to the desired options file. Using the `–f` option tells the `mbuild` script to use the specified options file for the current execution of `mbuild` only; it does not reset the default compiler.

Verifying mbuild

There is C source code for an example, `ex1.c`, included in the `<matlab>\extern\examples\cmath` directory, where `<matlab>` represents the top-level directory where MATLAB is installed on your system. To verify that `mbuild` is properly configured on your system to create stand-alone applications, enter at the MATLAB prompt:

```
mbuild ex1.c
```

This should create the file called `ex1.exe`. Stand-alone applications created on Windows 95/98 or Windows NT always have the extension `exe`. The created application is a 32-bit MS-DOS console application.

Shared Libraries

All the libraries (WIN32 Dynamic Link Libraries, or DLLs) for MATLAB, the MATLAB Compiler, and the MATLAB Math Library are in the directory

```
<matlab>\bin
```

The `.DEF` files for the Microsoft and Borland compilers are in the `<matlab>\extern\include` directory. All of the relevant libraries for building stand-alone applications are WIN32 Dynamic Link Libraries. Before running a stand-alone application, you must ensure that the directory containing the DLLs is on your path. The directory must be on your operating system `$PATH` environment variable. On Windows 95, set the value in your `AUTOEXEC.BAT` file; on Windows NT, use the Control Panel to set it.

Running Your Application

You can now run your stand-alone application by launching it from the DOS command line. For example,

```
ex1
ans =

     1     3     5
     2     4     6


ans =

   1.0000 + 7.0000i   4.0000 +10.0000i
   2.0000 + 8.0000i   5.0000 +11.0000i
   3.0000 + 9.0000i   6.0000 +12.0000i
```

Verifying the MATLAB Compiler

There is MATLAB code for an example, `hello.m`, included in the `<matlab>\extern\examples\compiler` directory. To verify that the MATLAB Compiler can generate stand-alone applications on your system, type the following at the MATLAB prompt:

```
mcc –m hello.m
```

This command should complete without errors. To run the stand-alone application, `hello`, invoke it as you would any other Windows console application, by typing its name on the MS-DOS command line. The application should run and display the message `Hello, World`.

When you execute the `mcc` command to link files and libraries, `mcc` actually calls the `mbuild` script to perform the functions.

About the mbuild Script

The `mbuild` script supports various options that allow you to customize the building and linking of your code. Many users do not need to know any additional details of the `mbuild` script; they use it in its simplest form. The information in Table 4-5 is provided for those users who require more flexibility with the tool. The `mbuild` syntax and options are:

```
mbuild [–options] filename1 [filename2 …]
```

Table 4-5: mbuild Options on Windows

Option	Description
`@filename`	Replace `@filename` on the `mbuild` command line with the contents of `filename`. `filename` is a response file, i.e., a text file that contains additional command line options to be processed.
`–c`	Compile only; do not link.
`–D<name>`	Define C preprocessor macro `<name>`.
`–f <file>`	Use `<file>` as the options file; `<file>` is a full pathname if the options file is not in current directory.
`–g`	Build an executable with debugging symbols included.
`–h[elp]`	Help; prints a description of `mbuild` and the list of options.

Table 4-5: mbuild Options on Windows (Continued)

Option	Description
–I<pathname>	Add <pathname> to the Compiler include search path.
–lang <language>	Override language choice implied by file extension. <language> = c for C (default) cpp for C++ This option is necessary when you use an unsupported file extension, or when you pass in all .o files and libraries.
–link <target>	Specify output type. <target> = exe for an executable (default) shared for DLL
–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 <name>	Create an executable named <name>. (An appropriate executable extension is automatically appended.)
–O	Build an optimized executable.
–setup	Set up default options file. This option should be the only argument passed.
–U<name>	Undefine C preprocessor macro <name>.
–v	Verbose; print all compiler and linker settings.

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 "mcc (Compiler 2.0)" in Chapter 6.

Using an IDE

The MathWorks provides a special tool, VisualMATLAB, a Developer Studio add-in that lets you easily work within the Microsoft Visual C++ environment. For more information about VisualMATLAB, including how to download it from The MathWorks, contact Technical Support.

Distributing Stand-Alone Windows Applications

To distribute a stand-alone application, you must include the application's executable as well as the shared libraries against which the application was linked. This package of files includes:

- Application (executable)
- libmmfile.dll
- libmatlb.dll
- libmat.dll
- libmx.dll
- libut.dll
- libsgl.dll (if applicable)

For example, to distribute the Windows version of the ex1 example, you need to include ex1.exe, libmmfile.dll, libmatlb.dll, libmat.dll, libmx.dll, libut.dll, and libsgl.dll (if applicable). The DLLs must be on the system path. You must either install them in a directory that is already on the path or modify the PATH variable to include the new directory.

Building Shared Libraries

You can use `mbuild` to build C shared libraries on both UNIX and the PC. All of the `mbuild` options that pertain to creating stand-alone applications also pertain to creating C shared libraries. To create a C shared library, you use the option

```
–link shared
```

and specify one or more files with the `.exports` extension. The `.exports` files are text files that contain the names of the functions to export from the shared library, one per line. You can include comments in your code by beginning a line (first column) with `#` or a `*`. `mbuild` treats these lines as comments and ignores them. `mbuild` merges multiple `.exports` files into one master exports list.

For example, given `file2.exports` as:

```
times2
times3
```

and `file1.c` as:

```
int times2(int x)
{
     return 2 * x;
}

int times3(int x)
{
     return 3 * x;
}
```

The command

```
mbuild –link shared file1.c file2.exports
```

creates a shared library named `file1.`*`ext`*, where *`ext`* is the platform-dependent shared library extension. For example, on the PC, it would be called `file1.dll`. The shared library exports the symbols `times2` and `times3`.

Troubleshooting

Troubleshooting 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

On UNIX, 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 PCs running 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.

mbuild Not a Recognized Command

If `mbuild` is not recognized, verify that `<MATLAB>\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

```
mbuild ex1.c
```

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 (PC)

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 5.0, 5.2, or 5.3
- `WATCOM` for the Watcom C/C++ Compiler
- `MSVCDIR` for Microsoft Visual C/C++, Version 5.0 or 6.0
- `MSDEVDIR` for Microsoft Visual C/C++, Version 4.2

Next, `mbuild` searches the Windows registry for compiler entries. Note that Watcom does not add an entry to the registry.

Internal Error When Using mbuild –setup (PC)

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 re-enable 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` or 508 647-7000.

Troubleshooting the 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` or 508 647-7000.

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. Coding an application in M-files allows you to take advantage of MATLAB's interpretive development environment. Then, after getting the M-file version of your program working properly, compile the code and build it into a stand-alone application.

Note It is good practice to avoid manually modifying the C or C++ code that the MATLAB Compiler generates. If the generated C or C++ code is not to your liking, modify the M-file (and/or the compiler options) and then recompile. If you do edit the generated C or C++ code, remember that your changes will be erased the next time you recompile the M-file. For more information, see "Compiling MATLAB-Provided M-Files Separately" in this chapter and "Interfacing M-Code to C/C++ Code" in Chapter 5.

Consider a very simple application whose source code consists of two M-files, `mrank.m` and `main.m`. This example involves C code; you use a similar process (described below) for C++ code. In this example, the line `r = zeros(n,1)` preallocates memory to help the performance of the Compiler.

`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
```

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

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

Note All stand-alone C programs require a main routine as their entry point.

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

```
mcc –mc main mrank
```

The `–m` option flag 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 a C functions named `mlfMain` and `mlfMrank`. (The `–c` option flag inhibits invocation of `mbuild`.)

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 twice, use `mbuild` to compile the files with your ANSI C compiler, and link the code) by using the command

```
mcc –m main mrank
```

Figure 4-2 illustrates the process of building a stand-alone C application from two M-files. The commands to compile and link depend on the operating system being used. See the "Building Stand-Alone C/C++ Applications" section for details.

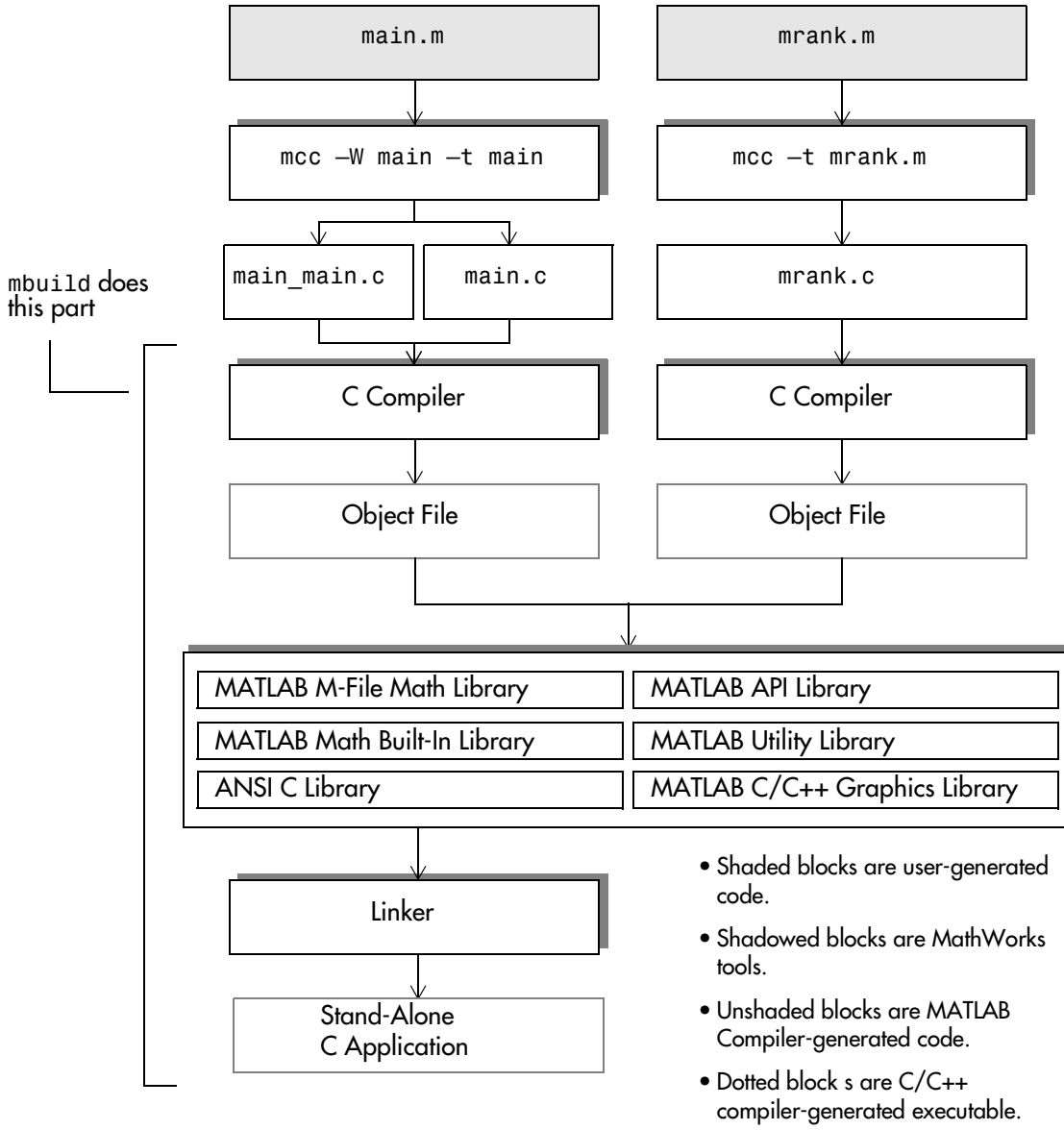


Figure 4-2: Building Two M-Files into a Stand-Alone C Application

For C++ code, add `-L cpp` to the previous commands, use a C++ compiler instead of a C compiler, and use the MATLAB C++ Math Library. See the *MATLAB C++ Math Library User's Guide* for details.

Alternative Ways of Compiling M-Files

The previous section showed how to compile `main.m` and `mrank.m` separately. This section explores two other ways of compiling M-files.

Note These two alternative ways of compiling M-files apply to C++ as well as to C code; the only difference is that you add `–L cpp` for C++.

Compiling MATLAB-Provided M-Files Separately

The M-file `mrank.m` contains a call to `rank`. The MATLAB Compiler translates the call to `rank` into a C call to `mlfRank`. The `mlfRank` routine is part of the MATLAB M-File Math Library. The `mlfRank` routine behaves in stand-alone applications exactly as the `rank` function behaves in the MATLAB interpreter. However, if this default behavior is not desirable, you can create your own version of `rank` or `mlfRank`.

One way to create a new version of `rank` is to copy MATLAB's own source code for `rank` and then to edit this copy. MATLAB implements `rank` as the M-file `rank.m` rather than as a built-in command. To see MATLAB's code for `rank.m`, enter:

```
type rank
```

Copy this code into a file named `rank.m` located in the same directory as `mrank.m` and `main.m`. Then, modify your version of `rank.m`. After completing the modifications, compile `rank.m`:

```
mcc –t rank
```

Compiling `rank.m` generates file `rank.c`, which contains a function named `mlfRank`. Then, compile the other M-files composing the stand-alone application:

```
mcc –t main.m                          (produces main.c)
mcc –t mrank.m                         (produces mrank.c)
mcc –W main main.m mrank.m rank.m      (produces main_main.c)
```

To compile and link all four C source code files (`main.c`, `rank.c`, `mrank.c`, and `main_main.c`) into a stand-alone application, use:

```
mcc main_main.c main.c rank.c mrank.c
```

The resulting stand-alone application uses your customized version of `mlfRank` rather than the default version of `mlfRank` stored in the MATLAB Toolbox Library.

Note On PCs running Windows, as well as SGI, SGI64, and IBM, if a function in the MATLAB Toolbox Library calls `mlfRank`, it will call the one found in the Library and *not* your customized version. We recommend that you call your version of `rank` something else, e.g., `myrank.m`.

Compiling mrank.m and rank.m as Helper Functions

Another way of building the `mrank` stand-alone application is to compile `rank.m` and `mrank.m` as helper functions to `main.m`. In other words, instead of invoking the MATLAB Compiler three separate times, invoke the MATLAB Compiler only once. For C:

```
mcc –m main rank
```

For C++:

```
mcc –p main rank
```

These commands create files containing the C or C++ source code. The macro options `–m` and `–p` automatically compile all helper functions.

Note The functions contained in the Compiler libraries will not be compiled when you use the `–h` option; they must be listed specifically on the command line.

Mixing M-Files and C or C++

Another 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.

This section presents three examples. One is a simple C example, and the other two are more sophisticated. All three examples illustrate how to mix M-files and C or C++ source code files.

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 the library sections in "Supported Executable Types" in Chapter 5.

Simple Example

The example below and the advanced example that follows involve mixing M-files and C code. See the "Advanced C++ Example" section for an example of 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.
3. Insert the call to the init routine in main.

To perform these steps, use:

```
mcc –t –W lib:Pkg –T link:exe mrank mrankp.c
```

The MATLAB Compiler generates C source code files named `mrank.c`, `Pkg.c`, and `Pkg.h`. This command invokes `mbuild` to compile the resulting Compiler-generated source files (`mrank.c`, `Pkg.c`, `Pkg.h`) with the existing C source file (`mrankp.c`) and links against the required libraries. For details, see the "Building Stand-Alone C/C++ Applications" section in this chapter.

The MATLAB Compiler provides two different versions of `mrankp.c` in the `<matlab>/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`.

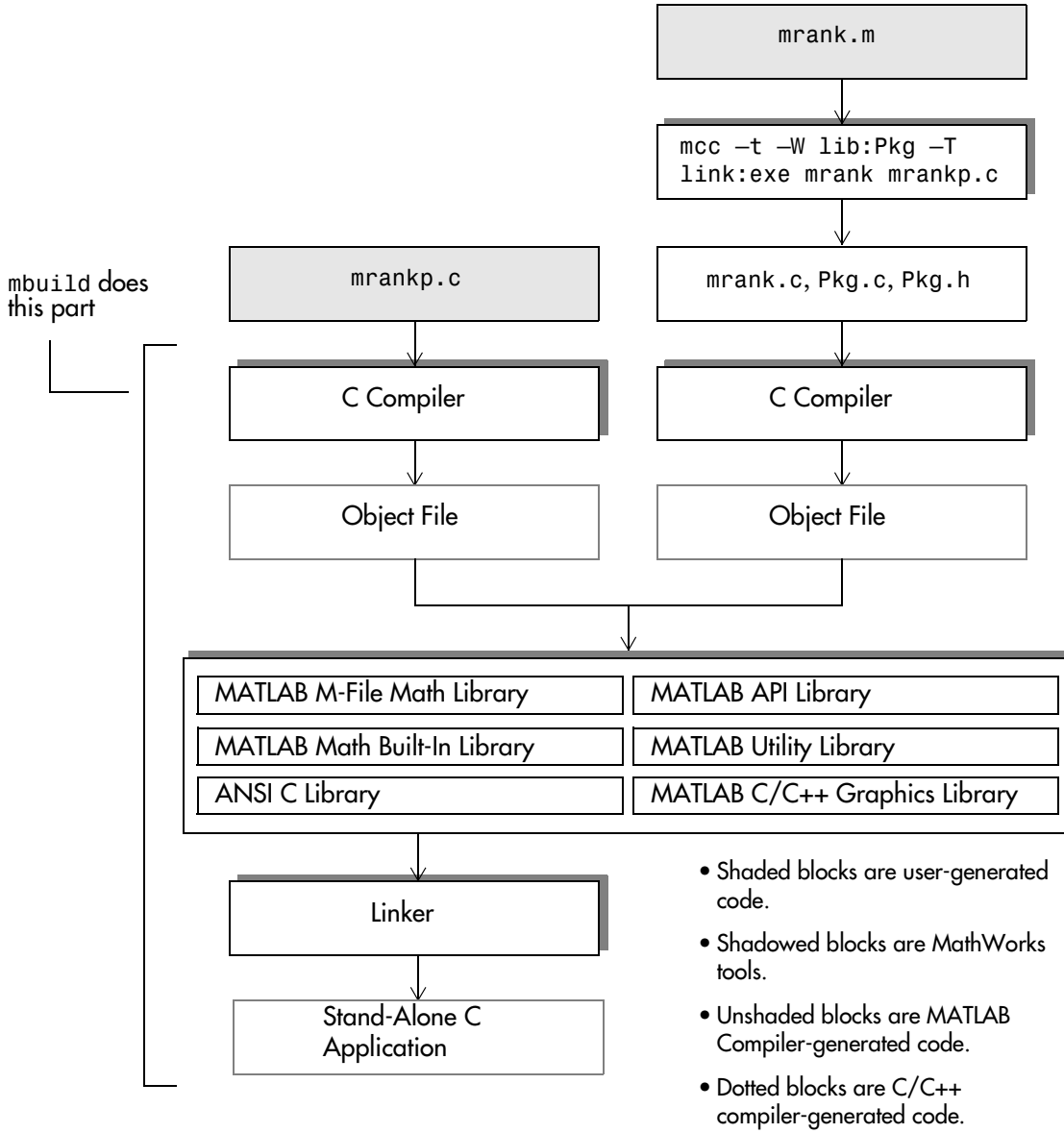


Figure 4-3: Mixing M-Files and C Code to Form a Stand-Alone Application

mrankp.c

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

```
#include <stdio.h>
#include <math.h>
#include "matlab.h"
#include "Pkg.h"     /* Include the Compiler-generated header
                        file */
int main( int argc, char **argv )
{
    mxArray *N;      /* Matrix containing n. */
    mxArray *R;      /* Result matrix. */
    int      n;      /* Integer parameter from command line. */

    PkgInitialize();  /* Call Pkg initialization */

    /* Get any command line parameter. */
    if (argc >= 2) {
        n = atoi(argv[1]);
    } else {
        n = 12;
    }

    /* Create a 1-by-1 matrix containing n. */
    N = mxCreateDoubleMatrix(1, 1, mxREAL);
    *mxGetPr(N) = n;

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

    /* Print the results. */
    mlfPrintMatrix(R);

    /* Free the matrices allocated during this computation. */
    mxDestroyArray(N);
    mxDestroyArray(R);
    PkgTerminate();   /* Call Pkg termination */

    return 0;
}
```

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 call the Compiler-generated library initialization function.

```
PkgInitialize();      /* Call Pkg initialization */
```

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

```
mxArray *mlfMrank(mxArray *n_rhs_)
```

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 *Application Program Interface Guide* for details on the `mxArray` data type.) To create and manipulate `mxArray *` variables in your C code, you should call the `mx` routines described in the *Application Program Interface Guide*. For example, to create a 1-by-1 `mxArray *` variable named `N` with real data, `mrankp` calls `mxCreateDoubleMatrix`:

```
N = mxCreateDoubleMatrix(1, 1, mxREAL);
```

Then, `mrankp` initializes the `pr` field of `N`. The `pr` field holds the real data of MATLAB `mxArray` variables. This code sets element 1,1 of `N` to whatever value you pass in at runtime:

```
*mxGetPr(N) = n;
```

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

```
R = mlfMrank(N);
```

`mlfMrank` returns a pointer to an `mxArray *` variable named `R`. The easiest way to display the contents of `R` is to call the `mlfPrintMatrix` convenience function:

```
mlfPrintMatrix(R);
```

`mlfPrintMatrix` is one of the many routines in the MATLAB Math Built-In Library, which is part of the MATLAB Math Library product.

Finally, `mrankp` must free the heap memory allocated to hold matrices and call the Compiler-generated termination function:

```
mxDestroyArray(N);
mxDestroyArray(R);
PkgTerminate();       /* Call Pkg termination */
```

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 "matlab.h"
#include "multpkg.h"/* Include Compiler-generated header file */

static void PrintHandler( const char *text )
{
    printf(text);
}

int main( )   /* Programmer written coded to call mlfMultarg */
{
#define ROWS  3
#define COLS  3
```

```
    mxArray *a, *b, *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;

    multpkgInitialize();  /* Call multpkg initialization */

    /* Install a print handler to tell mlfPrintMatrix how to
     * display its output.
     */
    mlfSetPrintHandler(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. */
    a = (mxArray *)mlfMultarg(&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);
    multpkgTerminate();   /* Call multpkg termination */
    return(0);
}
```

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

```
mcc –t –W lib:multpkg –T link:exe multarg 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 mxArray * mlfMultarg(mxArray * * b, mxArray * x,
                      mxArray * y);
extern void mlxMultarg(int nlhs, mxArray * plhs[], int nrhs,
                    mxArray * prhs[]);
```

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, COMPLEX);
memcpy(mxGetPr(x), x_pr, ROWS * COLS * sizeof(double));
memcpy(mxGetPi(y), y_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`:

```
a = (mxArray *)mlfMultarg(&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);
```

Advanced C++ Example

This example demonstrates how to use the MATLAB Compiler to produce a stand-alone C++ executable from a collection of M-files. You can also use these techniques to incorporate MATLAB functions into a pre-existing C++ program.

Most of the functions in the MATLAB Toolboxes are not present in the MATLAB C++ Math Library; however, the MATLAB Compiler makes them available to C++ programs. For example, the Image Processing Toolbox provides an edge-detection routine, `edge()`. This example uses `edge()` and other routines from the Image Processing Toolbox to perform edge detection in Microsoft Windows Bitmap files.

Note Version 4 of the MATLAB Image Processing Toolbox, *not* version 5, is the source for the image processing routines used in this example. The code included here is generated from MATLAB 4 code.

In this example, note the following:

- The MATLAB Compiler and the MATLAB C++ Math Library both use default arguments for optional inputs to functions. `mwArray::DIN` is the default input array used to initialize optional input arguments.
- For each accepted number of output arguments, the library provides an overloaded version of a function; the MATLAB Compiler does not. Output array pointers do not have a default value. However, you need to pass `NULL` to indicate a missing output argument.
- To compile a call to a function, the MATLAB Compiler must be able to determine the argument list for the function. If the function is built into the library, it must be present in the Compiler's internal table. If the function is in an M-file, that M-file must be on the MATLAB path.

  There are a number of built-in functions that, if called, will produce run-time errors in stand-alone code. See Table 3-3.

Algorithm for the Example

The example program consists of five steps:

1. Read a Microsoft Windows Bitmap file.
2. Convert the bitmap image into a grayscale image.
3. Perform edge-detection on the grayscale image.
4. Convert the resulting black-and-white Boolean mask into an image.
5. Write the image out to a new Microsoft Windows Bitmap file.

M-Files for the Example

These steps require direct calls to five Image Processing Toolbox routines: `bmpread()`, `ind2gray()`, `edge()`, `gray2ind()`, and `bmpwrite()`. As a group, these routines call five other Image Processing routines: `bestblk()`, `gray()`, `grayslice()`, `rgb2ntsc()`, and `fspecial()`. These last five routines don't call any other M-files, though they do call routines built into the MATLAB C++ Math Library. Therefore, 10 M-files need to be compiled to build this example.

You can find these examples in the `<matlab>\extern\examples\cppmath` directory of your MATLAB installation. Because the Image Processing Toolbox version of these routines contains calls to unsupported Handle Graphics® routines, the M-files used in this example have been modified so that they compile successfully into stand-alone code.

Building the Example

You can generate this C++ example with the following command:

```
mcc –p ex9 gray
```

This command compiles the M-file `ex9.m` and all other M-files called by `ex9`, searching for them on the MATLAB path, and creates multiple C++ output files. The `–p` switch also instructs the MATLAB Compiler to generate a C++

main wrapper. The default output language (without the –p switch) is C. Note that it is not necessary to specify the .m filename extensions.

Note Because the M-files in this example are modified versions of MATLAB 4.2 Image Processing Toolbox M-files, it is very important that you have the examples\cppmath directory on your MATLABPATH when you compile ex9.m. If you do not, the MATLAB Compiler may not find the required M-files, or find versions of the M-files that do not work with this example.

Running the Example

Run the program by typing

```
ex9 trees.bmp edges.bmp
```

at your system prompt.

If no errors occur, the program runs without printing any output and creates a file called edges.bmp in the current directory. This file contains the results of performing Sobel edge detection on the standard MATLAB image called trees.

Verify that the program worked by viewing the image in edges.bmp; almost any graphically-oriented Microsoft program (Paintbrush, MS Paint, etc.) will display a Microsoft Windows Bitmap file. On UNIX systems use a program like *xv* to view the image.

Compiler-Generated C++ Files

For *each* compiled M-file, the MATLAB Compiler creates two files:

- a corresponding .cpp file
- a corresponding .h file

The names of the output files derive from the name of the compiled M-file. For example, ex9.m produces ex9.cpp and ex9.h.

The Compiler places the main routine in a file called

- <function>_main.cpp

where <function> is the name of the first M-file specified in the mcc command. In this case, the file's name is ex9_main.cpp.

The Generated Main C++ Routine

The automatically-generated main routine creates MATLAB strings from any arguments passed to the function on the command line, assembles those strings into a variable length input argument list, and then uses `feval()` to call the top-level M-file, `ex9` in this example. The main routine wraps the `feval`() call in a try-catch block, which ensures that exceptions are caught and a corresponding error message printed.

```
int main(int argc, const char * * argv) {
    try {
        mwArray varargin(argc - 1, argv + 1);
        feval(mwAnsVarargout(), mlxEx9,
              mwVarargin(varargin.cell(colon())));
        return 0;
    } catch(mwException e) {
        cout << e;
        return 1;
    }
}
```

Generated feval() Function Table. In addition to the `main` routine, `ex9_main.cpp` contains the definition of the program's local `feval()` function table. To enable `feval()` to call any function, the MATLAB Compiler automatically places the functions it compiles into this local "function table." Initialization-time code adds this local table to `feval()`'s main table.

There are 11 entries in this example's function table.

```
static mlfFunctionTableEntry function_table[11] = {
    { "rgb2ntsc",  mlxRgb2ntsc, 1, 1 },
    { "grayslice", mlxGrayslice, 2, 1 },
     /* More entries go here */
};
```

Each entry maps the name of an M-file function (the quoted string) to information about the corresponding compiled `mlx`*`F`*, or `feval`, interface function. The second field in each entry is a pointer to `mlx`*`F`* function, and the third and fourth fields are the number of input and output arguments expected by the function. See Chapter 5 for more information on interface functions.

See "The Generated mlxF Interface Function" in this chapter for information on the `mlxEx9` and `mlxEdge` functions from this example.

C++ Functions Generated from each M-file Function

The MATLAB Compiler generates three C++ functions from each M-file function that it compiles.

- A `static` *implementation function* that contains the implementation of the M-file function
- Two public *interface functions* that represent the two interfaces to the static implementation function

For example, `ex9.cpp` contains the three functions generated from the M-file `ex9.m`.

The static implementation function is called the `Mf` implementation function, where *`f`* is replaced with the M-function name.

The first interface function is called the *`F`* interface function, or normal interface function, where *`F`* is replaced with the M-function name. It is the interface function that you would normally use to call the static implementation function.

The second interface function is the `mlx`*`F`* interface function, or `feval` interface function, where *`F`* is replaced with the M-function name. The MATLAB C++ Math Library function `feval` calls the `mlx`*`F`* interface function. The `feval` interface requires two additional arrays, one each for your input and your output arguments. While you can call the `feval` interface directly, it is easier to call the normal interface instead.

The Generated M*f* Implementation Function

The first function in `ex9.cpp` is the static M*f* implementation function: `Mex9`. This function contains the five step algorithm outlined at the beginning of this section.

Though you can't tell from the declaration of `Mex9`, the `Mf` implementation functions do not follow the standard calling conventions of the MATLAB C++ Math Library. The section "F Interface Functions That Return a Value or Take

Output Arguments" demonstrates a call to the `Mf` implementation function for edge(), which illustrates the full calling conventions.

```
(1)  static void Mex9(mwArray infile, mwArray outfile) {
(2)      mwArray bw(mwArray::UNASSIGNED);
         mwArray inten(mwArray::UNASSIGNED);
         mwArray map(mwArray::UNASSIGNED);
         mwArray x(mwArray::UNASSIGNED);
(3)      mwValidateInputs("ex9", 2, &infile, &outfile);

(4)      mbcharvector(infile);
         mbcharvector(outfile);
(5)      x = Nbmpread(2, &map, NULL, infile);
(6)      inten = ind2gray(x, map);
(7)      bw = edge(NULL, inten);
(8)      x = gray2ind(&map, bw);
(9)      bmpwrite(x, map, outfile);
     }
```

1 Declare the implementation function `Mex9()`. This static function is called by the *F* and `mlx`*F* interface functions. `Mex9()` takes two arguments, the names of the input and output files.

2 Initialize the local variables. In MATLAB a variable is always in one of three states: non-existent, unassigned, or initialized. The default C++ array constructor creates an array initialized to an empty array, which is the most appropriate default for writing code by hand. However, some M-file functions rely on the distinction between unassigned and initialized arrays, so variables in compiled functions must always be initialized to the unassigned state `mwArray::UNASSIGNED`.

3 Assert that each argument to this function has a value. In MATLAB, if you pass an unassigned variable to a function, you'll get an error message. `mwValidateInputs()` performs the same check in C++.

4 Assert that the input arguments are vectors of characters, or strings. The "`mb`" prefix in `mbcharvector` stands for "must be." In an M-file, these functions behave like type declaration hints for the MATLAB Compiler. If the argument to a "must be" function is not of the declared type, the function throws an exception.

5 Read in the bitmap. Assume the first command line argument is the name of an input file containing a Microsoft Windows Bitmap. `Nbmpread()` will fail if this is not the case. Store the image data in `x` and the colormap in `map`.

   Note that the MATLAB Compiler generated a call to the interface function `Nbmpread()` rather than the normal *`F`* `bmpread()` interface function because the M-function used the variable `nargout`. The `nargout` interface allows the number of requested outputs to be specified via the `nargout` argument, as opposed to the normal interface that dynamically calculates the number of outputs based on the number of non-NULL inputs it receives.

6 Convert the bitmap to a grayscale intensity image. `ind2gray()` returns a matrix the same size as the input image where all the pixel values range from `0.0` (no intensity, or black) to `1.0` (full intensity, or white).

7 Perform Sobel edge detection on the image. The edge detection routine supports four methods of edge detection: Sobel, Roberts, Prewitt, and Marr-Hildreth. Sobel is the default because it gives consistently good results; it finds edges where the first derivative of the image intensity is maximal or minimal.

8 Convert the grayscale intensity image to a black-and-white indexed image. The image returned by `edge()` contains only 1's (edge) and 0's (background). `gray2ind()` converts this intensity image into an indexed image and computes the associated colormap.

9 Write the image out to a bitmap file. Use the indexed image data and colormap generated by `gray2ind()`. Assume that the second argument on the command line (`argv[2]`) is the name of a file in which to write the output. Destroy any data currently in this file; if the file does not exist, create it.

The Generated *F* Interface Function

The next function in `ex9.cpp` is the *`F`* interface function, or normal interface function.

```
void ex9(mwArray infile, mwArray outfile) {
    Mex9(infile, outfile);
}
```

The *F* interface function has the same name as the first function in the M-file (`ex9` in this case) and is a wrapper around the static implementation function (`Mex9()` in this case).

The *F* interface function follows the usual calling conventions of the MATLAB C++ Math Library and is the function you'd call if you were incorporating `ex9.cpp` into a larger hand-written application.

The *F* interface function is responsible for converting from the standard MATLAB C++ Math Library calling conventions, which are designed for hand-written code, to the MATLAB Compiler's C++ calling conventions, which are designed for automatically generated code and are used by the static implementation functions.

***F* Interface Functions That Return a Value or Take Output Arguments.** It is instructive to compare the *F* interface function for the `edge()` function:

```
mwArray edge(mwArray * tol, mwArray a,  mwArray tol_,
             mwArray method, mwArray k)
{
    int nargout(1);
    mwArray e(mwArray::UNASSIGNED);
    mwArray tol__(mwArray::UNASSIGNED);
    if (tol == NULL) {
        tol = &tol__;
    } else {
        ++nargout;
    }
    e = Medge(tol, nargout, a, tol_, method, k);
    return e;
}
```

to the *F* interface function for `ex9`:

```
void ex9(mwArray infile, mwArray outfile) {
    Mex9(infile, outfile);
}
```

Notice that in the last two lines of `edge()`, the *F* interface function calls the implementation function `Medge()` with an argument list that includes an integer count of the number of output arguments and then returns the value returned by `Medge()` as its return value.

Passing the number of output arguments to `Medge()` allows it to tell if it was called with zero outputs. Many functions in MATLAB change their behavior when called with zero outputs.

In the last line of `ex9()`, the *F* interface function calls the implementation function `Mex9()` with two input arguments (*no* output arguments) and does not return a value.

The Generated mlx*F* Interface Function

The last function in `ex9.cpp` is the most complicated: the `mlx`*F*, or `feval`, interface function. The `mlx`*F* interface function is responsible for converting from the `feval` calling conventions to the standard MATLAB C++ Math Library calling conventions.

In MATLAB `feval()` can call any function. To maintain semantic consistency with MATLAB, every M-file function compiled by the MATLAB Compiler must therefore also have an `feval` interface function. At times the MATLAB Compiler needs to use `feval` to perform argument matching even if the user does not specifically call `feval`.

All functions callable by `feval` must have the same four parameter argument list:

- number of output parameters
- array of output parameters
- number of input parameters
- array of input parameters

Note The `mlxF` interface function has a C interface (uses `mxArray *` rather than `mwArray` arguments and returns) in order to allow a uniform `feval` interface between C and C++ in the MATLAB C/C++ Math Library.

```
(1)  void mlxEx9(int nlhs, mxArray * plhs[], int nrhs,
                 mxArray * prhs[]) {
(2)      MW_BEGIN_MLX();
         {
(3)          mwArray mprhs[2];
             int i;
(4)          if (nlhs > 0) {
                 error("Run-time Error: File: ex9 Line: 1 Column: 0
                        The function \"ex9\" was called with more than
                        the declared number of outputs (0)");
             }
(5)          if (nrhs > 2) {
                 error("Run-time Error: File: ex9 Line: 1 Column: 0
                        The function \"ex9\" was called with more than
                        the declared number of inputs (2)");
             }
(6)          for (i = 0; i < 2 && i < nrhs; ++i) {
                 mprhs[i] = mwArray(prhs[i], 0);
             }
(7)          for (; i < 2; ++i) {
                 mprhs[i] = mwArray::DIN;
             }
(8)          Mex9(mprhs[0], mprhs[1]);
         }
(9)      MW_END_MLX();
     }
```

1 Declare the `mlxF` interface function, `mlxEx9()`. The names of `feval` interface functions always begin with the three letter prefix `mlx`.

2 Insert `mlxF` `feval` interface function initialization code. Every `mlxF` interface function must invoke the macro `MW_BEGIN_MLX()` as the first statement in the function.

3. Declare local variables. This function uses two local variables, an array of `mwArray`s that will be the inputs to the M*f* implementation function and an integer for `for`-loops.

4. Verify that the number of output arguments is correct. The `Mf` implementation function, `Mex9()`, does not take any output arguments. The number of output arguments passed to the `mlxF` interface function must therefore be zero as well. Issue an error message if the number of output arguments exceeds zero.

5. Verify that the number of input arguments is correct. The `Mf` implementation function `Mex9()` expects at most two input arguments, so the number of input arguments passed to the `mlxF` interface function must not exceed two.

6. Initialize input arguments. The inputs to the `feval` interface function are pointers to `mxArray` structures, but the `Mf` implementation function expects `mwArray` objects. Create `mwArray` objects from the `mxArray` inputs. This process does not copy the data in the `mxArray` inputs, so making a copy is relatively inexpensive in terms of memory resources.

7. Provide default values for optional input arguments. MATLAB allows you to call a function with less than the maximum number of input arguments, but the `Mf` implementation function requires that you pass it all arguments, both required and optional. Any arguments that are not passed in by the user must be given default values before being passed to the `Mf` implementation function.

8. Call the `Mf` implementation function. Pass in the input arguments, processed by the previous two steps. This particular `Mf` implementation function has no return value or output arguments, so no post-processing is necessary.

9. Insert `mlxF` `feval` interface function termination code. Every `mlxF` interface function must invoke the `MW_END_MLX()` macro as the last statement in the function.

Mlx*F* Interface Functions That Return a Value or Take Output Arguments. It is useful to compare the last few lines of `mlxEdge()`, the `mlxF feval` interface function for `edge()`

```
mplhs[0] = Medge(&mplhs[1], nlhs, mprhs[0], mprhs[1], mprhs[2],
                 mprhs[3]);
plhs[0] = mplhs[0].FreezeData();
for (i = 1; i < 2 && i < nlhs; ++i) {
    plhs[i] = mplhs[i].FreezeData()
```

to the last few lines of `mlxEx9(), ex9()`'s `mlxF feval` interface function.

```
Mex9(mprhs[0], mprhs[1]);
```

The primary difference between the two is that `mlxEdge()` returns a value and takes an output argument; `mlxEx9()` does not. Because the `mlxF` interface functions use pointers to `mxArrays` rather than `mwArray` objects as arguments and return values, the `mwArray` objects returned by the `Mf` implementation function must be converted to `mxArray` pointers. Because each `mwArray` object contains a pointer to an `mxArray`, the most efficient way to perform this conversion is to extract the `mxArray` pointer from the `mwArray` object.

There is one problem, however: a local `mwArray` object destroys its `mxArray` pointer when it goes out of scope when the `mlxF` interface function returns. The function `FreezeData()` solves this problem by modifying the `mwArray` object to prevent it from destroying its `mxArray` pointer and by returning the `mxArray` pointer.

Calling `FreezeData()` introduces the potential for memory leaks because the `mxArray` pointer is essentially released from the MATLAB C++ Math Library's automatic memory management. However, the `mlxF feval` interface function returns these values to `feval()`, which in turn returns them to its caller, a C++ function where the return values from `feval()` will be used to construct `mwArray` objects once more, thereby placing the memory under automatic memory management again.

5

Controlling Code Generation

Introduction

This chapter describes the code generated by the MATLAB Compiler and the options that you can use to control code generation. In particular, it discusses:

- Compiling private and method functions
- The generated header files
- The generated C or C++ code
- Internal interface functions
- Supported executable types
- Formatting Compiler-generated code
- Including M-file source in Compiler output
- Interfacing M-code to C/C++ code

To generate the various files created by the Compiler, this chapter uses several different M-files — `gasket.m`, `foo.m`, `fun.m`, and `sample.m`.

Example M-Files

Sierpinski Gasket M-File

```
function theImage = gasket(numPoints)
%GASKET An image of a Sierpinski Gasket.
%   IM = GASKET(NUMPOINTS)
%
%   Example:
%   x = gasket(50000);
%   imagesc(x);colormap([0 0 0;1 1 1]);
%   axis equal tight

%   Copyright (c) 1984-98 by The MathWorks, Inc
%   $Revision: 1.1 $  $Date: 1998/09/11 20:05:06 $

theImage = zeros(1000,1000);

corners = [866 1;1 500;866 1000];
startPoint = [866 1];
theRand = rand(numPoints,1);
theRand = ceil(theRand*3);

for i=1:numPoints
    startPoint = floor((corners(theRand(i),:)+startPoint)/2);
    theImage(startPoint(1),startPoint(2)) = 1;
end
```

foo M-File

```
function [a, b] = foo(x, y)
if nargout == 0
elseif nargout == 1
    a = x;
elseif nargout == 2
    a = x;
    b = y;
end
```

fun M-File

```
function a = fun(b)
a(1) = b(1) .* b(1);
a(2) = b(1) + b(2);
a(3) = b(2) / 4;
```

sample M-File

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

Generated Code

This chapter investigates the generated header files, main source files, interface functions, and wrapper functions for the C MEX, stand-alone C and C++ targets, and C and C++ libraries.

When you use the MATLAB Compiler to compile an M-file, it generates these files:

- C or C++ code, depending on your target language (`–L`) specification
- Header file
- Wrapper file, depending on the `–W` option

The C or C++ code that is generated by the Compiler and the header file are independent of the final target type — MEX, executable, or library. That is, the C or C++ code and header file are identical no matter what the desired final output. The wrapper file provides the code necessary to support the output executable type. So, the wrapper file is different for each executable type.

Table 5-1 shows the names of the files generated when you compile a generic M-file (`file.m`) for the MEX and stand-alone targets. The table also shows the files generated when you compile a set of files (`filelist`) for the library target.

Table 5-1: Compiler-Generated Files

	C	C++
Header	`file.h`	`file.hpp`
Code	`file.c`	`file.cpp`
Main Wrapper (–W main)	`file_main.c`	`file_main.cpp`
MEX Wrapper (–W mex)	`file_mex.c`	N/A (C++ MEX-files are not supported.)
Simulink Wrapper (–W simulink)	`file_simulink.c`	N/A (C++ MEX-files are not supported.)

Library (–W lib:filelist)	`filelist.c`
	`filelist.h`
	`filelist.exports`
		`filelist.cpp`
		`filelist.hpp`

Note Many of the code snippets generated by the MATLAB Compiler that are used in this chapter use the `–F page-width` option to produce readable code that fits nicely on the book's printed page. For more information about the `page-width` option, see "Formatting Compiler-Generated Code" later in this chapter.

Compiling Private and Method Functions

Private functions are functions that reside in subdirectories with the special name `private`, and are visible only to functions in the parent directory. Since private functions are invisible outside of the parent directory, they can use the same names as functions in other directories. Because MATLAB looks for private functions before standard M-file functions, it will find a private function before a nonprivate one.

Method functions are implementations specific to a particular MATLAB type or user-defined object. Method functions are only invoked when the argument list contains an object of the correct class.

In order to compile a method function, you must specify the name of the method along with the classname so that the Compiler can differentiate the method function from a nonmethod (normal) function.

Note Although Compiler 2.0 can currently compile method functions, it does not support overloading of methods as implemented in MATLAB. This feature is provided in anticipation of support of overloaded methods being added.

Method directories can contain private directories. Private functions are found only when executing a method from the parent method directory. Taking all of this into account, the Compiler command line needs to be able to differentiate between these various functions that have the same name. A file called `foo.m` that contains a function called `foo` can appear in all of these locations at the same time. The conventions used on the Compiler command line are as documented in this table.

Name	Description
`foo.m`	Default version of `foo.m`
`xxx/private/foo.m`	`foo.m` private to the `xxx` directory
`@cell/foo.m`	`foo.m` method to operate on cell arrays
`@cell/private/foo.m`	`foo.m` private to methods that operate on cell arrays

This table lists the functions you can specify on the command line and their corresponding function and filenames.

Function	C Function	C++ Function	Filename
foo	mlfFoo mlxFoo mlNFoo mlfNFoo mlfVFoo	foo Nfoo Vfoo mlxFoo	foo.c foo.h foo.cpp foo.hpp
@cell/foo	mlf_cell_foo mlx_cell_foo mlN_cell_foo mlfN_cell_foo mlfV_cell_foo	_cell_foo N_cell_foo V_cell_foo mlx_cell_foo	_cell_foo.c _cell_foo.h _cell_foo.cpp _cell_foo.hpp
xxx/private/foo	mlfXXX_private_foo mlxXXX_private_foo mlNXXX_private_foo mlfNXXX_private_foo mlfVXXX_private_foo	XXX_private_foo NXXX_private_foo VXXX_private_foo mlxXXX_private_foo	_XXX_private_foo.c _XXX_private_foo.h _XXX_private_foo.cpp _XXX_private_foo.hpp
@cell/private/foo	mlfcell_private_foo mlxcell_private_Foo mlNcell_private_Foo mlfNcell_private_Foo mlfVcell_private_Foo	_cell_private_foo N_cell_private_foo V_cell_private_foo mlx_cell_private_foo	_cell_private_foo.c _cell_private_foo.h _cell_private_foo.cpp _cell_private_foo.hpp

For private functions, the name given in the table above may be ambiguous. The MATLAB Compiler generates a warning when it cannot distinguish which private function to use. For example, given these two `foo.m` private functions and their locations

```
/Z/X/private/foo.m
/Y/X/private/foo.m
```

the Compiler searches up only one level and determines the path to the file as

```
X/private/foo.m
```

Since it is ambiguous which `foo.m` you are requesting, it generates the warning

```
Warning: The specified private directory is not unique. Both
/Z/X/private and /Y/X/private are found on the path for this
private directory.
```

The Generated Header Files

This section highlights the two header files that the Compiler can generate for the Sierpinski Gasket (`gasket.m`) example.

C Header File

If the target language is C, the Compiler generates the header file, `gasket.h`. This example uses the Compiler command

```
mcc –t –L C –T codegen –F page–width:60 gasket
```

to generate the associated files. The C header file, `gasket.h`, is:

```
/*
 * MATLAB Compiler: 2.0b1
 * Date: Mon Dec 14 08:46:21 1998
 * Arguments: "–t" "–L" "C" "–T" "codegen" "–F"
 * "page-width:60" "gasket"
 */

#ifndef MLF_V2
#define MLF_V2 1
#endif

#ifndef __gasket_h
#define __gasket_h 1

#include "matlab.h"

extern mxArray * mlfGasket(mxArray * numPoints);
extern void mlxGasket(int nlhs,
                      mxArray * plhs[],
                      int nrhs,
                      mxArray * prhs[]);

#endif
```

C++ Header File

If the target language is C++, the Compiler generates the header file, `gasket.hpp`. This example uses the Compiler command

```
mcc –t –L Cpp –T codegen –F page–width:60 gasket
```

to generate the associated files. The C++ header file, `gasket.hpp`, is:

```
//
// MATLAB Compiler: 2.0b1
// Date: Mon Dec 14 08:48:50 1998
// Arguments: "–t" "–L" "Cpp" "–T" "codegen" "–F"
// "page-width:60" "gasket"
//
#ifndef __gasket_hpp
#define __gasket_hpp 1

#include "matlab.hpp"

extern mwArray gasket(mwArray numPoints = mwArray::DIN);
#ifdef __cplusplus
extern "C"
#endif
void mlxGasket(int nlhs,
               mxArray * plhs[],
               int nrhs,
               mxArray * prhs[]);

#endif
```

The Generated C/C++ Code

This section focuses on the generated C and C++ code from the `gasket.m` and `foo.m` examples. In each of these examples, the implementation function, `Mf`, contains the code generated from the M source file. These implementation functions are required to provide a uniform interface to other functions.

C Code from gasket.m

If the target language is C, the Compiler generates the C source file, `gasket.c`. This example uses the Compiler command

```
mcc –t –L C –T codegen –F page–width:60 gasket
```

to generate the associated files. Note that the interface functions have been omitted for readability. Refer to the section, " C Interface Functions" for more information about these functions. If you want to modify the formatting and content of the generated C code, use the `–F` and `–A` options of `mcc`. For more information on these options, see the "Formatting Compiler-Generated Code" and "Including M-File Information in Compiler Output" sections later in this chapter.

The C source file, `gasket.c`, is:

```
/*
 * MATLAB Compiler: 2.0b1
 * Date: Mon Dec 14 08:55:55 1998
 * Arguments: "–t" "–L" "C" "–T" "codegen" "–F"
 * "page-width:60" "gasket"
 */
#include "gasket.h"

static double __Array0_r[6]
  = { 866.0, 1.0, 866.0, 1.0, 500.0, 1000.0 };

static double __Array1_r[2] = { 866.0, 1.0 };

/*
 * The function "Mgasket" is the implementation version of
 * the "gasket" M-function from file
 * "<matlab>\extern\examples\compiler\gasket.m" (lines 1-24).
 * It contains the actual compiled code for that
```

```
 * M-function. It is a static function and must only be
 * called from one of the interface functions, appearing
 * below.
 */
/*
 * function theImage = gasket(numPoints)
 */
static mxArray * Mgasket(int nargout_,
                         mxArray * numPoints) {
    mxArray * theImage = mclUnassigned();
    mxArray * corners = mclGetUninitializedArray();
    mxArray * i = mclGetUninitializedArray();
    mclForLoopIterator iterator_0;
    mxArray * startPoint = mclGetUninitializedArray();
    mxArray * theRand = mclGetUninitializedArray();
    mclValidateInputs("gasket", 1, &numPoints);
    /*
     * %GASKET An image of a Sierpinski Gasket.
     * %   IM = GASKET(NUMPOINTS)
     * %
     * %   Example:
     * %   x = gasket(50000);
     * %   imagesc(x);colormap([0 0 0;1 1 1]);
     * %   axis equal tight
     *
     * %   Copyright (c) 1984-98 by The MathWorks, Inc
     * %   $Revision: 1.1 $  $Date: 1998/09/11 20:05:06 $
     *
     * theImage = zeros(1000,1000);
     */
    mlfAssign(
      &theImage,
      mlfZeros(mlfScalar(1000.0), mlfScalar(1000.0), NULL));
    /*
     *
     * corners = [866 1;1 500;866 1000];
     */
    mlfAssign(
      &corners, mlfDoubleMatrix(3, 2, __Array0_r, NULL));
    /*
```

```
 * startPoint = [866 1];
 */
mlfAssign(
  &startPoint, mlfDoubleMatrix(1, 2, __Array1_r, NULL));
/*
 * theRand = rand(numPoints,1);
 */
mlfAssign(
  &theRand, mlfRand(numPoints, mlfScalar(1.0), NULL));
/*
 * theRand = ceil(theRand*3);
 */
mlfAssign(
  &theRand,
  mlfCeil(mlfMtimes(theRand, mlfScalar(3.0))));
/*
 *
 * for i=1:numPoints
 */
for (mclForStart(
       &iterator_0, mlfScalar(1.0), numPoints, NULL);
     mclForNext(&iterator_0, &i);
     ) {
    /*
   *startPoint= floor((corners(theRand(i),:)+startPoint)/2);
     */
    mlfAssign(
      &startPoint,
      mlfFloor(
        mlfMrdivide(
          mlfPlus(
            mlfIndexRef(
              corners,
              "(?,?)",
              mlfIndexRef(theRand, "(?)", i),
              mlfCreateColonIndex()),
            startPoint),
          mlfScalar(2.0))));
    /*
     * theImage(startPoint(1),startPoint(2)) = 1;
```

```
         */
        mlfIndexAssign(
          &theImage,
          "(?,?)",
          mlfIndexRef(startPoint, "(?)", mlfScalar(1.0)),
          mlfIndexRef(startPoint, "(?)", mlfScalar(2.0)),
          mlfScalar(1.0));
    /*
     * end
     */
    }
    mclValidateOutputs("gasket", 1, nargout_, &theImage);
    mxDestroyArray(corners);
    mxDestroyArray(i);
    mxDestroyArray(startPoint);
    mxDestroyArray(theRand);
    return theImage;
}

/*
 * "mlfGasket" interface function
*/
     .
     .
     .

/*
 * "mlxGasket" interface function
*/
     .
     .
     .
```

C Code from foo.m

If the target language is C, the Compiler generates the C source file, `foo.c`. This example uses the Compiler command

```
mcc –t –L C –T codegen –F page–width:60 foo
```

to generate the associated files. Note that the interface functions have been omitted for readability. Refer to the section, " C Interface Functions" for more information about these functions.

```
/*
 * MATLAB Compiler: 2.0b1
 * Date: Mon Dec 14 09:06:03 1998
 * Arguments: "-t" "-L" "C" "-T" "codegen" "-F"
 * "page-width:60" "foo"
 */
#include "foo.h"

/*
 * The function "Mfoo" is the implementation version of the
 * "foo" M-function from file
 * "<matlab>\examples\compiler\foo.m" (lines 1-10). It
 * contains the actual compiled code for that M-function.
 * It is a static function and must only be called from one
 * of the interface functions, appearing below.
 */
/*
 * function [a, b] = foo(x, y)
 */
static mxArray * Mfoo(mxArray * * b,
                      int nargout_,
                      mxArray * x,
                      mxArray * y) {
    mxArray * a = mclUnassigned();
    mxArray * nargout = mclInitialize(mlfScalar(nargout_));
    mclValidateInputs("foo", 2, &x, &y);
    /*
     *
     * if nargout == 0
     */
    if (mlfTobool(mlfEq(nargout, mlfScalar(0.0)))) {
    /*
     * elseif nargout == 1
     */
    } else if (mlfTobool(mlfEq(nargout, mlfScalar(1.0)))) {
```

```
        /*
         * a = x;
         */
        mlfAssign(&a, x);
    /*
     * elseif nargout == 2
     */
    } else if (mlfTobool(mlfEq(nargout, mlfScalar(2.0)))) {
        /*
         * a = x;
         */
        mlfAssign(&a, x);
        /*
         * b = y;
         */
        mlfAssign(b, y);
    /*
     * end
     */
    }
    mclValidateOutputs("foo", 2, nargout_, &a, b);
    mxDestroyArray(nargout);
    return a;
}

/*
 * "mlfNFoo" interface function
*/
     .
     .
     .

/*
 * "mlfFoo" interface function
*/
     .
     .
     .
```

```
/*
 * "mlfVFoo" interface function
*/
     .
     .
     .

/*
 * "mlxFoo" interface function
*/
     .
     .
     .
```

C++ Code from gasket.m

If the target language is C++, the Compiler generates the C++ source file, `gasket.cpp`. Note that the interface functions have been omitted for readability. Refer to the section, " C++ Interface Functions" for more information about these functions. If you want to modify the formatting and content of the generated C++ code, use the –F and –A options of `mcc`. For more information on these options, see the "Formatting Compiler-Generated Code" and "Including M-File Information in Compiler Output" sections later in this chapter.

This example uses the Compiler command

```
mcc –t –L Cpp –T codegen –F page–width:60 gasket
```

to generate the associated files. The C++ source file, `gasket.cpp`, is:

```
//
// MATLAB Compiler: 2.0b1
// Date: Mon Dec 14 09:11:47 1998
// Arguments: "–t" "–L" "Cpp" "–T" "codegen" "–F"
// "page-width:60" "gasket"
//
#include "gasket.hpp"

static double __Array0_r[6]
  = { 866.0, 1.0, 866.0, 1.0, 500.0, 1000.0 };
```

```
static double __Array1_r[2] = { 866.0, 1.0 };

//
// The function "Mgasket" is the implementation version of
// the "gasket" M-function from file
// "<matlab>\extern\examples\compiler\gasket.m" (lines 1-24).
// It contains the actual compiled code for that
// M-function. It is a static function and must only be
// called from one of the interface functions, appearing
// below.
//
//
// function theImage = gasket(numPoints)
//
static mwArray Mgasket(int nargout_, mwArray numPoints) {
    mwArray theImage(mwArray::UNASSIGNED);
    mwArray corners(mclGetUninitializedArray());
    mwArray i(mclGetUninitializedArray());
    mwForLoopIterator iterator_0;
    mwArray startPoint(mclGetUninitializedArray());
    mwArray theRand(mclGetUninitializedArray());
    mwValidateInputs("gasket", 1, &numPoints);
    //
    // %GASKET An image of a Sierpinski Gasket.
    // %   IM = GASKET(NUMPOINTS)
    // %
    // %   Example:
    // %   x = gasket(50000);
    // %   imagesc(x);colormap([0 0 0;1 1 1]);
    // %   axis equal tight
    //
    // %   Copyright (c) 1984-98 by The MathWorks, Inc
    // %   $Revision: 1.1 $  $Date: 1998/09/11 20:05:06 $
    //
    // theImage = zeros(1000,1000);
    //
    theImage = zeros(mwVarargin(1000.0, 1000.0));
    //
    //
    // corners = [866 1;1 500;866 1000];
```

```
    //
    corners = mwArray(3, 2, __Array0_r, NULL);
    //
    // startPoint = [866 1];
    //
    startPoint = mwArray(1, 2, __Array1_r, NULL);
    //
    // theRand = rand(numPoints,1);
    //
    theRand = rand_func(mwVarargin(numPoints, 1.0));
    //
    // theRand = ceil(theRand*3);
    //
    theRand = ceil(theRand * mwArray(3.0));
    //
    //
    // for i=1:numPoints
    //
    for (iterator_0.Start(1.0, numPoints, mwArray::DIN);
         iterator_0.Next(&i);
         ) {
        //
      //startPoint= floor((corners(theRand(i),:)+startPoint)/2);
        //
        startPoint
          = floor(
              (corners(theRand(i), colon()) + startPoint)
              / mwArray(2.0));
        //
        // theImage(startPoint(1),startPoint(2)) = 1;
        //
        theImage(startPoint(1.0), startPoint(2.0)) = 1.0;
    //
    // end
    //
    }
    mwValidateOutputs("gasket", 1, nargout_, &theImage);
    return theImage;
}
```

```
//
// "gasket" interface function
//
     .
     .
     .

//
// "mlxGasket" interface function
//
     .
     .
     .
```

C++ Code from foo.m

If the target language is C++, the Compiler generates the C++ source file, `foo.cpp`. Note that the interface functions have been omitted for readability. Refer to the section, " C++ Interface Functions" for more information about these functions.

This example uses the Compiler command

```
mcc –t –L Cpp –T codegen –F page–width:60 foo
```

to generate the associated files. The C++ source file, `foo.cpp`, is:

```
//
// MATLAB Compiler: 2.0b1
// Date: Mon Dec 14 09:17:16 1998
// Arguments: "–t" "–L" "Cpp" "–T" "codegen" "–F"
// "page-width:60" "foo"
//
#include "foo.hpp"

//
// The function "Mfoo" is the implementation version of the
// "foo" M-function from file
// "<matlab>\extern\examples\compiler\foo.m" (lines 1-10). It
// contains the actual compiled code for that M-function.
// It is a static function and must only be called from one
// of the interface functions, appearing below.
```

```
//
//
// function [a, b] = foo(x, y)
//
static mwArray Mfoo(mwArray * b,
                    int nargout_,
                    mwArray x,
                    mwArray y) {
    mwArray a(mwArray::UNASSIGNED);
    mwArray nargout(nargout_);
    mwValidateInputs("foo", 2, &x, &y);
    //
    //
    // if nargout == 0
    //
    if (tobool(nargout == mwArray(0.0))) {
    //
    // elseif nargout == 1
    //
    } else if (tobool(nargout == mwArray(1.0))) {
        //
        // a = x;
        //
        a = x;
    //
    // elseif nargout == 2
    //
    } else if (tobool(nargout == mwArray(2.0))) {
        //
        // a = x;
        //
        a = x;
        //
        // b = y;
        //
        *b = y;
    //
    // end
    //
    }
```

```
    mwValidateOutputs("foo", 2, nargout_, &a, b);
    return a;
}

//
// "Nfoo" interface function
//
     .
     .
     .

//
// "foo" interface function
//
     .
     .
     .

//
// "Vfoo" interface function
//
     .
     .
     .

//
// "mlxFoo" interface function
//
     .
     .
     .
```

Internal Interface Functions

This section uses the Sierpinski Gasket example (`gasket.m`) to show several of the generated interface functions for the C and C++ cases. The remaining interface functions are generated by the example `foo.m`. as described earlier in this chapter.

Interface functions perform argument translation between the standard calling conventions and the Compiler-generated code.

C Interface Functions

The C interface functions process any input arguments and pass them to the implementation version of the function, `Mf`, as shown in the section, "C Code from gasket.m."

mlx*F* Interface Function

The Compiler always generates the `mlxF` interface function, which is used by `feval`. At times, the Compiler needs to use `feval` to perform argument matching even if the user does not specifically call `feval`. For example,

```
[ x{:} ] = gasket
```

would use the `feval` interface. The following C code is the corresponding `feval` interface (`mlxGasket`) from the Sierpinski Gasket example. This function calls the C `Mgasket` function shown in "C Code from gasket.m."

```
/*
 * The function "mlxGasket" contains the feval interface
 * for the "gasket" M-function from file
 * "<matlab>\extern\examples\compiler\gasket.m" (lines 1-24).
 * The feval function calls the implementation version of
 * gasket through this function. This function processes
 * any input arguments and passes them to the
 * implementation version of the function, appearing above.
 */
void mlxGasket(int nlhs,
               mxArray * plhs[],
               int nrhs,
               mxArray * prhs[]) {
    mxArray * mprhs[1];
```

```
    mxArray * mplhs[1];
    int i;
    if (nlhs > 1) {
        mlfError(
          mxCreateString(
            "Run-time Error: File: gasket Line: 1 Column: "
            "0 The function \"gasket\" was called with mor"
            "e than the declared number of outputs (1)"));
    }
    if (nrhs > 1) {
        mlfError(
          mxCreateString(
            "Run-time Error: File: gasket Line: 1 Column: "
            "0 The function \"gasket\" was called with mor"
            "e than the declared number of inputs (1)"));
    }
    for (i = 0; i < 1; ++i) {
        mplhs[i] = NULL;
    }
    for (i = 0; i < 1 && i < nrhs; ++i) {
        mprhs[i] = prhs[i];
    }
    for (; i < 1; ++i) {
        mprhs[i] = NULL;
    }
    mlfEnterNewContext(0, 1, mprhs[0]);
    mplhs[0] = Mgasket(nlhs, mprhs[0]);
    mlfRestorePreviousContext(0, 1, mprhs[0]);
    plhs[0] = mplhs[0];
}
```

Input argument processing

Call to C implementation function

Output argument processing

mlf*F* Interface Function

The Compiler always generates the `mlf`*F* interface function, which contains the "normal" C interface to the function. This code is the corresponding C interface

function (`mlfGasket`) from the Sierpinski Gasket example. This function calls the C `mgasket` function shown in "C Code from gasket.m."

```
/*
 * The function "mlfGasket" contains the normal interface
 * for the "gasket" M-function from file
 * "<matlab>\extern\examples\compiler\gasket.m" (lines 1-24).
 * This function processes any input arguments and passes
 * them to the implementation version of the function,
 * appearing above.
 */
mxArray * mlfGasket(mxArray * numPoints) {
  int nargout = 1;
  mxArray * theImage = mclUnassigned();
  mlfEnterNewContext(0, 1, numPoints);
  theImage = Mgasket(nargout, numPoints);
  mlfRestorePreviousContext(0, 1, numPoints);
  mlfReturnValue(theImage);
  return theImage;
}
```

Input argument processing

Call M-function

Output argument processing

mlfN*F* Interface Function

The Compiler produces this interface function only when the M-function uses the variable `nargout`.The `nargout` interface allows you to specify the number of requested outputs via the int `nargout` argument, as opposed to the normal interface that dynamically calculates the number of outputs based on the number of non-null inputs it receives.

This is the corresponding `mlfN`*`F`* interface function (`mlfNFoo`) for the `foo.m` example described earlier in this chapter. This function calls the `Mfoo` function that appears in `foo.c`.

```
/*
 * The function "mlfNFoo" contains the nargout interface
 * for the "foo" M-function from file
 * "<matlab>\extern\examples\compiler\foo.m" (lines 1-10).
 * This interface is only produced if the M-function uses
 * the special variable "nargout". The nargout interface
 * allows the number of requested outputs to be specified
 * via the nargout argument, as opposed to the normal
 * interface which dynamically calculates the number of
```

```
 * outputs based on the number of non-NULL inputs it
 * receives. This function processes any input arguments
 * and passes them to the implementation version of the
 * function, appearing above.
 */
mxArray * mlfNFoo(int nargout,
          mxArray * * b,
          mxArray * x,
          mxArray * y) {
    mxArray * a = mclUnassigned();
    mxArray * b__ = mclUnassigned();
    mlfEnterNewContext(1, 2, b, x, y);
    if (b == NULL) {
        b = &b__;
    }
    a = Mfoo(b, nargout, x, y);
    mlfRestorePreviousContext(1, 2, b, x, y);
    mxDestroyArray(b__);
    mlfReturnValue(a);
    return a;
}
```

Input argument processing

Call M-function

Output argument processing

mlfV*F* Interface Function

The Compiler produces this interface function only when the M-function uses the variable `nargout` and has at least one output. This `void` interface function specifies zero output arguments to the implementation version of the function, and in the event that the implementation version still returns an output (which, in MATLAB, would be assigned to the `ans` variable), it deallocates the output.

This is the corresponding `mlfV`*F* interface function (`mlfVFoo`) for the `foo.m` example described at the beginning of this section. This function calls the C `Mfoo` implementation function that appears in `foo.c`.

```
/*
 * The function "mlfVFoo" contains the void interface for
 * the "foo" M-function from file
 * "<matlab>\extern\examples\compiler\foo.m" (lines 1-10). The
 * void interface is only produced if the M-function uses
 * the special variable "nargout", and has at least one
```

```
 * output. The void interface function specifies zero
 * output arguments to the implementation version of the
 * function, and in the event that the implementation
 * version still returns an output (which, in MATLAB, would
 * be assigned to the "ans" variable), it deallocates the
 * output. This function processes any input arguments and
 * passes them to the implementation version of the
 * function, appearing above.
 */
void mlfVFoo(mxArray * x, mxArray * y) {
    mxArray * a = mclUnassigned();
    mxArray * b = mclUnassigned();
    mlfEnterNewContext(0, 2, x, y);
    a = Mfoo(&b, 0, x, y);
    mlfRestorePreviousContext(0, 2, x, y);
    mxDestroyArray(a);
    mxDestroyArray(b);
}
```

Input argument processing

Call M-function

Output argument processing

C++ Interface Functions

The C++ interface functions process any input arguments and pass them to the implementation version of the function as shown in the section, "C++ Code from gasket.m."

Note In C++, the `mlx`*F* interface functions are also C functions in order to allow the `feval` interface to be uniform between C and C++.

---

mlx*F* Interface Function

The Compiler always generates the `mlx`*F* interface function, which is used by `feval`. At times, the Compiler needs to use `feval` to perform argument matching even if the user does not specifically call `feval`. For example,

```
[ x{:} ] = gasket
```

would use the `feval` interface. The following C++ code is the corresponding `feval` interface (`mlxGasket`) from the Sierpinski Gasket example. This function calls the C++ `Mgasket` function shown in "C++ Code from gasket.m."

```
//
// The function "mlxGasket" contains the feval interface
// for the "gasket" M-function from file
// "<matlab>\extern\examples\compiler\gasket.m" (lines 1-24).
// The feval function calls the implementation version of
// gasket through this function. This function processes
// any input arguments and passes them to the
// implementation version of the function, appearing above.
//
void mlxGasket(int nlhs,
               mxArray * plhs[],
               int nrhs,
               mxArray * prhs[]) {
    MW_BEGIN_MLX();
{
        mwArray mprhs[1];
        mwArray mplhs[1];
        int i;
        if (nlhs > 1) {
            error(
              "Run-time Error: File: gasket Line:"
              " 1 Column: 0 The function \"gasket"
              "\" was called with more than the d"
              "eclared number of outputs (1)");
        }
        if (nrhs > 1) {
            error(
              "Run-time Error: File: gasket Line: 1 Column:"
              " 0 The function \"gasket\" was called with m"
              "ore than the declared number of inputs (1)");
        }
        for (i = 0; i < 1 && i < nrhs; ++i) {
            mprhs[i] = mwArray(prhs[i], 0);
        }
        for (; i < 1; ++i) {
            mprhs[i] = mwArray::DIN;
```

Input argument processing

```
            }
Call M-function ⟶   mplhs[0] = Mgasket(nlhs, mprhs[0]);
Output argument ⟶   plhs[0] = mplhs[0].FreezeData();
processing      }
        MW_END_MLX();
    }
```

F Interface Function

The Compiler always generates the *F* interface function, which contains the "normal" C++ interface to the function. This code is the corresponding C++ interface function (`gasket`) from the Sierpinski Gasket example. This function calls the C++ code shown in "C++ Code from gasket.m."

```
//
// The function "gasket" contains the normal interface for
// the "gasket" M-function from file
// "<matlab>\extern\examples\compiler\gasket.m" (lines 1-24).
// This function processes any input arguments and passes
// them to the implementation version of the function,
// appearing above.
//
mwArray gasket(mwArray numPoints) {
    int nargout(1);
    mwArray theImage(mwArray::UNASSIGNED);
Call M-function ⟶   theImage = Mgasket(nargout, numPoints);
Output argument ⟶   return theImage;
processing      }
```

N*F* Interface Function

The Compiler produces this interface function only when the M-function uses the variable `nargout`. The `nargout` interface allows the number of requested outputs to be specified via the `nargout` argument, as opposed to the normal interface that dynamically calculates the number of outputs based on the number of non-null inputs it receives.

This is the corresponding `NF` interface function (`NFoo`) for the `foo.m` example described earlier in this chapter. This function calls the `Mfoo` function appearing in `foo.cpp`.

```
//
// The function "Nfoo" contains the nargout interface for
```

```
// the "foo" M-function from file
// "<matlab>\extern\examples\compiler\foo.m" (lines 1-10).
// This interface is only produced if the M-function uses
// the special variable "nargout". The nargout interface
// allows the number of requested outputs to be specified
// via the nargout argument, as opposed to the normal
// interface which dynamically calculates the number of
// outputs based on the number of non-NULL inputs it
// receives. This function processes any input arguments
// and passes them to the implementation version of the
// function, appearing above.
//
mwArray Nfoo(int nargout,
             mwArray * b,
             mwArray x,
             mwArray y) {
    mwArray a(mwArray::UNASSIGNED);
    mwArray b__(mwArray::UNASSIGNED);
    if (b == NULL) {
        b = &b__;
    }
    a = Mfoo(b, nargout, x, y);
    return a;
}
```

Input argument processing

Call M-function →

Output argument processing →

V*F* Interface Function

The Compiler produces this interface function only when the M-function uses the variable `nargout` and has at least one output. The `void` interface function specifies zero output arguments to the implementation version of the function, and in the event that the implementation version still returns an output (which, in MATLAB, would be assigned to the `ans` variable), it deallocates the output.

This is the corresponding `VF` interface function (`VFoo`) for the `foo.m` example described earlier in this chapter. This function calls the `Mfoo` function appearing in `foo.cpp`.

```
//
// The function "Nfoo" contains the nargout interface for
// the "foo" M-function from file
```

```
// "<matlab>\extern\examples\compiler\foo.m" (lines 1-10).
// This interface is only produced if the M-function uses
// the special variable "nargout". The nargout interface
// allows the number of requested outputs to be specified
// via the nargout argument, as opposed to the normal
// interface which dynamically calculates the number of
// outputs based on the number of non-NULL inputs it
// receives. This function processes any input arguments
// and passes them to the implementation version of the
// function, appearing above.
//
void Vfoo(mwArray x, mwArray y) {
    mwArray a(mwArray::UNASSIGNED);
    mwArray b(mwArray::UNASSIGNED);
    a = Mfoo(&b, 0, x, y);
}
```

Input argument processing

Call M-function

Supported Executable Types

Wrapper functions create a link between the Compiler-generated code and a supported executable type by providing the required interface that allows the code to operate in the desired execution environment.

The wrapper functions differ depending on the execution environment, whereas the C and C++ header files and code that are generated by the Compiler are the same for MEX-functions, stand-alone applications, and libraries.

To provide the required interface, the wrapper:

- Defines persistent/global variables
- Initializes the `feval` function table for run-time `feval` support
- Performs wrapper-specific initialization and termination

This section discusses the various wrappers that can be generated using the MATLAB Compiler.

Note When the Compiler generates a wrapper function, it must examine all of the `.m` files that will be included into the executable. If you do not include all the files, the Compiler may not define all of the global variables.

Generating Files

You can use the –`t` option of the Compiler to generate source files in addition to wrapper files. For example

```
mcc –W main –h x.m
```

examines `x.m` and all M-files referenced by `x.m`, but generates only the `x_main.c` wrapper file. However, including the –`t` option in

```
mcc –W main –h –t x.m
```

generates `x_main.c`, `x.c`, and all M-files referenced by `x.m`.

MEX-Files

The –W mex –L C options produce the MEX-file wrapper, which includes the mexFunction interface that is standard to all MATLAB plug-ins. For more information about the requirements of the mex interface, see the *MATLAB Application Program Interface Guide*.

In addition to declaring globals and initializing the feval function table, the MEX-file wrapper function includes interface and definition functions for all M-files not included into the set of compiled files. These functions are implemented as callbacks to MATLAB.

Note By default, the –x option does not include any functions that do not appear on the command line. Functions that do not appear on the command line would generate a callback to MATLAB. Specify –h if you want all functions called to be compiled into your MEX-file.

This wrapper function uses the gasket.m file and is produced from the command:

```
mcc –W mex –L C –T codegen –F page–width:55 gasket
```

The generated MEX wrapper function, gasket_mex.c is:

```
/*
 * MATLAB Compiler: 2.0b1
 * Date: Mon Dec 14 14:02:42 1998
 * Arguments: "–W" "mex" "–L" "C" "–T"
 * "codegen" "–F" "page-width:55" "gasket"
 */

#ifndef MLF_V2
#define MLF_V2 1
#endif

#include "matlab.h"
#include "gasket.h"

static mlfFunctionTableEntry function_table[1]
  = { { "gasket", mlxGasket, 1, 1 } };
```

```
/*
 * The function "mexFunction" is a Compiler-generated
 * mex wrapper, suitable for building a MEX-function.
 * It initializes any persistent variables as well as
 * a function table for use by the feval function. It
 * then calls the function "mlxGasket". Finally, it
 * clears the feval table and exits.
 */
void mexFunction(int nlhs,
                 mxArray * * plhs,
                 int nrhs,
                 mxArray * * prhs) {
    mlfTry {
        mlfFunctionTableSetup(1, function_table);
        mclImportGlobal(0, NULL);
        mlxGasket(nlhs, plhs, nrhs, prhs);
        mlfFunctionTableTakedown(1, function_table);
    } mlfCatch {
        mlfFunctionTableTakedown(1, function_table);
        mclMexError();
    } mlfEndCatch
}
```

Main Files

You can generate C or C++ application wrappers that are suitable for building C or C++ stand-alone applications, respectively. These POSIX-compliant main wrappers accept strings from the POSIX shell and return a status code. They are meant to translate “command-like” M-files into POSIX main applications.

POSIX Main Wrapper

The POSIX `main()` function wrapper behaves exactly the same as the command/function duality mode of MATLAB. That is, any command of the form

```
command argument
```

can also be written in the functional form

```
command('argument')
```

If you write a function that accepts strings in MATLAB, that function will compile to a POSIX main wrapper in such a way that it behaves the same from the DOS/UNIX command line as it does from within MATLAB.

The Compiler processes the string arguments passed to the `main()` function and sends them into the compiled M-function as strings.

For example, consider this M-file:

```
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 Main Wrapper Function

The –W main –L C options produce a C application wrapper. This example uses the sample.m file and is produced from the command:

```
mcc –W main –L C –T codegen –F page–width:55 sample
```

The generated stand-alone C application wrapper function, sample_main.c is:

```
/*
 * MATLAB Compiler: 2.0b1
 * Date: Mon Dec 14 14:07:29 1998
 * Arguments: "–W" "main" "–L" "C" "–T"
 * "codegen" "–F" "page-width:55" "sample"
 */

#ifndef MLF_V2
#define MLF_V2 1
#endif

#include "matlab.h"
#include "sample.h"

static mlfFunctionTableEntry function_table[1]
  = { { "sample", mlxSample, –1, 1 } };

/*
 * The function "main" is a Compiler-generated main
 * wrapper, suitable for building a standalone
 * application. It initializes a function table for
 * use by the feval function, and then calls the
 * function "mlxSample". Finally, it clears the feval
 * table and exits.
 */
int main(int argc, const char * * argv) {
    int status = 0;
    mxArray * varargin = mclInitialize(NULL);
    mxArray * result = mclInitialize(NULL);
    mlfEnterNewContext(0, 0);
```

```
    mlfFunctionTableSetup(1, function_table);
    mlfAssign(
      &varargin,
      mclCreateCellFromStrings(argc – 1, argv + 1));
    mlfAssign(
      &result,
      mlfSample(
        mlfIndexRef(
          varargin, "{?}", mlfCreateColonIndex()),
        NULL));
    mxDestroyArray(varargin);
    if (mclIsInitialized(result)
        && ! mxIsEmpty(result)) {
        status = mclArrayToInt(result);
    }
    mxDestroyArray(result);
    mlfFunctionTableTakedown(1, function_table);
    mlfRestorePreviousContext(0, 0);
    return status;
}
```

C++ Wrapper Function

The `–W main –L Cpp` options produce a C++ application wrapper. This example uses the `sample.m` file and is produced from the command:

```
mcc –W main –L Cpp –T codegen –F page–width:55 sample
```

The generated stand-alone C++ application wrapper function, `sample_main.cpp` is:

```
//
// MATLAB Compiler: 2.0b1
// Date: Mon Dec 14 14:10:39 1998
// Arguments: "–W" "main" "–L" "Cpp" "–T"
// "codegen" "–F" "page-width:55" "sample"
//
#include "matlab.hpp"
#include "sample.hpp"

static mlfFunctionTableEntry function_table[1]
  = { { "sample", mlxSample, –1, 1 } };
```

```
static mwFunctionTableInit mcl_function_table_init(1,
                                                   function_table);

//
// The function "main" is a Compiler-generated main
// wrapper, suitable for building a standalone
// application. It initializes a function table for
// use by the feval function, and then calls the
// function "mlxSample". Finally, it clears the feval
// table and exits.
//
int main(int argc, const char * * argv) {
    try {
        int status(0);
        mwArray varargin(argc – 1, argv + 1);
        mwArray result;
        result
          = sample(
             mwVarargin(varargin.cell(colon())));
        if (result.IsInitialized()
            && ! result.IsEmpty()) {
            status = int(result(1));
        }
        return status;
    } catch(mwException e) {
        cout << e;
        return 1;
    }
}
```

Simulink S-Functions

The `–W simulink –L C` options produce a Simulink S-function wrapper. Simulink S-function wrappers conform to the Simulink C S-function conventions. The wrappers initialize:

- The `sizes` structure
- The S-function's sample times array
- The S-function's states and work vectors

For more information about Simulink S-function requirements, see the *Writing S-Functions* book.

Note By default, the –S command does not include any functions that do not appear on the command line. Functions that do not appear on the command line would generate a callback to MATLAB. Specify –h if you want all functions called to be compiled into your MEX-file.

This wrapper function uses the `fun.m` file and is produced from the command:

```
mcc –W simulink –L C –T codegen –F page–width:55 fun
```

Given the sample file, `fun.m`

```
function a = fun(b)
 a(1) = b(1) .* b(1);
 a(2) = b(1) + b(2);
 a(3) = b(2) / 4;
```

The generated Simulink S-function wrapper function, `fun_simulink.c` is:

```
/*
 * MATLAB Compiler: 2.0b1
 * Date: Mon Dec 14 14:26:41 1998
 * Arguments: "–W" "simulink" "–L" "C" "–T"
 * "codegen" "–F" "page-width:55" "fun"
 */

#ifndef MLF_V2
#define MLF_V2 1
#endif

#ifdef __cplusplus
extern "C" {
#endif

#define S_FUNCTION_NAME fun

#include "matlab.h"
#include "mccsimulink.h"
```

```
#include "fun.h"

static mlfFunctionTableEntry function_table[1]
  = { { "fun", mlxFun, 1, 1 } };

/*
 * The function mdlInitializeSizes is a Compiler
 * generated Simulink S-function wrapper. Simulink
 * calls this function to initialize the S-function's
 * "sizes" structure. The arguments to the –u and –y
 * mcc options are used to initialize the number of
 * inputs and outputs the S-function accepts and
 * returns. If –u or –y was not specified, this
 * function uses the value –1, which is the same as
 * the DYNAMICALLY_SIZED macro.
 */
static void mdlInitializeSizes(SimStruct * S) {
    mclInitializeSizes(S, –1, –1);
}

/*
 * The function mdlInitializeSampleTimes is a Compiler
 * generated Simulink S-function wrapper. Simulink
 * calls this function to initialize the S-function's
 * sample times array. This function initializes the
 * S-function's sample time to be inherited from the
 * block driving it.
 */
static void mdlInitializeSampleTimes(SimStruct * S) {
    mclInitializeSampleTimes(S);
}

/*
 * The function mdlInitializeConditions is a Compiler
 * generated Simulink S-function wrapper. Simulink
 * calls this function to initialize the S-function's
 * states and work vectors. Since Compiler-generated
 * S-functions have no states and no work vectors,
 * this function is empty.
 */
```

```
static void mdlInitializeConditions(real_T * x0,
                                    SimStruct * S) {
}

/*
 * The function mdlOutputs is a Compiler-generated
 * Simulink S-function wrapper. Simulink calls this
 * function to compute the S-function's output vector.
 * This function initializes a table for use by the
 * feval function and initializes any persistent
 * variables. It then calls the function "mlxFun".
 * Finally it clears the feval table and exits.
 */
static void mdlOutputs(real_T * y,
                       real_T const * x,
                       real_T const * u,
                       SimStruct * S,
                       int_T tid) {
    mlfTry {
        mlfFunctionTableSetup(1, function_table);
        mclImportGlobal(0, NULL);
        mclOutputs(y, x, u, S, tid, mlxFun);
        mlfFunctionTableTakedown(1, function_table);
    } mlfCatch {
        mlfFunctionTableTakedown(1, function_table);
        mclMexError();
    } mlfEndCatch
}

/*
 * The function mdlUpdate is a Compiler-generated
 * Simulink S-function wrapper. Simulink calls this
 * function to perform major time step updates.
 * Simulink does not call this function if the
 * S-function has no discrete states and has direct
 * feedthrough. Since Compiler-generated S-functions
 * meet these criteria, this function is never called,
 * and is therefore left empty.
 */
```

```
static void mdlUpdate(real_T * x,
                      real_T const * u,
                      SimStruct * S,
                      int_T tid) {
}

/*
 * The function mdlDerivatives is a Compiler-generated
 * Simulink S-function wrapper. Simulink calls this
 * function to compute derivatives for continuous
 * states. Since Compiler-generated S-functions are
 * stateless, this function is left empty.
 */
static void mdlDerivatives(real_T * dx,
                           real_T const * x,
                           real_T const * u,
                           SimStruct * S,
                           int_T tid) {
}

/*
 * The function mdlTerminate is a Compiler-generated
 * Simulink S-function wrapper. Simulink calls this
 * function to clean up the S-function at the
 * termination of a simulation. Since Compiler
 * generated S-functions need no cleanup, this
 * function is left empty.
 */
static void mdlTerminate(SimStruct * S) {
}
#ifdef  MATLAB_MEX_FILE    /*Is this file being compiled as a
                             MEX-file? */
#include "simulink.c"      /*MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /*Code generation registration
                             function */
#endif
```

```
#ifdef __cplusplus
}
#endif
```

C Libraries

The intent of the C library wrapper files is to allow the inclusion of an arbitrary set of M-files into a library or shared library. 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 should be exported from a C shared library.

Note Even if you are not producing a shared library, you must generate a library wrapper file when including any Compiler-generated code into a larger application.

This example uses several functions from the `toolbox\matlab\timefun` directory (`weekday`, `date`, `tic`, `calendar`, `toc`) to create a library wrapper. The `–W lib:sometimefun –L C` options produce the files shown in this table.

File	Description
`sometimefun.c`	C wrapper file
`sometimefun.h`	C header file
`sometimefun.exports`	C export list

sometimefun.c

The C wrapper file (`sometimefun.c`) contains the initialization (`sometimefunInitialize`) and termination (`sometimefunTerminate`) functions for the library. You must call `sometimefunInitialize` before you call any Compiler-generated code. This function initializes the state of Compiler-generated functions so that those functions can be called from C code not generated by the Compiler. You must also call `sometimefunTerminate` before you unload the library.

The library files in this example are produced from the command:

```
mcc –W lib:sometimefun –L C –F page–width:50 weekday date tic calendar toc
```

The generated wrapper function, `sometimefun.c` is:

```
/*
 * MATLAB Compiler: 2.0b1
 * Date: Mon Dec 14 14:36:29 1998
 * Arguments: "-W" "lib:sometimefun" "-L"
 * "C" "-F" "page-width:50" "weekday" "date"
 * "tic" "calendar" "toc"
 */

#ifndef MLF_V2
#define MLF_V2 1
#endif

#include "matlab.h"
#include "toc.h"
#include "calendar.h"
#include "tic.h"
#include "date.h"
#include "weekday.h"

mxArray * TICTOC = NULL;

static mlfFunctionTableEntry function_table[5]
  = { { "toc", mlxToc, 0, 1 },
      { "calendar", mlxCalendar, 2, 1 },
      { "tic", mlxTic, 0, 0 },
      { "date", mlxDate, 0, 1 },
      { "weekday", mlxWeekday, 1, 2 } };

/*
 * The function "sometimefunInitialize" is a
 * Compiler-generated initialization wrapper. It
 * is used to initialize the state of
 * Compiler-generated functions so that those
 * functions can be called from code not
 * generated by the Compiler. The function(s)
 * initialized can be found in the function_table
 * variable, appearing above.
 */
```

```
void sometimefunInitialize(void) {
    mlfAssign(&TICTOC, mclCreateGlobal());
    mlfFunctionTableSetup(5, function_table);
}

/*
 * The function "sometimefunTerminate" is a
 * Compiler-generated termination wrapper. It is
 * used to clean up the state that was set up by
 * the initialization wrapper function, also
 * found in this file. Call this function after
 * having called the initialization wrapper
 * function and after having finished making all
 * calls to the Compiler-generated function(s)
 * found in the function_table variable,
 * appearing above.
 */
void sometimefunTerminate(void) {
    mlfFunctionTableTakedown(5, function_table);
    mxDestroyArray(TICTOC);
}
```

sometimefun.h

The library header file (`sometimefun.h`) for this example is:

```
/*
 * MATLAB Compiler: 2.0b1
 * Date: Mon Dec 14 14:36:29 1998
 * Arguments: "-W" "lib:sometimefun" "-L"
 * "C" "-F" "page-width:50" "weekday" "date"
 * "tic" "calendar" "toc"
 */

#ifndef MLF_V2
#define MLF_V2 1
#endif

#ifndef __sometimefun_h
#define __sometimefun_h 1
```

```
#include "matlab.h"

extern mxArray * mlfWeekday(mxArray * * w,
                            mxArray * t);
extern void mlxWeekday(int nlhs,
                       mxArray * plhs[],
                       int nrhs,
                       mxArray * prhs[]);
extern mxArray * mlfDate(void);
extern void mlxDate(int nlhs,
                    mxArray * plhs[],
                    int nrhs,
                    mxArray * prhs[]);
extern void mlfTic(void);
extern void mlxTic(int nlhs,
                   mxArray * plhs[],
                   int nrhs,
                   mxArray * prhs[]);
extern mxArray * mlfNCalendar(int nargout,
                              mxArray * c,
                              mxArray * m);
extern mxArray * mlfCalendar(mxArray * c,
                             mxArray * m);
extern void mlfVCalendar(mxArray * c,
                         mxArray * m);
extern void mlxCalendar(int nlhs,
                        mxArray * plhs[],
                        int nrhs,
                        mxArray * prhs[]);
extern mxArray * mlfNToc(int nargout);
extern mxArray * mlfToc(void);
extern void mlfVToc(void);
extern void mlxToc(int nlhs,
                   mxArray * plhs[],
                   int nrhs,
                   mxArray * prhs[]);
extern void sometimefunInitialize(void);
extern void sometimefunTerminate(void);

#endif
```

`sometimefun.h` includes definitions for all publicly available symbols from the collection of M-files.

sometimefun.exports

The library export list file (`sometimefun.exports`) for this example is:

```
mlfWeekday
mlxWeekday
mlfDate
mlxDate
mlfTic
mlxTic
mlfNCalendar
mlfCalendar
mlfVCalendar
mlxCalendar
mlfNToc
mlfToc
mlfVToc
mlxToc
sometimefunInitialize
sometimefunTerminate
```

`sometimefun.exports` is a platform-independent list of symbols that is appropriate for exporting from a shared library.

C Shared Library

The MATLAB Compiler allows you to build a shared library from the files created in the previous section, "C Libraries." To build the shared library, `sometimefun.`*`ext`*, in one step, use

```
mcc –W lib:sometimefun –L C –t –T link:lib –h weekday date tic calendar toc
```

The `–t` option tells the Compiler to generate C code from each of the listed M-files. The `–T link:lib` option tells the Compiler to compile and link a shared library. The `–h` option tells the Compiler to include any other M-functions called from those listed on the `mcc` command line, i.e., helper functions.

C++ Libraries

The intent of the C++ library wrapper files is to allow the inclusion of an arbitrary set of M-files into a library. The header file contains all of the entry points for all of the compiled M functions.

Note Even if you are not producing a separate library, you must generate a library wrapper file when including any Compiler-generated code into a larger application.

This example uses several functions from the `toolbox\matlab\timefun` directory (`weekday`, `date`, `tic`, `calendar`, `toc`) to create a C++ library called `sometimefun`. The `–W lib:sometimefun –L Cpp` options produce the C++ library files shown in this table.

File	Description
`sometimefun.cpp`	C++ wrapper file
`sometimefun.hpp`	C++ header file

Note On some platforms, including Microsoft Windows NT, support for C++ shared libraries is limited and the C++ mangled function names must be exported. Refer to your vendor-supplied documentation for details on creating C++ shared libraries.

---

sometimefun.cpp

The C++ wrapper file (`sometimefun.cpp`) initializes the state of Compiler-generated functions so that those functions can be called from C++ code not generated by the Compiler. These files are produced from the command:

```
mcc –W lib:sometimefun –L Cpp –F page–width:55 weekday date tic calendar toc
```

The generated wrapper function, `sometimefun.cpp` is:

```
//
// MATLAB Compiler: 2.0b1
// Date: Mon Dec 14 14:43:48 1998
// Arguments: "-W" "lib:sometimefun" "-L"
// "Cpp" "-F" "page-width:55" "weekday" "date" "tic"
// "calendar" "toc"
//
#include "matlab.hpp"
#include "toc.hpp"
#include "calendar.hpp"
#include "tic.hpp"
#include "date.hpp"
#include "weekday.hpp"

mwArray TICTOC;

static mlfFunctionTableEntry function_table[5]
  = { { "toc", mlxToc, 0, 1 },
      { "calendar", mlxCalendar, 2, 1 },
      { "tic", mlxTic, 0, 0 },
      { "date", mlxDate, 0, 1 },
      { "weekday", mlxWeekday, 1, 2 } };

static mwFunctionTableInit mcl_function_table_init(5,
                                                   function_table);
```

sometimefun.hpp

The library header file (`sometimefun.hpp`) for this example is:

```
//
// MATLAB Compiler: 2.0b1
// Date: Mon Dec 14 14:43:48 1998
// Arguments: "-W" "lib:sometimefun" "-L"
// "Cpp" "-F" "page-width:55" "weekday" "date" "tic"
// "calendar" "toc"
//
#ifndef __sometimefun_hpp
#define __sometimefun_hpp 1
```

```
#include "matlab.hpp"

extern mwArray weekday(mwArray * w,
                       mwArray t = mwArray::DIN);
#ifdef __cplusplus
extern "C"
#endif
void mlxWeekday(int nlhs,
                mxArray * plhs[],
                int nrhs,
                mxArray * prhs[]);
extern mwArray date();
#ifdef __cplusplus
extern "C"
#endif
void mlxDate(int nlhs,
             mxArray * plhs[],
             int nrhs,
             mxArray * prhs[]);
extern void tic();
#ifdef __cplusplus
extern "C"
#endif
void mlxTic(int nlhs,
            mxArray * plhs[],
            int nrhs,
            mxArray * prhs[]);
extern mwArray Ncalendar(int nargout,
                         mwArray c = mwArray::DIN,
                         mwArray m = mwArray::DIN);
extern mwArray calendar(mwArray c = mwArray::DIN,
                        mwArray m = mwArray::DIN);
extern void Vcalendar(mwArray c = mwArray::DIN,
                      mwArray m = mwArray::DIN);
#ifdef __cplusplus
extern "C"
#endif
```

```
void mlxCalendar(int nlhs,
                 mxArray * plhs[],
                 int nrhs,
                 mxArray * prhs[]);
extern mwArray Ntoc(int nargout);
extern mwArray toc();
extern void Vtoc();
#ifdef __cplusplus
extern "C"
#endif
void mlxToc(int nlhs,
            mxArray * plhs[],
            int nrhs,
            mxArray * prhs[]);

#endif
```

Porting Generated Code to a Different Platform

The code generated by the MATLAB Compiler is portable among platforms. However, if you build an executable from `foo.m` on a PC running Windows, that same file will not run on a UNIX system.

For example, you cannot simply copy `foo.`*`mex`* (where the *`mex`* extension varies by platform) from a PC to a Sun system and expect the code to work, because binary formats are different on different platforms (all supported executable types are binary). However, you could copy either all of the generated C code or `foo.m` from the PC to the Sun system. Then, on the Sun platform you could use `mex` or `mcc` to produce a `foo.mex` that would work on the Sun system.

Note Stand-alone applications require that the MATLAB C/C++ Math Library be purchased for each platform where the Compiler-generated code will be executed. For more information, see the MATLAB C/C++ Math Library documentation.

Formatting Compiler-Generated Code

The formatting options allow you to control the look of the Compiler-generated C or C++ code. These options let you set the width of the generated code and the indentation levels for statements and expressions. To control code formatting, use

```
–F <option>
```

The remaining sections focus on the different choices you can use.

Listing All Formatting Options

To view a list of all available formatting options, use

```
mcc –F list …
```

Setting Page Width

Use the `page–width:`*n* option to set the maximum width of the generated code to *n*, an integer. The default is 80 columns wide, so not selecting any page width formatting option will automatically limit your columns to 80 characters.

Note Setting the page width to a desired value does not guarantee that all generated lines of code will not exceed that value. There are cases where, due to indentation perhaps, a variable name may not fit within the width limit. Since variable names cannot be split, they may extend beyond the set limit. Also, to maintain the syntactic integrity of the original M source, annotations included from the M source file are not wrapped.

Default Width

Not specifying a page width formatting option uses the default of 80. Using

```
mcc –x gasket
```

generates this code segment:

```
0        1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

for (mclForStart(&iterator_0, mlfScalar(1.0), numPoints, NULL);
         mclForNext(&iterator_0, &i);
         ) {
        /*
         * startPoint = floor((corners(theRand(i),:)+startPoint)/2);
         */
        mlfAssign(
          &startPoint,
          mlfFloor(
            mlfMrdivide(
              mlfPlus(
                mlfIndexRef(
                  corners,
                  "(?,?)",
                  mlfIndexRef(theRand, "(?)", i),
                  mlfCreateColonIndex()),
                startPoint),
              mlfScalar(2.0))));
            .
            .
            .
```

Page Width = 40

This example specifies a page width of 40.

```
mcc –x –F page–width:40 gasket
```

The segment of generated code is:

```
0        1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

mlfAssign(
      &theImage,
      mlfZeros(
        mlfScalar(1000.0),
        mlfScalar(1000.0),
        NULL));
    /*
     *
     * corners = [866 1;1 500;866 1000];
     */
    mlfAssign(
      &corners,
      mlfDoubleMatrix(
        3, 2, __Array0_r, NULL));
    /*
     * startPoint = [866 1];
     */
    mlfAssign(
      &startPoint,
      mlfDoubleMatrix(
        1, 2, __Array1_r, NULL));
    /*
     * theRand = rand(numPoints,1);
     */
    mlfAssign(
      &theRand,
      mlfRand(
        numPoints,
        mlfScalar(1.0),
        NULL));
    /*
     * theRand = ceil(theRand*3);
     */
    mlfAssign(
      &theRand,
      mlfCeil(
        mlfMtimes(
          theRand, mlfScalar(3.0))));

        .
        .
        .
```

Setting Indentation Spacing

Use the `statement–indent:n` option to set the indentation of all statements to `n`, an integer. The default is 4 spaces of indentation. To set the indentation for expressions, use `expression–indent:n`. This sets the number of spaces of indentation to `n`, an integer, and defaults to 2 spaces of indentation.

Default Indentation

Not specifying indent formatting options uses the default of four spaces for statements and two spaces for expressions. For example, using

```
mcc –x gasket
```

generates the following code segment:

```
0        1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

void mlxGasket(int nlhs, mxArray * plhs[], int nrhs, mxArray * prhs[]) {
    mxArray * mprhs[1];
    mxArray * mplhs[1];
    int i;
    if (nlhs > 1) {
        mlfError(
          mxCreateString(
            "Run-time Error: File: gasket Line: 1 Column: "
            "0 The function \"gasket\" was called with mor"
            "e than the declared number of outputs (1)"));
    }
    if (nrhs > 1) {
        mlfError(
          mxCreateString(
            "Run-time Error: File: gasket Line: 1 Column: "
            "0 The function \"gasket\" was called with mor"
            "e than the declared number of inputs (1)"));
    }
    for (i = 0; i < 1; ++i) {
        mplhs[i] = NULL;
    }
    for (i = 0; i < 1 && i < nrhs; ++i) {
        mprhs[i] = prhs[i];
    }
    for (; i < 1; ++i) {
        mprhs[i] = NULL;
    }
    mlfEnterNewContext(0, 1, mprhs[0]);
    mplhs[0] = Mgasket(nlhs, mprhs[0]);
    mlfRestorePreviousContext(0, 1, mprhs[0]);
    plhs[0] = mplhs[0];
}
```

Modified Indentation

This example shows the same segment of code using a statement indentation of 2 and an expression indentation of 1.

```
mcc –F statement–indent:2 –F expression–indent:1 –x gasket
```

generates the following code segment:

```
0        1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890

void mlxGasket(int nlhs, mxArray * plhs[], int nrhs, mxArray * prhs[]) {
  mxArray * mprhs[1];
  mxArray * mplhs[1];
  int i;
  if (nlhs > 1) {
    mlfError(
     mxCreateString(
      "Run-time Error: File: gasket Line: 1 Column: 0 The function \"gaske"
      "t\" was called with more than the declared number of outputs (1)"));
  }
  if (nrhs > 1) {
    mlfError(
     mxCreateString(
      "Run-time Error: File: gasket Line: 1 Column: 0 The function \"gaske"
      "t\" was called with more than the declared number of inputs (1)"));
  }
  for (i = 0; i < 1; ++i) {
    mplhs[i] = NULL;
  }
  for (i = 0; i < 1 && i < nrhs; ++i) {
    mprhs[i] = prhs[i];
  }
  for (; i < 1; ++i) {
    mprhs[i] = NULL;
  }
  mlfEnterNewContext(0, 1, mprhs[0]);
  mplhs[0] = Mgasket(nlhs, mprhs[0]);
  mlfRestorePreviousContext(0, 1, mprhs[0]);
  plhs[0] = mplhs[0];
}
```

Including M-File Information in Compiler Output

The annotation options allow you to control the type of annotation in the Compiler-generated C or C++ code. These options let you include the comments and/or source code from the initial M-file(s) as well as `#line` preprocessor directives. You can also use an annotation option to generate source file and line number information when you receive run-time error messages. To control code annotation, use:

```
–A <option>
```

You can combine annotation options, for example selecting both comments and `#line` directives. The remaining sections focus on the different choices you can use.

Controlling Comments in Output Code

Use the `annotation:type` option to include your initial M-file comments and code in your generated C or C++ output. The possible values for `type` are:

- `all`
- `comments`
- `none`

Not specifying any annotation type uses the default of `all`, which includes the complete source of the M-file (comments and code) interleaved with the generated C/C++ source.

The following sections show segments of the generated code from this simple Hello, World example.

```
function hello
% This is the hello, world function written in M code
% $Revision: 1.1 $
%
        fprintf(1,'Hello, World\n' );
```

Comments Annotation

To include only comments from the source M-file in the generated output, use:

```
mcc –A annotation:comments …
```

This code snippet shows the generated code containing only the comments.

Comments

```
static void Mhello(void) {
    mxArray * ans = mclInitializeAns();
    /*
     * This is the hello, world function written in M code
     * $Revision: 1.1 $
     *
     */
    mclAssignAns(
      &ans,
      mlfFprintf(mlfScalar(1.0), mxCreateString("Hello, World\\n"), NULL));
    mxDestroyArray(ans);
}
       .
       .
       .
```

All Annotation

To include both comments and source code from the source M-file in the generated output, use:

```
mcc –A annotation:all …
```

or do not stipulate the annotation option, thus using the default of `all`.

The code snippet contains both comments and source code:

Comments

Code

```
static void Mhello(void) {
    mxArray * ans = mclInitializeAns();
    /*
     * % This is the hello, world function written in M code
     * % $Revision: 1.1 $
     * %
     * fprintf(1,'Hello, World\n' );
     */
    mclAssignAns(
      &ans,
      mlfFprintf(mlfScalar(1.0), mxCreateString("Hello, World\\n"), NULL));
    mxDestroyArray(ans);
/*
 *
 */
}
       .
       .
       .
```

No Annotation

To include no source from the initial M-file in the generated output, use:

```
mcc –A annotation:none …
```

This code snippet shows the generated code without comments and source code.

```
static void Mhello(void) {
    mxArray * ans = mclInitializeAns();
    mclAssignAns(
      &ans,
      mlfFprintf(mlfScalar(1.0), mxCreateString("Hello, World\\n"), NULL));
    mxDestroyArray(ans);
}
       .
       .
       .
```

Controlling #line Directives in Output Code

`#line` preprocessing directives inform a C/C++ compiler that the C/C++ code was generated by another tool (MATLAB Compiler) and they identify the correspondence between the generated code and the original source code (M-file). You can use the `#line` directives to help debug your M-file(s). Most C language debuggers can display your M-file source code. These debuggers allow you to set breakpoints, single step, and so on at the M-file code level when you use the `#line` directives.

Use the `line:setting` option to include `#line` preprocessor directives in your generated C or C++ output. The possible values for `setting` are:

- `on`
- `off`

Not specifying any `line` setting uses the default of `off`, which does not include any `#line` preprocessor directives in the generated C/C++ source.

Include #line Directives

To include `#line` directives in your generated C or C++ code, use:

```
mcc –A line:on …
```

The Hello, World example produces the following code segment when this option is selected.

```
Line 1 ---->  #line 1 "<matlab>\\extern\\examples\\compiler\\hello.m"
              static void Mhello(void) {
Line 1 ---->      #line 1 "<matlab>\\extern\\examples\\compiler\\hello.m"
                  mxArray * ans = mclInitializeAns();
                  /*
                   * % This is the hello, world function written in M code
                   * % $Revision: 1.1 $
                   * %
                   * fprintf(1,'Hello, World\n' );
                   */
Line 5 ---->      #line 5 "<matlab>\\extern\\examples\\compiler\\hello.m"
                  mclAssignAns(
Line 5 ---->        #line 5 "<matlab>\\extern\\examples\\compiler\\hello.m"
                    &ans,
Line 5 ---->        #line 5 "<matlab>\\extern\\examples\\compiler\\hello.m"
                    mlfFprintf(mlfScalar(1.0), mxCreateString("Hello, World\\n"), NULL));
Line 5 ---->      #line 5 "<matlab>\\extern\\examples\\compiler\\hello.m"
                  mxDestroyArray(ans);

                     .
                     .
                     .
```

In this example, Line 1 points to lines in the generated C code that were produced by line 1 from the M-file, that is

```
function hello
```

Line 5 points to lines in the C code that were produced by line 5 of the M-file, or

```
fprintf(1,'Hello, World\n' );
```

Controlling Information in Run-Time Errors

Use the `debugline:setting` option to include source filenames and line numbers in run-time error messages. The possible values for `setting` are:

- `on`
- `off`

Not specifying any `debugline` setting uses the default of `off`, which does not include filenames and line numbers in the generated run-time error messages.

For example, given the M-file, `tmmult.m`, which in MATLAB would produce the error message `Inner matrix dimensions must agree`

```
function tmmult
        a = ones(2,3);
        b = ones(4,5);

        y = mmult(a,b)

function y = mmult( a, b )
        y = a * b;
```

If you create a Compiler-generated MEX-file with the command

```
mcc –x tmmult
```

and run it, your results are:

```
tmmult
??? Inner matrix dimensions must agree.

Error in ==> <matlab>\toolbox\compiler\tmmult.dll
```

The information about where the error occurred is not available. However, if you compile `tmmult.m` and use the `–A debugline:on` option as in

```
mcc –x –A debugline:on tmmult
```

your results are

File/line # ⟶

```
tmmult
??? Inner matrix dimensions must agree.
Error in File: "<matlab>\extern\examples\compiler\tmmult.m",
Function: "tmmult", Line: 5.

Error in ==> <matlab>\extern\examples\compiler\tmmult.dll
```

Note When using the –A `debugline:on` option, the `lasterr` function returns a string that includes the line number information. If, in your M-code, you compare against the string value of `lasterr`, you will get different behavior when using this option.

Since `try…catch…end` is not available in `g++`, do not use the –A `debugline:on` option.

Interfacing M-Code to C/C++ Code

The MATLAB Compiler 2.0 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); %Pre-allocate the matrix
for i = 1:100
   y(i) = collect_one;
end

function y = collect_one

persistent t;
if (isempty(t))
   t = 0;
else
   t = t + 0.05;
end
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 implementation version of the function (`Mf`) 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 *file*_external.h or *file*_external.hpp, where *file* is the name of the initial M-file 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.

The Compiler will still generate a .c or .cpp file from the .m file in question. The Compiler will generate the feval table, which includes the function and all of the required interface functions for the M-function, but the body of M-code from that function will be ignored. It will be replaced by the hand-written code. The Compiler will generate the interface for any functions that contain the %#external pragma into a separate file called *file*_external.h or *file*_external.hpp. 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); % pre-allocate the matrix
for i = 1:100
   y(i) = collect_one;
end

function y = collect_one

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

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

```
extern mxArray *Mcollect_collect_one( int nargout_ );
```

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 "matlab.h"
#include "collect_external.h"
#include <math.h>

extern double measure_from_device(void);

mxArray *Mcollect_collect_one( int nargout_ )
{
   return( mlfScalar( 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 C code and header file, use

```
mcc –c collect.m
```

By examining the Compiler-generated C code, you should easily be able to determine how to implement this interface. To compile `collect.m` to a MEX-file, use

```
mcc –x collect.m measure.c
```

Using feval

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

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

informs the MATLAB Compiler that the specified function(s) will be called through an `feval` call or through a MATLAB function that accepts a function to `feval` as an argument (e.g., `fmin` or the ode solvers). Without this pragma, the `–h` option will not be able to locate and compile all M-files used in your application.

If you are using the `%#function` pragma to define functions that are not available in M-code, you must write a dummy M-function that identifies the number of input and output parameters to the M-file function with the same name used on the `%#function` line. 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, such as:

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

and includes the `%#external` pragma. This statement indicates that the function takes three inputs (`a`, `b`, `c`) and returns a single output variable (`y`). No other lines need to be present in the M-function.

Print Handlers

A print handler is a routine that controls how your application displays the output generated by calls to `mlf` routines.

The system provides a default print handler for your application. The default print handler writes output to the standard output stream. If this print handler is suitable for your application, you do not need to write and register another print handler.

However, you can override the default behavior by writing and registering an alternative print handler. In fact, if you are coding a stand-alone application with a GUI, then you *must* register another print handler to display application output inside a GUI mechanism, such as a Windows message box or a Motif Label widget.

You write an alternative print handler routine in C or C++ and register its name at the beginning of your stand-alone application.

The way you register a print handler depends on whether or not "the `main` routine" (or first routine called) for your application is written in C or in M.

Note The print handlers and registration functions discussed in this section are written for C applications. Although they will work in C++ applications, we recommend that you use a C++ print handler and the C++ registration routine `mwSetPrintHandler()` for C++ applications. See the *MATLAB C++ Math Library User's Guide* for details about C++ print handlers.

Main Routine Written in C

If your main routine is coded in C (as opposed to being written as an M-file), you must:

- Register a print handler in your main routine.
- Write the print handler.

This section references source code from a sample stand-alone application written for Microsoft Windows. The main routine `WinMain` is written in C. The source code illustrates how to register and write a print handler.

The application is built from two files:

- `mrankwin.c`, which contains `WinMain`, `WinPrint` (the print handler), and a related function `WinFlush`
- The MATLAB Compiler C translation of `mrank.m`

Both `mrankwin.c` and `mrank.m` are located in the `<matlab>/extern/examples/compiler/` directory of your installation.

The `WinMain` routine in `mrankwin.c` is straightforward.

- It registers a print handler.
- It assigns an integer input from the command line to a scalar array, or defaults the contents of the array to 12.
- It passes that array to `mlfMrank`, which determines the rank of the magic squares from 1 to n.
- It prints the array returned by `mlfMrank`.

The first and last items in this list refer to print handlers.

Registering a Print Handler

To register a print handler routine, call the MATLAB C Math Library routine `mlfSetPrintHandler` as the first executable line in `WinMain` (or `main`). `mlfSetPrintHander` takes a single argument, a pointer to a print handler function.

For example, the first line of `WinMain` in `mrankwin.c` registers the print handler routine named `WinPrint` by calling `mlfSetPrintHandler`:

```
mlfSetPrintHandler(WinPrint);
```

Writing a Print Handler

Whenever an `mlf` function within a stand-alone application makes a request to write data, the application automatically intercepts the request and calls the registered print handler, passing the text to be displayed. In fact, the application calls the print handler once for *every* line of data to be output.

The print handler that you write must:

- Take a single argument of type `const char *` that points to the text to be displayed.
- Return `void`.

The print handler routine `WinPrint` in the example program illustrates one possible approach to writing a print handler for a Windows program.

When the example `WinMain` routine prints the array returned by `mlfMrank`,

```
mlfPrintMatrix(R);
```

the registered print handler, `WinPrint`, is called. If array `R` contains a 12-row result from the call to `mlfMrank`, the application calls `WinPrint` 12 times, each time passing the next line of data. The print handler `WinPrint` dynamically allocates a buffer to hold the printable contents of array `R` and appends each text string passed to it to the buffer.

In this design, the print handler prints to a buffer rather than the screen. A companion function `WinFlush` actually displays the 12 lines of data in a Windows message box.

In the example, `WinMain` calls `WinFlush` immediately following the call to `mlfPrintMatrix`.

```
mlfPrintMatrix(R);
WinFlush();
```

Though `WinFlush` is not part of the print handler, this implementation of a print handler requires that you call `WinFlush` after any `mlf` function that causes a series of calls to the print handler. For this short program, this design is appropriate.

Here is the source code from `mrankwin.c` for `WinPrint` and `WinFlush`. It includes:

- The static, global variables used by the two routines

```
static int totalcnt  = 0;
static int upperlim  = 0;
static int firsttime = 1;
char *OutputBuffer;
```

- The print handler routine itself that deposits the text passed to it in an output buffer

```
void WinPrint( char *text)
{
        int cnt;

        /* Allocate a buffer for the output */
        if (firsttime) {
                OutputBuffer = (char *)mxCalloc(1028, 1);
                upperlim += 1028;
                firsttime = 0;
        }

        /* Make sure there's enough room in the buffer */
        cnt = strlen(text);
        while (totalcnt + cnt >= upperlim) {
                char *TmpOut;
                upperlim += 1028;
              TmpOut = (char *)mxRealloc(OutputBuffer, upperlim);
                if (TmpOut != NULL)
                    OutputBuffer = TmpOut;
        }

        /* Concatenate the next line of text */
        strncat(OutputBuffer, text, cnt);

       /* Update the number of characters stored in the buffer */
        totalcnt += cnt;
}
```

- The related function `WinFlush` that actually displays the text from the output buffer in a Windows message box.

```
void WinFlush(void)
{
        MessageBox(NULL, OutputBuffer, "MRANK", MB_OK);
        mxFree(OutputBuffer);
}
```

For more details on `mlfPrintMatrix`, see the *MATLAB C Math Library User's Guide*.

Main Routine Written in M-Code

If your main routine is an M-file, you must:

- Write a print handler in C.
- Register the print handler in your main M-file.

Registering the print handler requires several steps, some performed in C and some in M-code. To register a print handler from your main M-file, you call a dummy print handler initialization function written in M-code. The MATLAB Compiler translates that call into a call to the actual print handler initialization function written in C or C++.

To set up for this translation, you must write two print handler initialization functions:

- A print handler initialization function in C or C++ that registers the print handler
- A dummy print handler initialization routine in M-code that does nothing (the body of the function is empty) except enable the MATLAB Compiler to make the proper translation

You call the dummy print handler initialization function from your main M-file. The MATLAB Compiler translates that call into a call to your print handler initialization function written in C or C++.

Example Files

In this example, two M-files and one C file are built into a stand-alone application. The main routine is `mr.m`.

- `mr.m` contains the main M routine.

```
function mr(m)
initprnt
m=str2num(m);
r=mrank(m);
r

function initprnt
%#external
```

- `mrank.m` determines the rank 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
```

- `myph.c` contains the print handler and the print handler initialization routine, in that order. In the example, this C file gets created.

```
#include "matlab.h"
#include “mr_external.h”
static void myPrintHandler(const char *s)
{
   printf("%s\n",s);
}

void Mmr_initprnt(void)
{
   mlfSetPrintHandler(myPrintHandler);
}
```

Writing the Print Handler in C/C++

First, write a print handler in C following the standard rules for a print handler: it must take one argument of type `const char *s` and return `void`.

The print handler in this example is very simple.

```
static void myPrintHandler(const char *s)
{
   printf("%s\n",s);
}
```

The file `myph.c` contains this code.

Registering the Print Handler

Registering the print handler requires several steps, some performed in C and some in M. Be careful to name your C and M print handler initialization functions according to the rules presented below. Otherwise, the correspondence between the two is missing.

Naming the Print Handler Initialization Routine in C. When you write the print handler initialization routine in C, you must follow the naming convention used by the MATLAB C Math Library. This name will appear in a header file that is generated by the MATLAB Compiler when it compiles the stub M-function, `initprnt` in this example. See the earlier section, "Interfacing M-Code to C/C++ Code," for more information.

You should include this Compiler-generated header file when you define the C function. For example, the print handler initialization routine developed here is called `MInitprnt` and is found in `mr_external.h`.

Naming the Dummy Print Handler Initialization Routine in M-Code. When you name the dummy print handler initialization routine in M-code, you must name it after the base part of the actual print handler initialization routine (the one written in C or C++).

For example, the dummy print handler initialization routine shown here is called `initprnt`.

Writing the Initialization Routine in C. First, write the print handler initialization routine in C. All print handler initialization functions register the name of the print handler function by calling `mlfSetPrintHandler`, passing a pointer to the print handler (the function name) as an argument.

Your initialization function must take no arguments and return `void`. For example,

```
void Mmr_initprnt(void)
{
   mlfSetPrintHandler(myPrintHandler);
}
```

The file `myph.c` contains this code.

Writing a Dummy Initialization Function in M-Code. Next, write the dummy print handler initialization routine in M-code. The body of this function is empty, but without the function declaration, the MATLAB Compiler can't successfully translate the call to `initprnt` in M-code into a call to `MInitprnt()` in C.

The function can be placed in the same M-file that defines the main `mr.m` in this example. It is declared as `function initprnt` and contains the `%#external` pragma.

Initializing the Print Handler in Your Main M-File. Call the dummy print handler initialization routine in the first executable line of your main M-file. For example, in `mr.m` the call to `initprnt` immediately follows the function declaration.

```
function mr(m)
initprnt;          % Call print handler initialization routine

m=str2num(m);

r=mrank(m);
r

function initprnt
%#external
```

Building the Executable

You must compile `myph.c` with one of the supported C compilers, and you must ensure that the resulting object file is linked into the stand-alone application. To build the C stand-alone executable, at the MATLAB prompt type:

```
mcc –t –L C –W main –T link:exe mr.m mrank.m myph.c
```

Testing the Executable

Run the executable by typing at the MATLAB prompt:

```
!mr 5
```

The output displays as

```
r =

     1

     2

     3

     3

     5
```

6

Reference

Pragmas

Pragmas are compiler-specific commands that provide special information to the compiler. This section contains the reference pages for the MATLAB Compiler 2.0 pragmas, namely, `%#external` and `%#function`.

Note The Compiler 1.2 pragmas documented in Appendix D, "Using Compiler 1.2," are recognized and ignored by Compiler 2.0.

Purpose

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

Syntax

```
%#external
```

Description

The `%#external` pragma informs the Compiler that the implementation version of the function (`Mf`) 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 *`file`*`_external.h` or *`file`*`_external.hpp`, where *`file`* is the name of the initial M-file 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" in Chapter 5.

%#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` call. 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.

Functions

This section contains the reference pages for the Compiler 2.0 functions. Many of these functions are included to maintain backward compatibility with previous versions of the Compiler.

Note In Compiler 2.0, the functions `mbchar`, `mbcharscalar`, `mbcharvector`, `mbint`, `mbintscalar`, `mbintvector`, `mbreal`, `mbrealscalar`, `mbrealvector`, `mbscalar`, and `mbvector` are not used for type imputation, but rather to test if values are a specific data type.

---

mbchar

Purpose

Assert variable is a MATLAB character string.

Syntax

```
mbchar(x)
```

Description

The statement

```
mbchar(x)
```

causes the MATLAB Compiler to impute that `x` is a `char` matrix. At runtime, if `mbchar` determines that `x` does not hold a `char` matrix, `mbchar` issues an error message and halts execution of the MEX-file.

`mbchar` tells the MATLAB interpreter to check whether `x` holds a `char` matrix. If `x` does not, `mbchar` issues an error message and halts execution of the M-file. The MATLAB interpreter does not use `mbchar` to impute `x`.

Note that `mbchar` only tests `x` at the point in an M-file or MEX-file where an `mbchar` call appears. In other words, an `mbchar` call tests the value of `x` only once. If `x` becomes something other than a `char` matrix after the `mbchar` test, `mbchar` cannot issue an error message.

A `char` matrix is any scalar, vector, or matrix that contains only the `char` data type.

Example

This code in MATLAB causes `mbchar` to generate an error message because `n` does not contain a `char` matrix:

```
n = 17;
mbchar(n);
??? Error using ==> mbchar
Argument to mbchar must be of class 'char'.
```

See Also

`mbcharvector`, `mbcharscalar`, `mbreal`, `mbscalar`, `mbvector`, `mbintscalar`, `mbintvector`

Purpose

Assert variable is a character scalar.

Syntax

```
mbcharscalar(x)
```

Description

The statement

```
mbcharscalar(x)
```

causes the MATLAB Compiler to impute that `x` is a character scalar, i.e., an unsigned short variable. At runtime, if `mbcharscalar` determines that `x` holds a value other than a character scalar, `mbcharscalar` issues an error message and halts execution of the MEX-file.

`mbcharscalar` tells the MATLAB interpreter to check whether `x` holds a character scalar value. If `x` does not, `mbcharscalar` issues an error message and halts execution of the M-file. The MATLAB interpreter does not use `mbcharscalar` to impute `x`.

Note that `mbcharscalar` only tests `x` at the point in an M-file or MEX-file where an `mbcharscalar` call appears. In other words, an `mbcharscalar` call tests the value of `x` only once. If `x` becomes a vector after the `mbcharscalar` test, `mbcharscalar` cannot issue an error message.

`mbcharscalar` defines a character scalar as any value that meets the criteria of both `mbchar` and `mbscalar`.

Example

This code in MATLAB generates an error message:

```
n = ['hello' 'world'];
mbcharscalar(n)
??? Error using ==> mbcharscalar
Argument of mbcharscalar must be scalar.
```

See Also

`mbchar`, `mbcharvector`, `mbreal`, `mbscalar`, `mbvector`, `mbintscalar`, `mbintvector`

mbcharvector

Purpose

Assert variable is a character vector, i.e., a MATLAB string.

Syntax

```
mbcharvector(x)
```

Description

The statement

```
mbcharvector(x)
```

causes the MATLAB Compiler to impute that `x` is a `char` vector. At runtime, if `mbcharvector` determines that `x` holds a value other than a `char` vector, `mbcharvector` issues an error message and halts execution of the MEX-file.

`mbcharvector` tells the MATLAB interpreter to check whether `x` holds a `char` vector value. If `x` does not, `mbcharvector` issues an error message and halts execution of the M-file. The MATLAB interpreter does not use `mbcharvector` to impute `x`.

Note that `mbcharvector` only tests `x` at the point in an M-file or MEX-file where an `mbcharvector` call appears. In other words, an `mbcharvector` call tests the value of `x` only once. If `x` becomes something other than a `char` vector after the `mbcharvector` test, `mbcharvector` cannot issue an error message.

`mbcharvector` defines a `char` vector as any value that meets the criteria of both `mbchar` and `mbvector`. Note that `mbcharvector` considers `char` scalars as `char` vectors as well.

Example

This code in MATLAB causes `mbcharvector` to generate an error message because, although `n` is a vector, `n` contains one value that is not a `char`:

```
n = [1:5];
mbcharvector(n)
??? Error using ==> mbcharvector
Argument to mbcharvector must be of class 'char'.
```

See Also

`mbchar`, `mbcharscalar`, `mbreal`, `mbscalar`, `mbvector`, `mbintscalar`, `mbintvector`

Purpose

Assert variable is integer.

Syntax

```
mbint(n)
```

Description

The statement

```
mbint(x)
```

causes the MATLAB Compiler to impute that `x` is an integer. At runtime, if `mbint` determines that `x` holds a noninteger value, the generated code issues an error message and halts execution of the MEX-file.

`mbint` tells the MATLAB interpreter to check whether `x` holds an integer value. If `x` does not, `mbint` issues an error message and halts execution of the M-file. The MATLAB interpreter does not use `mbint` to impute a data type to `x`.

Note that `mbint` only tests `x` at the point in an M-file or MEX-file where an `mbint` call appears. In other words, an `mbint` call tests the value of `x` only once. If `x` becomes a noninteger after the `mbint` test, `mbint` cannot issue an error message.

`mbint` defines an integer as any scalar, vector, or matrix that contains only integer or string values. For example, `mbint` considers `n` to be an integer because all elements in `n` are integers:

```
n = [5 7 9];
```

If even one element of `n` contains a fractional component, for example,

```
n = [5 7 9.2];
```

then `mbint` assumes that `n` is not an integer.

`mbint` considers all strings to be integers.

If `n` is a complex number, then `mbint` considers `n` to be an integer if both its real and imaginary parts are integers. For example, `mbint` considers the value of `n` an integer:

```
n = 4 + 7i
```

`mbint` does not consider the value of `x` an integer because one of the parts (the imaginary) has a fractional component:

```
x = 4 + 7.5i;
```

Example

This code in MATLAB causes `mbint` to generate an error message because `n` does not hold an integer value:

```
n = 17.4;
mbint(n);
??? Error using ==> mbint
Argument to mbint must be integer.
```

See Also

`mbintscalar`, `mbintvector`

Purpose

Assert variable is integer scalar.

Syntax

```
mbintscalar(n)
```

Description

The statement

```
mbintscalar(x)
```

causes the MATLAB Compiler to impute that `x` is an integer scalar. At runtime, if `mbintscalar` determines that `x` holds a value other than an integer scalar, `mbintscalar` issues an error message and halts execution of the MEX-file.

`mbintscalar` tells the MATLAB interpreter to check whether `x` holds an integer scalar value. If `x` does not, `mbintscalar` issues an error message and halts execution of the M-file. The MATLAB interpreter does not use `mbintscalar` to impute `x`.

Note that `mbintscalar` only tests `x` at the point in an M-file or MEX-file where an `mbintscalar` call appears. In other words, an `mbintscalar` call tests the value of `x` only once. If `x` becomes a vector after the `mbintscalar` test, `mbintscalar` cannot issue an error message.

`mbintscalar` defines an integer scalar as any value that meets the criteria of both `mbint` and `mbscalar`.

Example

This code in MATLAB causes `mbintscalar` to generate an error message because, although `n` is a scalar, `n` does not hold an integer value:

```
n = 4.2;
mbintscalar(n)
??? Error using ==> mbintscalar
Argument to mbintscalar must be integer.
```

See Also

`mbint`, `mbscalar`

mbintvector

Purpose

Assert variable is integer vector.

Syntax

```
mbintvector(n)
```

Description

The statement

```
mbintvector(x)
```

causes the MATLAB Compiler to impute that `x` is an integer vector. At runtime, if `mbintvector` determines that `x` holds a value other than an integer vector, `mbintvector` issues an error message and halts execution of the MEX-file.

`mbintvector` tells the MATLAB interpreter to check whether `x` holds an integer vector value. If `x` does not, `mbintvector` issues an error message and halts execution of the M-file. The MATLAB interpreter does not use `mbintvector` to impute `x`.

Note that `mbintvector` only tests `x` at the point in an M-file or MEX-file where an `mbintvector` call appears. In other words, an `mbintvector` call tests the value of `x` only once. If `x` becomes a two-dimensional matrix after the `mbintvector` test, `mbintvector` cannot issue an error message.

`mbintvector` defines an integer vector as any value that meets the criteria of both `mbint` and `mbvector`. Note that `mbintvector` considers integer scalars to be integer vectors as well.

Example

This code in MATLAB causes `mbintvector` to generate an error message because, although all the values of `n` are integers, `n` is a matrix rather than a vector:

```
n = magic(2)
n =
     1     3
     4     2
mbintvector(n)
??? Error using ==> mbintvector
Argument to mbintvect must be a vector.
```

See Also

`mbint`, `mbvector`, `mbintscalar`

Purpose

Assert variable is real.

Syntax

```
mbreal(n)
```

Description

The statement

```
mbreal(x)
```

causes the MATLAB Compiler to impute that `x` is real (not complex). At runtime, if `mbreal` determines that `x` holds a complex value, `mbreal` issues an error message and halts execution of the MEX-file.

`mbreal` tells the MATLAB interpreter to check whether `x` holds a real value. If `x` does not, `mbreal` issues an error message and halts execution of the M-file. The MATLAB interpreter does not use `mbreal` to impute `x`.

Note that `mbreal` only tests `x` at the point in an M-file or MEX-file where an `mbreal` call appears. In other words, an `mbreal` call tests the value of `x` only once. If `x` becomes complex after the `mbreal` test, `mbreal` cannot issue an error message.

A real value is any scalar, vector, or matrix that contains no imaginary components.

Example

This code in MATLAB causes `mbreal` to generate an error message because `n` contains an imaginary component:

```
n = 17 + 5i;
mbreal(n);
??? Error using ==> mbreal
Argument to mbreal must be real.
```

See Also

`mbrealscalar`, `mbrealvector`

mbrealscalar

Purpose

Assert variable is real scalar.

Syntax

```
mbrealscalar(n)
```

Description

The statement

```
mbrealscalar(x)
```

causes the MATLAB Compiler to impute that `x` is a real scalar. At runtime, if `mbrealscalar` determines that `x` holds a value other than a real scalar, `mbrealscalar` issues an error message and halts execution of the MEX-file.

`mbrealscalar` tells the MATLAB interpreter to check whether `x` holds a real scalar value. If `x` does not, `mbrealscalar` issues an error message and halts execution of the M-file. The MATLAB interpreter does not use `mbrealscalar` to impute `x`.

Note that `mbrealscalar` only tests `x` at the point in an M-file or MEX-file where an `mbrealscalar` call appears. In other words, an `mbrealscalar` call tests the value of `x` only once. If `x` becomes a vector after the `mbrealscalar` test, `mbrealscalar` cannot issue an error message.

`mbrealscalar` defines a real scalar as any value that meets the criteria of both `mbreal` and `mbscalar`.

Example

This code in MATLAB causes `mbrealscalar` to generate an error message because, although `n` contains only real numbers, `n` is not a scalar:

```
n = [17.2 15.3];
mbrealscalar(n)
??? Error using ==> mbrealscalar
Argument of mbrealscalar must be scalar.
```

See Also

`mbreal`, `mbscalar`, `mbrealvector`

Purpose

Assert variable is a real vector.

Syntax

```
mbrealvector(n)
```

Description

The statement

```
mbrealvector(x)
```

causes the MATLAB Compiler to impute that `x` is a real vector. At runtime, if `mbrealvector` determines that `x` holds a value other than a real vector, `mbrealvector` issues an error message and halts execution of the MEX-file.

`mbrealvector` tells the MATLAB interpreter to check whether `x` holds a real vector value. If `x` does not, `mbrealvector` issues an error message and halts execution of the M-file. The MATLAB interpreter does not use `mbrealvector` to impute `x`.

Note that `mbrealvector` only tests `x` at the point in an M-file or MEX-file where an `mbrealvector` call appears. In other words, an `mbrealvector` call tests the value of `x` only once. If `x` becomes complex after the `mbrealvector` test, `mbrealvector` cannot issue an error message.

`mbrealvector` defines a real vector as any value that meets the criteria of both `mbreal` and `mbvector`. Note that `mbrealvector` considers real scalars to be real vectors as well.

Example

This code in MATLAB causes `mbrealvector` to generate an error message because, although `n` is a vector, `n` contains one imaginary number:

```
n = [5 2+3i];
mbrealvector(n)
??? Error using ==> mbrealvector
Argument to mbrealvector must be real.
```

See Also

`mbreal`, `mbrealscalar`, `mbvector`

mbscalar

Purpose

Assert variable is scalar.

Syntax

```
mbscalar(n)
```

Description

The statement

```
mbscalar(x)
```

causes the MATLAB Compiler to impute that `x` is a scalar. At runtime, if `mbscalar` determines that `x` holds a nonscalar value, `mbscalar` issues an error message and halts execution of the MEX-file.

mbscalar tells the MATLAB interpreter to check whether `x` holds a scalar value. If `x` does not, `mbscalar` issues an error message and halts execution of the M-file. The MATLAB interpreter does not use `mbscalar` to impute `x`.

Note that `mbscalar` only tests `x` at the point in an M-file or MEX-file where an `mbscalar` call appears. In other words, an `mbscalar` call tests the value of `x` only once. If `x` becomes nonscalar after the `mbscalar` test, `mbscalar` cannot issue an error message.

`mbscalar` defines a scalar as a matrix whose dimensions are 1-by-1.

Example

This code in MATLAB causes `mbscalar` to generate an error message because `n` does not hold a scalar:

```
n = [1 2 3];
mbscalar(n);
??? Error using ==> mbscalar
Argument of mbscalar must be scalar.
```

See Also

`mbint`, `mbintscalar`, `mbintvector`, `mbreal`, `mbrealscalar`, `mbrealvector`, `mbvector`

Purpose

Assert variable is vector.

Syntax

```
mbvector(n)
```

Description

The statement

```
mbvector(x)
```

causes the MATLAB Compiler to impute that `x` is a vector. At runtime, if `mbvector` determines that `x` holds a nonvector value, `mbvector` issues an error message and halts execution of the MEX-file.

`mbvector` causes the MATLAB interpreter to check whether `x` holds a vector value. If `x` does not, `mbvector` issues an error message and halts execution of the M-file. The MATLAB interpreter does not use `mbvector` to impute `x`.

Note that `mbvector` only tests `x` at the point in an M-file or MEX-file where an `mbvector` call appears. In other words, an `mbvector` call tests the value of `x` only once. If `x` becomes a nonvector after the `mbvector` test, `mbvector` cannot issue an error message.

`mbvector` defines a vector as any matrix whose dimensions are 1-by-n or n-by-1. All scalars are also vectors (though most vectors are not scalars).

Example

This code in MATLAB causes `mbvector` to generate an error message because the dimensions of `n` are 2-by-2:

```
n = magic(2)
n =
     1     3
     4     2
mbvector(n)
??? Error using ==> mbvector
Argument to mbvector must be a vector.
```

See Also

`mbint`, `mbintscalar`, `mbintvector`, `mbreal`, `mbrealscalar`, `mbscalar`, `mbrealvector`

reallog

Purpose

Natural logarithm for nonnegative real inputs.

Syntax

```
Y = reallog(X)
```

Description

`reallog` is an elementary function that operates element-wise on matrices. `reallog` returns the natural logarithm of `X`. The domain of `reallog` is the set of all nonnegative real numbers. If `X` is negative or complex, `reallog` issues an error message.

`reallog` is similar to the MATLAB `log` function; however, the domain of `log` is much broader than the domain of `reallog`. The domain of `log` includes all real and all complex numbers. If `Y` is real, you should use `reallog` rather than `log` for two reasons.

First, subsequent access of `Y` may execute more efficiently if `Y` is calculated with `reallog` rather than with `log`. Using `reallog` forces the MATLAB Compiler to impute a real type to `X` and `Y`. Using `log` typically forces the MATLAB Compiler to impute a complex type to `Y`.

Second, the compiled version of `reallog` may run somewhat faster than the compiled version of `log`. (However, the interpreted version of `reallog` may run somewhat slower than the interpreted version of `log`.)

See Also

`exp`, `log`, `log2`, `logm`, `log10`, `realsqrt`

Purpose

Array power function for real-only output.

Syntax

```
Z = realpow(X,Y)
```

Description

`realpow` returns `X` raised to the `Y` power. `realpow` operates element-wise on matrices. The range of `realpow` is the set of all real numbers. In other words, if `X` raised to the `Y` power yields a complex answer, then `realpow` does not return an answer. Instead, `realpow` signals an error.

If `X` is negative and `Y` is not an integer, the resulting power is complex and `realpow` signals an error.

`realpow` is similar to the array power operator (`.^`) of MATLAB. However, the range of `.^` is much broader than the range of `realpow`. (The range of `.^` includes all real and all imaginary numbers.) If `X` raised to the `Y` power yields a complex answer, then you must use `.^` instead of `realpow`. However, if `X` raised to the `Y` power yields a real answer, then you should use `realpow` for two reasons.

First, subsequent access of `Z` may execute more efficiently if `Z` is calculated with `realpow` rather than `.^`. Using `realpow` forces the MATLAB Compiler to impute that `Z`, `X`, and `Y` are real. Using `.^` typically forces the MATLAB Compiler to impute the complex type to `Z`.

Second, the compiled version of `realpow` may run somewhat faster than the compiled version of `.^`. (However, the interpreted version of `realpow` may run somewhat slower than the interpreted version of `.^`.)

See Also

`reallog`, `realsqrt`

realsqrt

Purpose

Square root for nonnegative real inputs.

Syntax

```
Y = realsqrt(X)
```

Description

`realsqrt(X)` returns the square root of the elements of `X`. The domain of `realsqrt` is the set of all nonnegative real numbers. If `X` is negative or complex, `realsqrt` issues an error message.

`realsqrt` is similar to `sqrt`; however, `sqrt`'s domain is much broader than `realsqrt`'s. The domain of `sqrt` includes all real and all complex numbers. Despite this larger domain, if `Y` is real, then you should use `realsqrt` rather than `sqrt` for two reasons.

First, subsequent access of `Y` may execute more efficiently if `Y` is calculated with `realsqrt` rather than with `sqrt`. Using `realsqrt` forces the MATLAB Compiler to impute a real type to `X` and `Y`. Using `sqrt` typically forces the MATLAB Compiler to impute a complex type to `Y`.

Second, the compiled version of `realsqrt` may run somewhat faster than the compiled version of `sqrt`. (However, the interpreted version of `realsqrt` may run somewhat slower than the interpreted version of `sqrt`.)

See Also

`reallog`, `realpow`

Command Line Tools

This section contains the reference pages for the Compiler 2.0 command line tools, namely, `mbuild` and `mcc`. This section contains complete information about how to use the Compiler (`mcc`); Appendix A contains a summary of the options in a convenient table form.

mbuild

Purpose

Create an application using the MATLAB C/C++ Math Library.

Syntax

```
mbuild [–options] [filename1 filename2 …]
```

Description

`mbuild` is a script that supports various options that allow you to customize the building and linking of your code. This table lists the `mbuild` options. If no platform is listed, the option is available on both UNIX and Microsoft Windows.

Option	Description
`@filename`	(Windows) Replace `@filename` on the `mbuild` command line with the contents of `filename`.
`–c`	Compile only; do not link.
`–D<name>[=<def>]`	(UNIX) Define C preprocessor macro `<name>` as having value `<def>`.
`–D<name>`	(Windows) Define C preprocessor macro `<name>`.
`–f <file>`	(UNIX and Windows) Use `<file>` as the options file; `<file>` is a full pathname if the options file is not in current directory. (Not necessary if you use the `–setup` option.)
`–g`	Build an executable with debugging symbols included.
`–h[elp]`	Help; prints a description of `mbuild` and the list of options.
`–I<pathname>`	Add `<pathname>` to the Compiler `include` search path.
`–l<file>`	(UNIX) Link against library `lib<file>`.
`–L<pathname>`	(UNIX) Include `<pathname>` in the list of directories to search for libraries.

Option	Description
–lang <language>	Override language choice implied by file extension. <language> = c for C (default) cpp for C++ This option is necessary when you use an unsupported file extension, or when you pass in all .o files and libraries.
–link <target>	Specify the type of output file. <target> = exe for executable file (default) shared for shared library
<name>=<def>	(UNIX) Override options file setting 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'.
–n	(UNIX) No execute flag. Using this option causes the commands used to compile and link the target to be displayed without executing them.
–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 <name>	Create an executable named <name>. (An appropriate executable extension is automatically appended.)

Option	Description
`-O`	Build an optimized executable.
`-setup`	Set up default options file. This option should be the only argument passed.
`-U<name>`	Undefine C preprocessor macro `<name>`.
`-v`	Verbose; print all compiler and linker settings.

Purpose

Invoke MATLAB Compiler 2.0.

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).

Note Compiler 2.0 is the default compiler. If you want to use Compiler 1.2, you must include the –V1.2 option on the mcc command line. See "mcc (Compiler 1.2)" for the mcc reference page for Compiler 1.2.

Command Line Syntax

You may specify one or more MATLAB Compiler option flags to mcc. (The complete list of option flags appears later in this reference page.) Most option flags have a one-letter name. You can list options separately on the command line, for example:

```
mcc –m –g myfun
```

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 –m –A full myfun       % Options listed separately
mcc –mA full myfun         % Options combined, A option last
```

This format is *not* valid:

```
mcc –Am full myfun         % Options combined, A option not last
```

In cases where you have more than one option that take 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.

Note The –V1.2 option cannot be combined with other options; it must stand by itself. For example, you cannot use

```
mcc –V1.2ir myfun
```

You would use

```
mcc –V1.2 –ir myfun
```

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

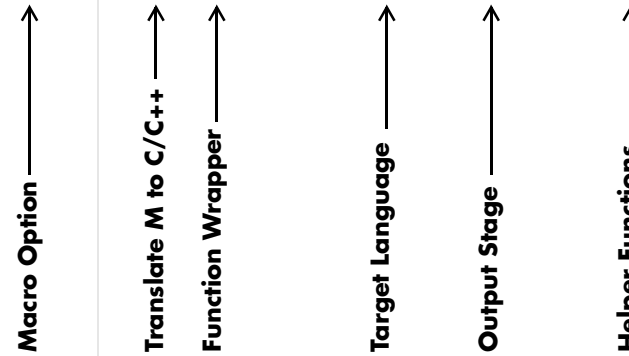
Simplifying the Compilation Process

Compiler 2.0, 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. If you want to take advantage of the power of the Compiler, you can do whatever you desire to do by choosing various Compiler options.

Table 6-1 shows the relationship between the simple, macro approach to accomplish a standard compilation and the more advanced, multi-option alternative.

Table 6-1: Basic Compiler Option Equivalencies

To build a	Simple	Advanced
Stand-alone C application	`-m`	`-t -W main -L C -T link:exe -h`
Stand-alone C++ application	`-p`	`-t -W main -L Cpp -T link:exe -h`
MEX-function	`-x`	`-t -W mex -L C -T link:mex`
Simulink S-function	`-S`	`-t -W simulink -L C -T link:mex`



The remainder of this reference page provides complete descriptions of these options.

Differences Between Compiler 2.0 and Compiler 1.2 Options

Although most of the Compiler options perform the same functions in Compiler 2.0 and Compiler 1.2, there are several options whose functions differ. In particular, some of the macros described in the previous section, "Simplifying the Compilation Process," behave differently depending on which Compiler you

use. Table 6-2 describes the functionality of the macro options in Compiler 2.0 and their corresponding functionality in Compiler 1.2.

Table 6-2: Macro Options

Option	Compiler 2.0	Compiler 1.2
`m`	Generates a stand-alone C application.	Generates a C function named `main`.
`p`	Generates a stand-alone C++ application.	Generates a stand-alone C++ application when the file name is `main.m` or a C++ file is specified.
`x`	Generates a MEX-function.	Did not exist. (It was the default.)
`S`	Generates a Simulink MEX-function.	Generates a Simulink MEX-function.

To use the Compiler 1.2 version of these options, you must specify the `–V1.2` option. For example, to generate a C function named `main` using Compiler 1.2, use:

```
mcc –V1.2 –m myfunc
```

To generate a stand-alone C application using Compiler 2.0, use:

```
mcc –m myfunc
```

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:

1. The current working directory, or
2. `$HOME/matlab` (UNIX) or `<matlab>\bin` (PC)

`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.

Setting a MATLAB Path in the Stand-Alone MATLAB Compiler

Unlike the MATLAB version of the Compiler, which inherits a MATLAB path from MATLAB, the stand-alone version has no initial path. If you want to set up a default path, you can do so by making an mccpath file. To do this:

1. Create a text file containing the text –I <your_directory_here> for each directory you want on the default path, and name this file mccpath. (Alternately, you can call the MCCSAVEPATH M-function from MATLAB to create an mccpath file.)
2. Place this file in one of two directories:

   The current working directory, or

   $HOME/matlab (UNIX) or <matlab>\bin (PC)

The stand-alone version of the MATLAB Compiler searches for the mccpath file in these two directories in the order shown above. If it finds an mccpath file, it processes the directories specified within the file and uses them to initialize its search path. Note that you may still use the –I option on the command line or in mccstartup files to add other directories to the search path. Directories specified this way are searched after those directories specified in the mccpath file.

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 6-1,

```
mcc –m –W none test.m
```

is equivalent to

```
mcc –t –W main –L C –T link:exe –h –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.

Handling Full Pathnames

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

1. Breaks the full name into the corresponding path- 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.

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.

Compiling Embedded M-Files

If the M-file you are compiling calls other M-files, you can list the called M-files on the command line. Doing so causes the MATLAB Compiler to build all the M-files into a single MEX-file, which usually executes faster than separate MEX-files. Note, however, that the single MEX-file has only one entry point regardless of the number of input M-files. The entry point is the first M-file on the command line. For example, suppose that `bell.m` calls `watson.m`. Compiling with

```
mcc –x bell watson
```

creates `bell.mex`. The entry point of `bell.mex` is the compiled code from `bell.m`. The compiled version of `bell.m` can call the compiled version of `watson.m`. However, compiling as

```
mcc –x watson bell
```

creates `watson.mex`. The entry point of `watson.mex` is the compiled code from `watson.m`. The code from `bell.m` never gets executed.

As another example, suppose that `x.m` calls `y.m` and that `y.m` calls `z.m`. In this case, make sure that `x.m` is the first M-file on the command line. After `x.m`, it does not matter which order you specify `y.m` and `z.m`.

MATLAB Compiler 2.0 Option Flags

The MATLAB Compiler option flags perform various functions that affect the generated code and how the Compiler behaves. Table 6-3 shows the categories of the Compiler options.

Table 6-3: Compiler Option Categories

Category	Purpose
Macros	The macro options simplify the compilation process by combining the most common compilation tasks into single options.
Code Generation	These options affect the actual code that the Compiler generates. For example, `–L` specifies the target language as either C or C++.
Compiler and Environment	These options provide information to the Compiler such as where to put (`–d`) and find (`–I`) particular files.
`mbuild/mex`	These options provide information for the `mbuild` and/or `mex` scripts.

The remainder of this reference page is subdivided into sections that correspond to the Compiler option categories. Each section provides a full description of all of the options in the category.

Note If you use the option flags that optimize the generated code, you must use the `–V1.2` option, making Compiler 1.2 the active compiler. The `–V1.2` option is not supported in the stand-alone Compiler.

Macro Options

The macro options provide a simplified way to accomplish basic compilation tasks.

-m (Stand-Alone C)

Produce a stand-alone C application. It includes helper functions by default (`–h`), and then generates a stand-alone C wrapper (`–W main`). In the final stage, this option compiles your code into a stand-alone executable and links it to the MATLAB C/C++ Math Library (`–T link:exe`). For example, to translate an M-file named `mymfile.m` into C and to create a stand-alone executable that can be run without MATLAB, use:

```
mcc –m mymfile
```

The `–m` option is equivalent to the series of options:

```
–t –W main –L C –T link:exe –h
```

-p (Stand-Alone C++)

Produce a stand-alone C++ application. It includes helper functions by default (`–h`), and then generates a stand-alone C++ wrapper (`–W main`). In the final stage, this option compiles your code into a stand-alone executable and links it to the MATLAB C/C++ Math Library (`–T link:exe`). For example, to translate an M-file named `mymfile.m` into C++ and to create a stand-alone executable that can be run without MATLAB, use:

```
mcc –p mymfile
```

The `–p` option is equivalent to the series of options:

```
–t –W main –L Cpp –T link:exe –h
```

-S (Simulink S-Function)

Produce a Simulink S-function that is compatible with the Simulink S-function block. For example, to translate an M-file named `mymfile.m` into C and to create the corresponding Simulink S-function using dynamically sized inputs and outputs, use:

```
mcc –S mymfile
```

The `–S` option is equivalent to the series of options:

```
–t –W simulink –L C –T link:mex
```

-x (MEX-Function)

Produce a MEX-function. For example, to translate an M-file named `mymfile.m` into C and to create the corresponding MEX-file that can be called directly from MATLAB, use:

```
mcc –x mymfile
```

The –x option is equivalent to the series of options:

```
–t –W mex –L C –T link:mex
```

Code Generation Options

-A (Annotation Control for Output Source)

Control the type of annotation in the resulting C/C++ source file. The types of annotation you can control are:

- M-file code and/or comment inclusion (`annotation`)
- `#line` preprocessor directive inclusion (`line`)
- Whether error messages report the source file and line number (`debugline`)

To control the M-file code that is included in the generated C/C++ source, use:

```
mcc –A annotation:type …
```

Table 6-4 shows the available types of code and comment annotation options.

Table 6-4: Code/Comment Annotation Options

type	Description
`all`	Provides the complete source of the M-file interleaved with the generated C/C++ source. The default is `all`.
`comments`	Provides all of the comments from the M-file interleaved with the generated C/C++ source.
`none`	No comments or code from the M-file are added to code.

To control the `#line` preprocessor directives that are included in the generated C/C++ source, use:

```
mcc –A line:setting …
```

Table 6-5 shows the available `#line` directive settings.

Table 6-5: Line Annotation Options

Setting	Description
`on`	Adds `#line` preprocessor directives to the generated C/C++ source code to enable source M-file debugging.
`off`	Adds no `#line` preprocessor directives to the generated C/C++ source code. The default is `off`.

To control if run-time error messages report the source file and line number, use:

```
mcc –A debugline:on …
```

Table 6-6 shows the available `debugline` directive settings.

Table 6-6: Run-Time Error Annotation Options

Setting	Description
`on`	Specifies the presence of source file and line number information in run-time error messages.
`off`	Specifies no source file and line number information in run-time error messages. The default is `off`.

For example:

To include all of your M-code, including comments, in the generated file and the standard #line preprocessor directives, use:

```
mcc –A annotation:all –A line:on …
```
or
`mcc –A line:on …` (The default is all for code/comment inclusion.)

To include none of your M-code and no `#line` preprocessor directives, use:

```
mcc –A annotation:none –A line:off …
```

To include the standard `#line` preprocessor directives in your generated C/C++ source code as well as source file and line number information in your run-time error messages, use:

```
mcc –A line:on –A debugline:on …
```

-F <option> (Formatting)

Control the formatting of the generated code. Table 6-7 shows the available options.

Table 6-7: Formatting Options

<Option>	Description
`list`	Generates a table of all the available formatting options.
`expression-indent:n`	Sets the number of spaces of indentation for all expressions to `n`, where `n` is an integer. The default indent is 4.
`page-width:n`	Sets maximum width of generated code to `n`, where `n` is an integer. The default width is 80.
`statement-indent:n`	Sets the number of spaces of indentation for all statements to `n`, where `n` is an integer. The default indent is 2.

-l (Line Numbers)

Generate C/C++ code that prints filename and line numbers on run-time errors. This option flag is useful for debugging, but causes the executable to run slightly slower. This option is equivalent to:

```
mcc –A debugline:on …
```

-L <language> (Target Language)

Specify the target language of the compilation. Possible values for language are `C` or `Cpp`. The default is `C`. Note that these values are case insensitive.

-u (Number of Inputs)

Provide more control over the number of valid inputs for your Simulink S-function. This option specifically sets the number of inputs (`u`) for your function. If –`u` is omitted, the input will be dynamically sized. (Used with –`S` option.)

-W <type> (Function Wrapper)

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. Table 6-8 shows the valid type options.

Table 6-8: Function Wrapper Types

<Type>	Description
`mex`	Produces a `mexFunction()` interface.
`main`	Produces a POSIX shell `main()` function.
`simulink`	Produces a Simulink C MEX S-function interface.
`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 non-static function names.
`none`	Does not produce a wrapper file. `none` is the default.

Caution When generating function wrappers, you *must* specify all M-files that are being linked together on the command line. These files are used to produce the initialization and termination functions as well as global variable definitions. If the functions are not specified in this manner, undefined symbols will be produced at link time.

-y (Number of Outputs)

Provides more control over the number of valid outputs for your Simulink S-function. This option specifically sets the number of outputs (`y`) for your function. If `–y` is omitted, the output will be dynamically sized. (Used with `–S` option.)

Compiler and Environment Options

-B <filename> (Bundle of Compiler Settings)

Replace `–B <filename>` on the `mcc` command line 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.You can place options that you always set in an `mccstartup` file. For more information, see “Setting Up Default Options.”

-c (C Code Only)

Generate C code but do not invoke `mex` or `mbuild`, i.e., do not produce a MEX-file or stand-alone application. This is equivalent to `–T codegen` placed at the end of the `mcc` command line.

-d <directory> (Output Directory)

Place the output files from the compilation in the directory specified by the `–d` option.

-h (Helper Functions)

Compile helper functions by default. Any helper functions that are called will be compiled into the resulting MEX or stand-alone application. The –m option automatically compiles all helper functions, so –m effectively calls –h.

Using the –h option is equivalent to listing the M-files explicitly on the mcc command line.

The –h option purposely does not include built-in functions or functions that appear in the MATLAB M-File Math Library portion of the C/C++ Math Libraries. This prevents compiling functions that are already part of the C/C++ Math Libraries. If you want to compile these functions as helper functions, you should specify them explicitly on the command line. For example, use

```
mcc –m minimize_it fmins
```

instead of

```
mcc –m –h minimize_it
```

-I <directory> (Directory 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.

-o <outputfile>

Specify the basename of the final executable output (MEX-file or application) of the Compiler. A suitable, possibly platform-dependent, extension is added to the specified basename (e.g., .exe for PC stand-alone applications, .mexsol for Solaris MEX-files).

-t (Translate M to C/C++)

Translate M-files specified on the command line to C/C++ files.

-T <target> (Output Stage)

Specify the desired output stage. Table 6-9 gives the possible values of `target`.

Table 6-9: Output Stage Options

<Target>	Description
`codegen`	Translates M-files to C/C++ files. The default is `codegen`.
`compile:<bin>`	Translates M-files to C/C++ files; compiles to object form.
`link:<bin>`	Translates M-files to C/C++ files; compiles to object form; links to executable form (MEX or stand-alone application.)

where `<bin>` can be `mex`, `exe`, or `lib`. `mex` uses the `mex` script to build a MEX-file; `exe` uses the `mbuild` script to build an executable; `lib` uses `mbuild` to build a shared library.

-v (Verbose)

Display the steps in compilation, 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 `mex` or `mbuild`

The –v option passes the –v option to `mex` or `mbuild` and displays information about `mex` or `mbuild`.

-V1.2 (MATLAB Compiler 1.2)

Invoke the MATLAB Compiler 1.2. This option is not supported in the stand-alone Compiler mode; it works only from the MATLAB prompt. If you obtained good optimization from Compiler 1.2, this option enables you to continue to benefit from the performance advantages of that Compiler. For more information about the MATLAB Compiler 1.2 options, see the “mcc (Compiler 1.2)” reference page.

-V2.0 (MATLAB Compiler 2.0)

Invoke the MATLAB Compiler 2.0. This option works from both the MATLAB prompt and the DOS or UNIX command line.

-w (Warning)

Display warning messages. Table 6-10 shows the various ways you can use the –w option.

Table 6-10: Warning Option

Syntax	Description
(no –w option)	Default; displays only serious warnings.
`–w list`	Generates a table that maps `<string>` to warning message for use with `enable`, `disable`, and `error`. Appendix B lists the same information.
`–w`	Enables complete warnings.
`–w disable[:<string>]`	Disables specific warning associated with *<string>*. Appendix B 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 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.

-Y <license.dat File>

Use license information in `license.dat` file when checking out a Compiler license.

mbuild/mex Options

-f <filename> (Specifying Options File)

Use the specified options file when calling `mex` or `mbuild`. This option allows you to use different compilers for different invocations of the MATLAB Compiler. This option is a direct pass-through to the `mex` or `mbuild` script. See the *Application Program Interface Guide* for more information about using this option with the `mex` script.

Note Although this option works as documented, it is suggested that you use `mex –setup` or `mbuild –setup` to switch compilers.

-g (Debugging Information)

Cause `mex` or `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 MEX-file or stand-alone application with a debugger.

The `–g` option flag has no influence on the source code that the MATLAB Compiler generates, though it does have some influence on the binary code that the C/C++ compiler generates.

-M "string" (Direct Pass Through)

Pass `string` directly to the `mex` or `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 last `–M` option is used.

-z <path> (Specifying Library Paths)

Specify the path to use for library and include files. This option uses the specified path for compiler libraries instead of the path returned by `matlabroot`.

Examples

Make a C translation and a MEX-file for `myfun.m`:

```
mcc –x myfun
```

Make a C translation and a stand-alone executable for `myfun.m`:

```
mcc –m myfun
```

Make a C++ translation and a stand-alone executable for `myfun.m`:

```
mcc –p myfun
```

Make a C translation and a Simulink S-function for `myfun.m` (using dynamically sized inputs and outputs):

```
mcc –S myfun
```

Make a C translation and a Simulink S-function for `myfun.m` (explicitly calling for one input and two outputs):

```
mcc –S –u 1 –y 2 myfun
```

Make a C translation and 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 C translation and a MEX-file for `myfun.m`. Also translate and include all M-functions called directly or indirectly by `myfun.m`. Incorporate the full text of the original M-files into their corresponding C files as C comments:

```
mcc –x –h –A annotation:all myfun
```

Make a generic C translation of `myfun.m`:

```
mcc –t –L C myfun
```

Make a generic C++ translation of `myfun.m`:

```
mcc –t –L Cpp myfun
```

Make a C MEX wrapper file from `myfun1.m` and `myfun2.m`:

```
mcc –W mex –L C myfun1 myfun2
```

Make a C translation and a stand-alone executable from `myfun1.m` and `myfun2.m` (using one `mcc` call):

```
mcc –m myfun1 myfun2
```

Make a C translation and a stand-alone executable from `myfun1.m` and `myfun2.m` (by generating each output file with a separate `mcc` call):

```
mcc –t –L C myfun1                 % yields myfun1.c
mcc –t –L C myfun2                 % yields myfun2.c
mcc –W main –L C myfun1 myfun2     % yields myfun1_main.c
mcc –T compile:exe myfun1.c        % yields myfun1.o
mcc –T compile:exe myfun2.c        % yields myfun2.o
mcc –T compile:exe myfun1_main.c   % yields myfun1_main.o
mcc –T link:exe myfun1.o myfun2.o myfun1_main.o
```

Note On PCs, filenames ending with `.o` above would actually end with `.obj`.

Purpose

Invoke MATLAB Compiler 1.2.

Syntax

```
mcc –V1.2 [–options] mfile1 [mfile2 ... mfileN]
                            [fun1=feval_arg1 ... funN=feval_argN]
```

Description

By including the –V1.2 option on the mcc command line, you tell the Compiler that you want to use Compiler 1.2. Compiler 2.0 is the default; not specifying the –V1.2 option automatically uses Compiler 2.0.

Note This reference page provides information on Compiler 1.2. See "mcc (Compiler 2.0)" for the mcc reference page for Compiler 2.0. You can make Compiler 1.2 your default by using an mccstartup file. See "Setting Up Default Options" in this chapter for complete details.

---

mcc is the MATLAB command that invokes the MATLAB Compiler. You *must* issue the mcc –V1.2 command from the MATLAB prompt to use Compiler 1.2. When using Compiler 1.2, the only additional required argument to mcc is the name of an M-file. For example, the command to compile the M-file stored in file myfun.m is:

```
mcc –V1.2 myfun
```

If you specify a relative pathname for an M-file, the M-file must be in a directory on your MATLAB search path, or in your current directory.

Specifying Options

You may specify one or more MATLAB Compiler option flags to mcc. (The list of options appears later in this reference page.) Most options have a one-letter name. Precede the list of option flags with a single dash (–), or list the options separately. For example, either syntax is acceptable:

```
mcc –V1.2 –ir myfun
mcc –V1.2 –i –r myfun
```

Note The –V1.2 option cannot be combined with other options; it must stand by itself. For example, you cannot use

```
mcc –V1.2ir myfun
```

You would use

```
mcc –V1.2 –ir myfun
```

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:

1. The current working directory, or
2. $HOME/matlab (UNIX) or <matlab>\bin (PC)

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.

Building a MEX-File From Multiple M-Files

If the M-file you are compiling calls other M-files, you can list the called M-files on the command line. Doing so causes the MATLAB Compiler to build all the M-files into a single MEX-file, which usually executes faster than separate MEX-files. Note, however, that the single MEX-file has only one entry point regardless of the number of input M-files. The entry point is the first M-file on the command line. For example, suppose that bell.m calls watson.m. Compiling with

```
mcc –V1.2 bell watson
```

creates `bell.mex`. The entry point of `bell.mex` is the compiled code from `bell.m`. The compiled version of `bell.m` can call the compiled version of `watson.m`. However, compiling as

```
mcc –V1.2 watson bell
```

creates `watson.mex`. The entry point of `watson.mex` is the compiled code from `watson.m`. The code from `bell.m` never gets executed.

As another example, suppose that `x.m` calls `y.m` and that `y.m` calls `z.m`. In this case, make sure that `x.m` is the first M-file on the command line. After `x.m`, it does not matter which order you specify `y.m` and `z.m`.

Calling feval

If the M-file you are compiling contains a call to `feval`, you can use the `fun=`*`feval_arg`* syntax to specify the name of the function to be passed to `feval`.

Do not put any spaces on either side of the equals sign. Consider these right and wrong ways to specify a *`feval_arg`*:

```
mcc –V1.2 citrus fun1=lemon    % correct
mcc –V1.2 citrus fun1 = lemon  % incorrect
```

MATLAB Compiler 1.2 Option Flags

Some MATLAB Compiler option flags optimize the generated code, other option flags generate compilation or runtime information, and some option flags are Simulink-specific.

If you use the MATLAB Compiler `mcc` option flag –p to generate C++ code, then these options are not supported:

- `–i`
- `–l`
- `–r`

-B <filename> (Bundle of Compiler Settings)

Replace `–B <filename>` on the `mcc` command line 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. You can place options that you always set in an mccstartup file. For more information, see "Setting Up Default Options."

-c (C/C++ Code Only)

Generate C/C++ code but do not invoke mex or mbuild, i.e., do not produce a MEX-file.

-e (Stand-Alone External C Code)

Generate C code for stand-alone applications. The resulting C code cannot be used to create MEX-files; it can be linked with the MATLAB C Math Library and executed as a stand-alone application. Stand-alone applications do not require MATLAB at runtime. Stand-alone applications can run even if MATLAB is not installed on the system. See Chapter 4 for complete details on stand-alone applications.

If you specify –e, the MATLAB Compiler does not invoke mex, consequently, it does not produce a MEX-file. However, it does invoke mbuild to create a stand-alone application when the name of the M-file is main.

-f <filename> (Specifying Options File)

Use the specified options file when calling mex or mbuild. This option allows you to use different C or C++ compilers for different invocations of the MATLAB Compiler. This option is a direct pass-through to the mex script. See the *Application Program Interface Guide* for more information about using this option with the mex script.

Note Although this option works as documented, it is suggested that you use mex –setup or mbuild –setup to switch compilers.

-g (Debugging Information)

Cause mex or 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 MEX-file or stand-alone application with a debugger.

The –g option flag has no influence on the source code that the MATLAB Compiler generates, though it does have some influence on the binary code that the C/C++ compiler generates.

-h (Helper Functions)

Compile helper functions by default. Any helper functions that are called will be compiled into the resulting MEX or stand-alone application.

Using the –h option is equivalent to listing the M-files explicitly on the mcc command line.

The –h option purposely does not include built-in functions or functions that appear in the MATLAB M-File Math Library portion of the C/C++ Math Libraries. This prevents compiling functions that are already part of the C/C++ Math Libraries. If you want to compile these functions as helper functions, you should specify them explicitly on the command line. For example, use

```
mcc –V1.2 minimize_it fmins
```

instead of

```
mcc –V1.2 –h minimize_it
```

Note Due to MATLAB Compiler 1.2 restrictions, some of the MATLAB 5 versions of the M-files for the C and C++ Math libraries do not compile as is. The MathWorks has rewritten these M-files to conform to the Compiler 1.2 restrictions. The modified versions of these M-files are in

```
<matlab>/extern/src/tbxsrc
```

where <matlab> represents the top-level directory where MATLAB is installed on your system.

-i (Inbounds Code)

Generate C/C++ code that does not:

- Check array subscripts to determine if array indexes are within range.
- Reallocate the size of arrays when the code requests a larger array. For example, if you preallocate a 10-element vector, the generated code cannot assign a value to the 11th element of the vector.
- Check input arguments to determine if they are real or complex.

The –i option flag can make a program run significantly faster, but not every M-file is a good candidate for –i. For instance, you can specify –i only if your M-file preallocates all arrays. You typically preallocate arrays with the `zeros` or `ones` function.

If an M-file contains code that causes an array to grow, then you cannot compile with the –i option. Using –i on such an M-file produces a MEX-file that fails at runtime.

Note This option flag has no effect on C++ generated code, i.e., if the –p MATLAB Compiler option flag is used.

-l (Line Numbers)

Generate C/C++ code that prints line numbers on internally detected errors. This option flag is useful for debugging, but causes the MEX-file to run slightly slower.

Note This option flag has no effect on C++ generated code, i.e., if the –p MATLAB Compiler option flag is used.

-m (main Routine)

Generate a C function named `main`. The name the MATLAB Compiler gives to your C function depends on the combination of option flags.

Option Flags	Resulting Function Name
neither –m nor –e	`mexFunction`
–m only	`main`
–e only	`mlf`*`Mfile1`*
–m and –e	`main`

If you specify –m and place multiple M-files on the compilation command line:

- The MATLAB Compiler applies the –m option to the first M-file only.
- The MATLAB Compiler assumes the –e option for all subsequent M-files.

The generated `main` function reads and writes from the standard input and standard output streams. POSIX-compliant operating systems include UNIX and Microsoft Windows. Other operating systems may require window-system specific changes for this code to work.

If your `main` M-function includes input or output arguments, the MATLAB Compiler will instantiate the input arguments using the command line arguments passed in by the POSIX shell. It will return the first output value produced by the M-file as the status. In this way, you can compile command line M-files into POSIX-compliant command line applications. For example,

```
function   sts = echo(a, b, c)
           display(a);
           display(b);
           display(c);
           sts = 0;
```

This function echoes its three input arguments. It will function as an M-file exactly the same way as it functions as a stand-alone application generated using the –m option. Note that only strings are passed as input variables from the POSIX shell and only a single scalar integer status is returned. Therefore, the –m option does not work well for mathematical M-files.

-M "string" (Direct Pass Through)

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

-p (Stand-Alone External C++ Code)

Generate C++ code for stand-alone applications. The resulting C++ code cannot be used to create MEX-files; it can be linked with the MATLAB C++ Math Library and executed as a stand-alone application. Stand-alone applications do not require MATLAB at runtime. Stand-alone applications can run even if MATLAB is not installed on the system. See Chapter 4 for complete details on stand-alone applications.

-q (Quick Mode)

Quick mode executes only one pass through the type imputation and assumes that complex numbers are contained in the inputs. Use Quick mode if at least one of the parameters to the functions being compiled is complex.

-r (Real)

Generate C code based on the assumption that all input, output, and temporary data in the MEX-file are real (not complex).

The `–r` option flag can make a program run significantly faster. However, if your program contains any complex data, do not compile with `–r`.

Placing the `%#realonly` pragma anywhere in the input M-file has the same effect as compiling with `–r`.

If you compile with `–r` but specify complex input data at runtime, the MEX-file issues a fatal error. However, if you compile with both `–r` and `–i`, the MEX-file does not check to see if the input data is complex. If the input data is complex, the MEX-file will fail at runtime.

Note This option flag has no effect on C++ generated code, i.e., if the `–p` MATLAB Compiler option flag is used.

-s (Static)

Translate MATLAB `global` variables to `static` C (local) variables. Compiling without –s causes the MATLAB Compiler to generate code that preserves the global status of any variables marked as `global` in an M-file.

Suppose that you tag `n` as a `global` variable in the MATLAB interpreter workspace:

```
global n
n = 3;
```

Consider an M-file that accesses `global` variable `n`:

```
function m = earth
global n;
m = magic(n);
```

Compiling `earth.m` without –s yields a MEX-file that can access the value of global variable `n`:

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

Compiling `earth.m` with –s yields a MEX-file that cannot access the value of global variable `n`:

```
mcc –s earth
earth
??? Error using ==> magic
Size vector must be a row vector with integer elements.
```

MEX-files access global variables very slowly because MEX-files have to call back to the MATLAB interpreter to obtain the value of a global variable.

Some programmers tag variables as `global` solely to recall a value from one invocation of the MEX-file to the next. If you are using `global` for this reason, then you should consider specifying the –s option flag when you compile. Doing so makes your MEX-file run faster. Compiling with –s causes the MEX-file to store the value of the variable locally; no callback is needed to access the

variable's value. The disadvantage to –s is that global variables are no longer really global; other programs cannot access them.

Note This option flag has no effect on C++ generated code, i.e., if the –p MATLAB Compiler option flag is used.

-S (Simulink S-Function)

Output a Simulink S-function with a dynamically sized number of inputs and outputs. You can pass any number of inputs and outputs in or out of the generated S-function. Since the MATLAB Fcn block and the S-Function block are single-input, single-output blocks, only one line can be connected to the input or output of these blocks. However, each line may be a vector signal, essentially giving these blocks multi-input, multi-output capability.

Note The MATLAB Compiler option that generates a C language S-function is a capital S (–S). Do not confuse it with the lowercase –s option that translates MATLAB global variables to static C (local) variables.

-t (Tracing Statements)

Generate tracing print statements. This option flag is useful for debugging, though it tends to generate a significant amount of information.

Note This option flag has no effect on C++ generated code, i.e., if the –p MATLAB Compiler option flag is used.

-u (Number of Inputs)

Provide more control over the number of valid inputs for your Simulink S-function. This option specifically sets the number of inputs for your function. If –u is omitted, the input will be dynamically sized.

-v (Verbose)

Display the steps in compilation, including:

- The command that is invoked
- The Compiler version number
- The invocation of `mex`/`mbuild`

The –v flag passes the –v flag to `mex` or `mbuild` and displays information about `mex` or `mbuild`. This option passes the full pathname of the file being compiled.

-w (Warning) and -ww (Complete Warnings)

Display warning messages indicating where sections of generated code are likely to slow execution (for example, where the code contains callbacks to MATLAB). This option flag does not affect the performance of the generated code.

If you omit –`w`, the MATLAB Compiler suppresses most warning messages, but still displays serious warning messages.

If you specify –`w`, the MATLAB Compiler displays up to 30 warning messages. If there are more than 30 warning messages, the MATLAB Compiler suppresses those past the 30th. To see *all* warning messages, use the –`ww` option.

-y (Number of Outputs)

Provide more control over the number of valid outputs for your Simulink S-function. This option specifically sets the number of outputs (`y`) for your function. If –`y` is omitted, the output will be dynamically sized.

-z <path> (Specifying Library Paths)

Specify the path to use for library and include files. This option uses the specified path for compiler libraries instead of the path returned by `matlabroot`.

Examples

Compile `gazelle.m` to create `gazelle.mex`:

```
mcc –V1.2 gazelle
```

Optimize the performance of `gazelle.mex` by compiling with two optimization option flags:

```
mcc –V1.2 –ri gazelle
```

Compile two M-files (`gazelle.m` and `cheetah.m`) to create one MEX-file (`gazelle.mex`):

```
mcc –V1.2 gazelle cheetah
```

Compile `gazelle.m` to create `gazelle.c` (C source code for a stand-alone application):

```
mcc –V1.2 –e gazelle
```

Given `leopard.m`, an M-file containing a call to `feval`:

```
function leopard(fun1,x)
y = feval(fun1,x);
plot(x,y,'r+');
```

Compile `leopard.m`, telling the MATLAB Compiler that function `fun1` corresponds to `myfun`:

```
mcc –V1.2 leopard fun1=myfun
```

Compile `myfun.m` to create a real, external version that includes a direct call to `auxfun1`, and replaces `feval(afun, . . .)` by `feval(auxfun2, . . .)`:

```
mcc –V1.2 –ire myfun auxfun1 afun=auxfun2
```

Make a quick-compiled version of `myfun.m` and compile all of the `helper.m` functions into a single object:

```
mcc –V1.2 –q –h myfun
```

See Also

`mbint`, `mbreal`, `mbscalar`, `mbvector`, `reallog`, `realpow`, `realsqrt`

A

MATLAB Compiler Quick Reference

Common Uses of the Compiler

This section summarizes how to use the MATLAB Compiler to generate some of its more standard results. The first four examples take advantage of the macro options.

Create a MEX-File

To translate an M-file named `mymfile.m` into C and to create the corresponding C MEX-file that can be called directly from MATLAB, use:

```
mcc –x mymfile
```

Create a Simulink S-Function

To translate an M-file named `mymfile.m` into C and to create the corresponding Simulink S-function using dynamically sized inputs and outputs, use:

```
mcc –S mymfile
```

Create a Stand-Alone C Application

To translate an M-file named `mymfile.m` into C and to create a stand-alone executable that can be run without MATLAB, use:

```
mcc –m mymfile
```

Create a Stand-Alone C++ Application

To translate an M-file named `mymfile.m` into C++ and to create a stand-alone executable that can be run without MATLAB, use:

```
mcc –p mymfile
```

Create a C Shared Library

To create a C shared library that performs specialized calculations that you can call from your own programs, use:

```
mcc –W lib:mylib –L C –t –T link:lib –h Function1 Function2 …
```

mcc (Compiler 2.0)

Bold entries in the Comment/Options column indicate default values.

Option	Description	Comment/Options

`A annotation:`*`type`*	Controls M-file code/ comment inclusion in generated C/C++ source	*`type`* = **`all`**
		`comments`
		`none`

`A debugline:`*`setting`*	Controls the inclusion of source filename and line numbers in run-time error messages	*`setting`* = `on` **`off`**
`A line:`*`setting`*	Controls the `#line` preprocessor directives included in the generated C/C++ source	*`setting`* = `on` **`off`**
`B` *`filename`*	Replaces `–B` *`filename`* on the `mcc` command line with the contents of *`filename`*	The file should contain only `mcc` command line options.
`c`	Generates C code only	Overrides `–T` option; equivalent to `–T codegen`
`d` *`directory`*	Places output in specified *`directory`*	
`f` *`filename`*	Uses the specified options file, *`filename`*	`mex –setup` and `mbuild –setup` are recommended.
`F` *`option`*	Specifies format parameters	*`option`* = `list` `expression-indent:`*`n`* `page-width:`*`n`* `statement-indent:`*`n`*
`g`	Generates debugging information	

Option	Description	Comment/Options
`h`	Compiles helper functions	
`I` *`directory`*	Adds new *`directory`* to path	
`l`	Generates code that reports file and line numbers on run-time errors	Equivalent to: `–A debugline:on`
`L` *`language`*	Specifies output target language	*`language`* = **`C`** `Cpp`
`m`	Macro to generate a C stand-alone application	Equivalent to: `–t –W main –L C –T link:exe –h`
`M "`*`string`*`"`	Passes *`string`* to `mex` or `mbuild`	Use to define compile-time options.
`o` *`outputfile`*	Specifies name/location of final executable	
`p`	Macro to generate a C++ stand-alone application	Equivalent to: `–t –W main –L Cpp –T link:exe –h`
`S`	Macro to generate Simulink S-function	Equivalent to: `–t –W simulink –L C –T link:mex`
`t`	Translates M code to C/C++ code	

`T` *`target`*	Specifies output stage	*`target`* = **`codegen`**
		`compile:`*`bin`*
		`link:`*`bin`*
		where *`bin`* = `mex`
		`exe`
		`lib`

`u` *`number`*	Specifies number of inputs for Simulink S-function	
`v`	Verbose; Displays compilation steps	

Option	Description	Comment/Options
`V1.2`	Invokes MATLAB Compiler 1.2	Not supported in stand-alone mode
`V2.0`	Invokes MATLAB Compiler 2.0	

`w` *`option`*	Displays warning messages	*`option`* = `list`
		`level`
		`level``:`*`string`*
		where *`level`* = `disable`
		`enable`
		`error`
		No `w` option displays only serious warnings (default).

`W` *`type`*	Controls the generation of function wrappers	*`type`* = `mex`
		`main`
		`simulink`
		`lib:`*`string`*
		`none`

`x`	Macro to generate MEX-function	Equivalent to: `–t –W mex –L C –T link:mex`
`y` *`number`*	Specifies number of outputs for Simulink S-function	
`Y` *`licensefile`*	Uses *`licensefile`* when checking out a Compiler license	
`z` *`path`*	Specifies *`path`* for library and include files	
`?`	Displays help message	

mcc (Compiler 1.2)

To use the MATLAB Compiler 1.2, you must use the –V1.2 option on the mcc command line.

Option	Description	Comment/Options
B *filename*	Replaces –B *filename* on the mcc command line with the contents of *filename*	The file should contain only mcc command line options.
c	Generates C code only	
e	Generates C code for stand-alone application	
f *filename*	Uses the specified options file, *filename*	mex –setup and mbuild –setup are recommended.
g	Generates debugging information	
h	Compiles helper functions	
i	(Optimization) Assumes subscripts are within bounds	Only valid when generating C code.
l	Generates code that reports file and line numbers on run-time errors	
m	Generates a main C function	
M "*string*"	Passes *string* to mex or mbuild	Defines compile-time options.
o *outputfile*	Specifies name/location of final executable	
p	Generates C++ code for stand-alone application	
q	Quick mode; Executes one pass through type imputation and assumes inputs contain complex numbers	

Option	Description	Comment/Options
`r`	(Optimization) Assumes input, output, and temporary data are real	Only valid when generating C code.
`s`	Translates MATLAB global variables to static C local variables	
`S`	Generates Simulink S-function	
`t`	Generates tracing print statements (C mode only) in code.	
`u` *number*	Specifies number of inputs for Simulink S-function	
`v`	Verbose; Displays compilation steps	
`V2.0`	Invokes MATLAB Compiler 2.0	
`w`	Displays warning messages (up to 30)	
`ww`	Displays complete list of all warning messages	
`y` *number*	Specifies number of outputs for Simulink S-function	
`z` *path*	Specifies *path* for library and include files	
`?`	Displays help message	

B

Error and Warning Messages

Introduction

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

Note 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` or 508 647-7000.

Compile-Time Messages

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: Could not check out a Compiler license.

No additional Compiler licenses are available for your workgroup.

Error: Could not find license file.

(*Windows only*) The `license.dat` file could not be found in `<MATLAB>\bin`.

Error: Could not identify file to compile.

No M, C, or C++ files were specified on the `mcc` command line.

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" in Chapter 3.

Error: File: *filename* Line: # Column: # "*string1*" expected, "*string2*" found.

There is a syntax error in the specified line. See the online MATLAB Function Reference pages accessible from the Help Desk.

Error: File: *filename* Line: # Column: # () indexing must appear last in an index expression.

If you use cell array indexing, `{}`, or the structure field access operator, `.`, to index into an expression, it must be last in the index expression. For example, you can use `X(1).value` and `X{2}(1)`, but you cannot use `X.value(1)` or `X(1){2}`.

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: File: *filename* Line: # Column: # An array for multiple LHS assignment must be a vector.

If the left-hand side of a statement is a multiple assignment, the list of left-hand side variables must be a vector. For example,

```
[p1, p2, p3] = myfunc(a)     % is correct
[p1; p2; p3] = myfunc(a)     % is incorrect
```

Error: File: *filename* Line: # Column: # An array for multiple LHS assignment cannot be empty.

If the left-hand side of a statement is a multiple assignment, the list of left-hand side variables cannot be empty. For example,

```
[p1, p2, p3] = myfunc(a)     % is correct
[ ] = myfunc(a)              % is incorrect
```

Error: File: *filename* Line: # Column: # An array for multiple LHS assignment cannot contain *token*.

If the left-hand side of a statement is a multiple assignment, the vector cannot contain this token. For example, you cannot assign to constants.

```
[p1] = myfunc(a)             % is correct
[3] = myfunc(a)              % is incorrect
```

Error: File: *filename* Line: # Column: # Expected Literal, found "*string*".

There is a syntax error in the specified line. See the online MATLAB Function Reference pages accessible from the Help Desk.

Error: File: *filename* Line: # Column: # Expected one of , ; % or EOL, got "*string*".

There is a syntax error in the specified line. See the online MATLAB Function Reference pages accessible from the Help Desk.

Error: File: *filename* Line: # Column: # Functions cannot be indexed using {} or . indexing.

You cannot use the cell array constructor, `{}`, or the structure field access operator, `.`, to index into a function.

Error: File: *filename* Line: # Column: # Invalid multiple left-hand-side assignment.

For example, you try to assign to constants

```
[] = sin(1);                % is incorrect
```

Error: File: *filename* Line: # Column: # MATLAB assignment cannot be nested.

You cannot use a syntax such as `x = y = 2`. Use `y= 2, x = y` instead.

Error: File: *filename* Line: # Column: # Only functions can return multiple values.

In this example, `foo` must be a function, it cannot be a variable:

```
[a, b] = foo;
```

Error: File: *filename* Line: # Column: # The end operator can only be used within an array index expression.

You can use the `end` operator in an array index expression such as `sum(A(:, end))`; you cannot use the `end` operator outside of such an expression, for example, `y = 1 + end`.

Error: File: *filename* Line: # Column: # The name *parametername* occurs twice as an input parameter.

The variable names specified on the function declaration line must be unique. For example,

```
function foo(bar1, bar2)     % is correct
function foo(bar, bar)       % is incorrect
```

Error: File: *filename* Line: # Column: # The name *parametername* occurs twice as an output parameter.

The variable names specified on the function declaration line must be unique. For example,

```
function [bar1, bar2] = foo  % is correct
function [bar, bar] = foo    % is incorrect
```

Error: File: *filename* Line: # Column: # The single colon operator (:) can only be used within an array index expression.

You can only use the `:` operator by itself as an array index. For example, `A(:) = 5;` is okay, but `y = :;` is not.

Error: File: *filename* Line: # Column: # Variable argument (varargin) must be last in input argument list.

The function call must specify the required arguments first followed by `varargin`. For example,

```
function [out1, out2] = example1(a, b, varargin)   % is correct
function [out1, out2] = example1(a, varargin, b)   % is incorrect
```

Error: File: *filename* Line: # Column: # Variable argument (varargout) must be last in output argument list.

The function call must specify the required arguments first followed by `varargout`. For example,

```
function [i, j, varargout]= ex2(x1, y1, x2, y2, val)% is correct
function [i, varargout, j]= ex2(x1, y1, x2, y2, val)% is incorrect
```

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('–A', 'none')            % is correct
mcc('–A none')               % is 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 the section, "MATLAB Compiler 2.0 Option Flags," in Chapter 6 or type `mcc –?` at the command prompt.

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 argument.

You must use an argument that corresponds to the option. For example,

```
mcc –L Cpp …                % is correct
mcc –L COBOL …              % is 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 *optionname* option must contain a colon.

The format for this argument requires a colon. For more information, see "MATLAB Compiler 2.0 Option Flags," in Chapter 6 or type `mcc –?` at the command prompt.

Error: The environment variable *variablename* is undefined.

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 license.dat file was not found in the directory in which mcc.exe resides.

(*Windows only*) The Compiler requires that your Compiler license file (`license.dat`) be in the same directory as `mcc.exe`, i.e., `<MATLAB>\bin`.

Error: The option –*optionname* is invalid in *modename* mode (specify –? for help).

The specified option is not available in the specified mode (`V1.2` or `V2.0`). You can use the –`V1.2` or –`V2.0` options to switch modes.

Error: The option –*optionname* must be immediately followed by whitespace (e.g. "*proper_example_usage*").

These options require additional information, so they cannot be combined: –`A`, –`B`, –`d`, –`f`, –`F`, –`I`, –`L`, –`M`, –`o`, –`T`, –`u`, –`W`, –`x`, –`y`, –`Y`, –`z`. For example, you can use `mcc –vc`, but you cannot use `mcc –Ac annotation:all`.

Error: The options specified will not generate any output files. Please use one of the following options to generate an executable output file:

–x (generates a MEX-file executable using C)
–m (generates a stand-alone executable using C)
–p (generates a stand-alone executable using C++)
–S (generates a Simulink MEX S-function using C)
Or type mcc –? for more usage information

Use one of these options or another option that generates output file(s). See the section, "MATLAB Compiler 2.0 Option Flags," in Chapter 6 or type `mcc –?` at the command prompt for more information.

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 stand-alone MCC Compiler does not support translating M-code to C/C++ code in V1.2 mode.

To use the stand-alone Compiler, you must use `V2.0` mode. Compiler 1.2 (`V1.2` mode) is only available from the MATLAB command prompt.

Error: The –*optionname* option cannot be combined with other options.

The –V1.2 and –V2.0 options must appear separate from other options on the command line. For example,

```
mcc –V2.0 –L Cpp …          % is correct
mcc –V2.0L Cpp …            % is incorrect
```

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 the section, "MATLAB Compiler 2.0 Option Flags," in Chapter 6 or type `mcc –?` at the command prompt.

Error: This version of MCC does not support the creation of C++ MEX code.

You cannot create C++ MEX functions with the current Compiler.

Error: Unable to open file *filename*.

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 annotation option: *optionname*.

An invalid string was specified after the –A option. For a complete list of the valid annotation options, see "MATLAB Compiler 2.0 Option Flags," in Chapter 6 or type `mcc –?` at the command prompt.

Error: Unknown typesetting option: *optionname*.

The valid typesetting options available with –F are `expression-indent:n`, `list`, `page-width`, and `statement-indent:n`.

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" section of this Appendix.

Error: Unrecognized option: *–optionname*.

The option is not one of the valid options for this version of the Compiler. See the section, "MATLAB Compiler 2.0 Option Flags," in Chapter 6 for a complete list of valid options for Compiler 2.0 or type `mcc –?` at the command prompt.

Error: Use "–V1.2" or "–V2.0" to specify desired version.

You specified `–V` without a version number. You must use either `–V1.2` or `–V2.0`.

Error: *versionnumber* is not a valid version number. Use "–V1.2" or "–V2.0".

If you specify a Compiler version number, it must be either `–V1.2` or `–V2.0`. The default is `–V2.0`.

Warning Messages

This section lists the warning messages that the MATLAB Compiler 2.0 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
```

For additional information about the –w option, see "MATLAB Compiler 2.0 Option Flags" in Chapter 6.

Warning: (PM) Warning: *message*.

(path_manager_warning) The path manager can experience file I/O problems when reading the directory structure (permissions).

Warning: (PMI): *message*.

(path_manager_inform) This is an informational path manager message.

Warning: A line has *number* characters, violating the maximum page width *width*.

(max_page_width_violation) To increase the maximum page width, use the –F page-width:n option and set n to a larger value.

Warning: File: *filename* Line: # Column: # A BREAK statement appeared outside of a loop. This BREAK is interpreted as a RETURN.

(break_without_loop) The break statement should be used in conjunction with the for or while statements. When not used in conjunction with these statements, the break statement acts as a return from a function.

Warning: File: *filename* Line: # Column: # Attempt to call an unknown function *functionname*. A run-time error will occur if this code is executed.

(undefined_function) The called function was not found on the search path.

Warning: File: *filename* Line: # Column: # Attempt to clear *value* when it has not been previously defined.

(clear_undefined_value) The variable was cleared with the `clear` command prior to being defined.

Warning: File: *filename* Line: # Column: # Empty clear. This statement will cause a run-time error if executed.

(clear_empty) In compiled M-files, you must specifically list variable(s) to clear when using the `clear` command.

Warning: File: *filename* Line: # Column: # References to *functionname* require the C/C++ Graphics Library when executing in stand-alone mode. A run-time error will occur if the C/C++ Graphics Library is not present.

(using_graphics_function) This warning is produced when a Graphics Library call is present in the code. It is only generated when producing the main or lib wrapper and not during normal compilation, unless it is specifically enabled.

Warning: File: *filename* Line: # Column: # References to *variablename* will produce a run-time error because it is an undefined function or variable.

(undefined_variable_or_unknown_function) This warning appears if you refer to a variable but never provide it with a value. The most likely cause of this warning is when you call a function that is not on the path or it is a method function. **Note** INLINE objects are not supported in this release and will produce this warning when used.

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 is used so that the `–h` option will automatically compile the selected functions.

Warning: File: *filename* Line: # Column: # The call to function *functionname* on this line passed *quantity1* inputs and the function is declared with *quantity2*. A run-time error will occur if this code is executed.

(too_many_inputs) There is an inconsistency between the number of formal and actual inputs to the function.

Warning: File: *filename* Line: # Column: # The call to function *functionname* on this line requested *quantity1* outputs and the function is declared with *quantity2*. A run-time error will occur if this code is executed.

(too_many_outputs) There is an inconsistency between the number of formal and actual outputs for the function.

Warning: File: *filename* Line: # Column: # The clear function cannot process the *optionname* option in compiled code.

(clear_cannot_handle_flag) You cannot use `clear variables`, `clear mex`, `clear functions`, or `clear all` in compiled M-code.

Warning: File: *filename* Line: # Column: # The clear statement did not specifically list the names of variables to be cleared as constant strings. A run-time error will be reported if this code is executed.

(clear_non_constant_strings) Use one of the forms of the `clear` command that contains the names of the variables to be cleared. Use `clear name` or `clear('name')`; do not use `clear(name)`.

Warning: File: *filename* Line: # Column: # The Compiler does not support the *optionname* option to save. This option is ignored.

(save_option_ignored) You cannot use `–ascii`, `–double`, or `–tabs` with the `save` command in compiled M-code.

Warning: File: *filename* Line: # Column: # The function *functionname* mentioned in this #function pragma was not found.

(pragma_function_name_not_found) The specified function provided in the list of function names will be called through an `feval` call. If the Compiler cannot locate the function, the Compiler will produce a run-time error, hence the warning.

Warning: File: *filename* Line: # Column: # The *functionname* function is only available in MEX mode. A run-time error will occur if this code is executed in stand-alone mode.

(using_mex_only_function) This warning is produced if you call any built-in function that is only available in `mex` mode. It is only generated when producing the main or lib wrapper and not during normal compilation, unless specifically enabled.

Warning: File: *filename* Line: # Column: # The load statement cannot be translated unless it specifically lists the names of variables to be loaded as constant strings.

(load_without_constant_strings) Use one of the forms of the `load` command that contains the names of the variables to be loaded, for example, `load filename num` or `y = load('filename')`

Warning: File: *filename* Line: # Column: # The save statement cannot be translated unless it specifically lists the names of variables to be saved as constant strings.

(save_without_constant_strings) Use one of the forms of the `save` command that contains the names of the variables to be saved, for example, `save filename num`

Warning: File: *filename* Line: # Column: # This load statement did not provide a list of names and will not be translated. Rewrite this load to specifically mention the variables being loaded.

(load_without_names) You must specify the names of the variables to load, such as `load x.mat num1` as opposed to `load x.mat`, which loads all variables in `x.mat`.

Warning: File: *filename* Line: # Column: # This save statement did not provide a list of names and will not be translated. Rewrite this save to specifically mention the variables being saved.

(save_without_names) You must specify the names of the variables to save, such as `save x.mat num1` as opposed to `save x.mat`, which saves all variables in `x.mat`.

Warning: File: *filename* Line: # Column: # Unmatched "end".

(end_without_block) The `end` statement does not have a corresponding `for`, `while`, `switch`, `try`, or `if` statement.

Warning: File: *filename* Line: # Column: # Unrecognized Compiler pragma *pragmaname*.

(unrecognized_pragma) Use one of the Compiler pragmas as described in Chapter 6, “Reference”.

Warning: File: *filename* Line: # Column: # *name* has been used as both a function and a variable, the variable is ignored.

(inconsistent_variable) When a name represents both a function and a variable, it is used as the function only.

Warning: File: *filename* Line: # Column: # *variablename* has not been defined prior to use on this line.

(undefined_variable) Variables should be defined prior to use.

Warning: File:*filename* Line: # Column: # The Compiler does not support EVAL or INPUT functions. Code will still be generated, but calls to these functions are replaced with a run-time error.

(eval_not_supported) Currently, these are unsupported functions.

Warning: Ignoring parameter with '=' in it: *parametername*. Specify –V1.2 to use this command line syntax.

(*equals_ignored*) In Compiler 2.0, you cannot pass the names of input functions at compile time using the `feval` function.

Warning: Line: # Column: # Duplicate function with name *functionname1* has been renamed to *functionname2* and no calls will be generated to it from M code.

(duplicate_function_name) This warning occurs when an M-file contains more than one function with the same name.

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 –x –I /dir1 /dir2/afile.m
```

Warning: No M-function source available for *functionname*, assuming function [varargout] = *functionname*(varargin).

(using_stub_function) The Compiler found a `.p` or `.mex` version of the function and is substituting a generic function declaration in its place.

Warning: The function *functionname* is an intrinsic MATLAB function. The signature of the function found in file "*filename*" does not match the known signature for this function:

known number of inputs = *quant1*, found number of inputs = *quant2*
known number of outputs = *quant1* found number of outputs = *quant2*
known varargin used = *quant1*, found varargin used = *quant2*
known varargout used = *quant1*, found varargout used = *quant2*
known nargout used = *quant1*, found nargout used = *quant2*

(*builtin_signature_mismatch*) When compiling an M-file that is contained in The MathWorks libraries, the number of inputs/outputs and the signatures to the function must match exactly.

Warning: The option *optionname* is ignored in *modename* mode (specify –? for help)

(*switch_ignored*) *Modename* = 1.2 or 2.0. Certain options only have meaning in one or the other mode. For example, if you use the `–e` option, you can't use the `–V2.0` option. For more information about Compiler options, see the section, "MATLAB Compiler 2.0 Option Flags," in Chapter 6.

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. For more information, see "Compiling Private and Method Functions" in Chapter 5.

Run-Time Messages

Note The error messages described in this section are generated by the Compiler into the code exactly as they are written, but are not the only source of run-time errors. You also can receive run-time errors can from the C/C++ Math Libraries; these errors are not documented in this book. Math Library errors do not include the source file and line number information. If you receive such an error and are not certain if it is coming from the C/C++ Math Libraries or your M-code, compile with the `–A debugline:on` option to get additional information about which part of the M source code is causing the error. For more information about `–A` (the annotation option), see "Code Generation Options" in Chapter 6.

Run-time Error: File: *filename* Line: # Column: # Attempt to call an unknown function *functionname*.

The function was not found at compile time.

Run-time Error: File: *filename* Line: # Column: # Empty clear.

The source M-function called `clear` with no arguments.

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 clear statement did not specifically list the names of variables to be cleared as constant strings.

Use one of the forms of the `clear` command that contains the names of the variables to be cleared, for example,

```
clear name
```

Run-time Error: File: *filename* Line: # Column: # The Compiler does not support EVAL or INPUT functions.

Currently, these are unsupported functions.

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.

Run-time Error: File: *filename* Line: # Column: # The load statement did not specifically list the names of variables to be loaded as constant strings.

Use one of the forms of the `load` command that contains the names of the variables to be loaded, for example,

```
load filename num value
```

Run-time Error: File: *filename* Line: # Column: # The save statement did not specifically list the names of variables to be saved as constant strings.

Use one of the forms of the `save` command that contains the names of the variables to be saved, for example,

```
save testdata num value
```

Run-time Error: File: *filename* Line: # Column: # This load statement did not provide a list of names and was not translated.

Specify the names of the variables to load, such as `load x.mat num1` as opposed to `load x.mat`, which loads all variables in `x.mat`. You can also use the form `y = load('file')`.

Run-time Error: File: *filename* Line: # Column: # This save statement did not provide a list of names and was not translated.

Specify the names of the variables to save, such as `save x.mat num1` as opposed to `save x.mat`, which saves all variables in `x.mat`.

Run-time Error: File: *filename* Line: # Column: # *variablename* has not been defined prior to use on this line.

Variables should be defined prior to use.

C

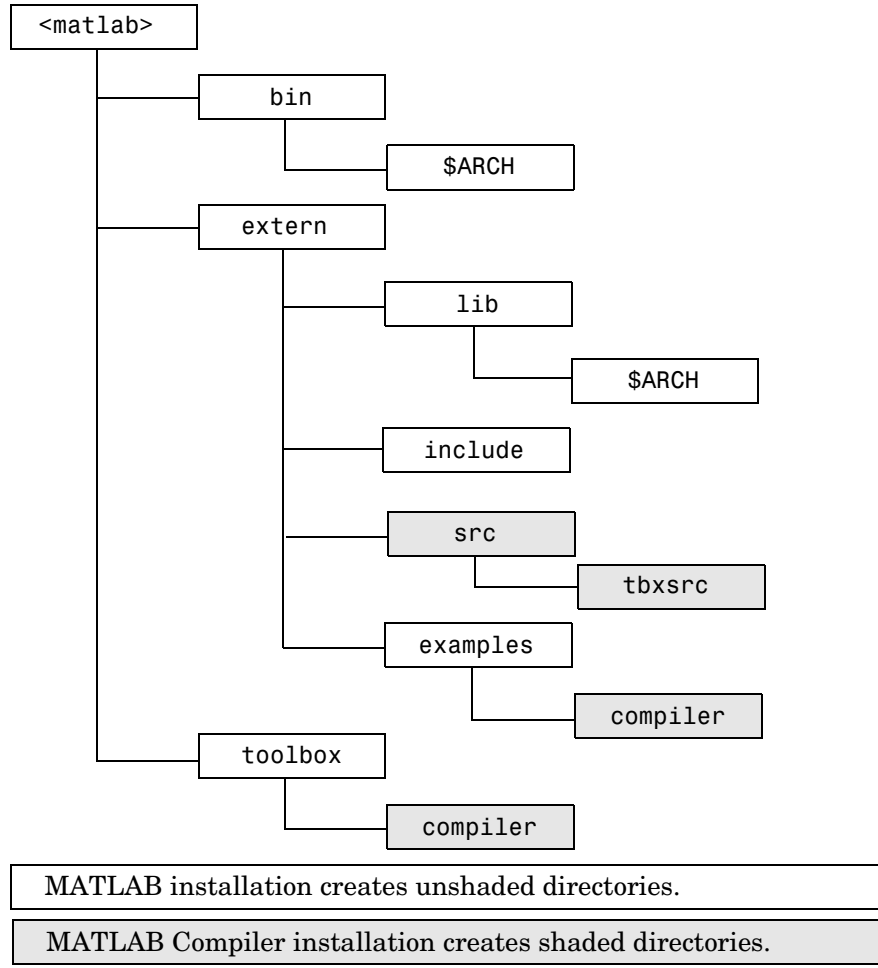
Directory Organization

This chapter describes the directory organization of the MATLAB Compiler on UNIX and Microsoft Windows systems.

Note that if you also install the MATLAB C/C++ Math Library, the directory organization is different from those shown in this chapter. See the chapters about directory organization in the *MATLAB C Math Library User's Guide* (for the C Math Library) or the *MATLAB C++ Math Library User's Guide* (for the C++ Math Library).

Directory Organization on UNIX

Installation of the MATLAB Compiler places many new files into directories already used by MATLAB. In addition, installing the MATLAB Compiler creates several new directories. This figure illustrates the directories in which the MATLAB Compiler files are located.



In the illustration, `<matlab>` symbolizes the top-level directory where MATLAB is installed on your system.

<matlab>

The <matlab> directory, in addition to containing many other directories, can contain one MATLAB Compiler document.

Compiler_Readme	Optional document that describes configuration details, known problems, workarounds, and other useful information.

<matlab>/bin

This table lists the files in the <matlab>/bin directory that are relevant to compiling.

matlab	UNIX shell script that initializes your environment and then invokes the MATLAB interpreter. MATLAB also provides a matlab script, but the MATLAB Compiler version overwrites the MATLAB version. The difference between the two versions is that the MATLAB Compiler version adds the appropriate directory (<matlab>/extern/lib/arch) to the shared library path.
mbuild	UNIX shell script that controls the building and linking of your code.
mex	UNIX shell script that creates MEX-files from C MEX-file source code. See the *Application Program Interface Guide* for more details on mex. MATLAB also installs mex; the MATLAB Compiler installation copies the existing version of mex to mex.old prior to installing the new version of mex.
gccopts.sh	Options file for building gcc MEX-files.
mbuildopts.sh	Shell script used by mbuild and mbuild.m.

`mcc`	UNIX shell script that initializes the environment and invokes MATLAB Compiler in the platform-specific `bin` directory.
`mexopts.sh`	Options file for building MEX-files using the system's native compiler.

<matlab>/bin/$ARCH

The `<matlab>/bin/$ARCH` directory, where `$ARCH` specifies a particular UNIX platform, contains the platform-specific Compiler executable and Compiler shared libraries.

`mcc`	Platform-specific MATLAB Compiler executable.

`libmwcompiler.so`
`libmwcompiler.sl`
`libmwcompiler.a`
	Compiler shared library that contains shared code between the DOS/UNIX Compiler and the MEX-file based Compiler.

`libmatlbmx.so`
`libmatlbmx.sl`
`libmatlbmx.a`
	Shared library that contains the Math Library API for use by MEX-files.

<matlab>/extern/lib/$ARCH

The `<matlab>/extern/lib/$ARCH` directory contains libraries.

This table lists the library for the MATLAB Compiler.

`libmccmx.so`
`libmccmx.sl`
`libmccmx.a`
	MATLAB Compiler Library for MEX-files. Contains the `mcc` and `mcm` routines required for building MEX-files (`–V1.2` mode only).

This table lists the libraries for the MATLAB C Math Library, a separate product.

`libmmfile.so`
`libmmfile.sl`
`libmmfile.a`
	MATLAB M-File Math Library. Contains callable versions of the M-files in the MATLAB Toolbox. Needed for building stand-alone applications that require MATLAB toolbox functions.

`libmcc.so`
`libmcc.sl`
`libmcc.a`
	MATLAB Compiler Library for stand-alone applications. Contains the `mcc` and `mcm` routines required for building stand-alone applications (`–V1.2` mode only).

`libmatlb.so`
`libmatlb.sl`
`libmatlb.a`
	MATLAB Math Built-In Library. Contains callable versions of MATLAB built-in math functions and operators. Required for building stand-alone applications.

This table lists the library for the MATLAB C++ Math Library, a separate product.

`libmatpp.so`
`libmatpp.sl`
`libmatpp.a`
	MATLAB C++ Math Library. Contains callable versions of MATLAB built-in math functions and operators. Required for building stand-alone C++ applications.

<matlab>/extern/include

The `<matlab>/extern/include` directory contains the header files for developing C applications that interface with MATLAB.

`matlab.h`	Header file for MATLAB Math Built-In Library and MATLAB M-File Math Library.
`libmatlb.h`	Header file for MATLAB Built-In Library.
`libmmfile.h`	Header file for M-File Math Library.

`libsgl.h`	Header file for C/C++ Graphics Library API.

This table lists the relevant header files from MATLAB.

`mat.h`	Header file for programs accessing MAT-files. Contains function prototypes for `mat` routines; installed with MATLAB.
`matrix.h`	Header file containing a definition of the `mxArray` type and function prototypes for matrix access routines; installed with MATLAB.
`mcc.h`	Header files used by Compiler in `–V1.2` mode.
`mccsimulink.h`	Contains library functions used to support the building of Simulink S-functions.
`mex.h`	Header file for building MEX-files. Contains function prototypes for `mex` routines; installed with MATLAB.

<matlab>/extern/include/cpp

This table lists the header file for the MATLAB C++ Math Library, a separate product.

`matlab.hpp`	Header file for MATLAB C++ Math Library.

Note There are numerous other files included by `matlab.hpp` in this directory.

<matlab>/extern/src/tbxsrc

The `<matlab>/extern/src/tbxsrc` directory contains the MATLAB Compiler-compatible M-files.

<matlab>/extern/examples/compiler

The `<matlab>/extern/examples/compiler` directory holds sample M-files, C functions, UNIX shell scripts, and makefiles described in this book. For some examples, the online version may differ from the version in this book. In those cases, assume that the online versions are correct.

`earth.m`	M-file that accesses a global variable (page 6-53).
`fibocert.m`	M-file that explores assertions (page D-19).
`fibocon.m`	M-file used to show MEX-file source code (page D-38).
`fibomult.m`	M-file that explores helper functions (page D-29).
`gasket.m`	M-file that determines the coordinates of a Sierpinski Gasket (page 3-3).
`hello.m`	M-file that displays `Hello, World`.
`houdini.m`	Script M-file that cubes each element of a 2-by-2 matrix (page 3-17).
`lu2.m`	M-file example that benefits from `%#ivdep` .
`DOT.mexrc.sh`	Example `.mexrc.sh` file that supports ANSI C compilers.
`Makefile`	Example `gmake`-compatible makefile for building stand-alone applications. (Only `gmake` can process this makefile.)
`README`	Short description of each file in this directory.
`main.m`	M-file “main program” that calls `mrank` (page 4-34).
`mrank.m`	M-file to calculate the rank of magic squares (page 4-33).

`mrankp.c`	POSIX-compliant C function "main program" that calls `mrank`. Demonstrates how to write C code that calls a function generated by the MATLAB Compiler. Input for this function comes from the standard input stream and output goes to the standard output stream (page 4-43).
`mrankwin.c`	Windows version of `mrankp.c`.
`multarg.m`	M-file that contains two input and two output arguments (page 4-44).
`multargp.c`	C function "main program" that calls `multarg`. Demonstrates how to write C code that calls a function generated by the MATLAB Compiler (page 4-44).
`mycb.m`	M-file example used to study callbacks (page D-29).
`mydep.m`	M-file example used to show a case when `%#ivdep` generates incorrect results.
`myfunc.m`	M-file that explores helper functions (page D-18).
`myph.c`	C function print handler initialization (page 5-72).
`mypoly.m`	M-file example used to study callbacks (page D-31).
`novector.m`	M-file example that demonstrates the influence of vectorization (page D-36).
`plot1.m`	M-file example that calls `feval` (page D-33).
`plotf.m`	M-file example that calls `feval` multiple times (page D-33).

squares1.m	M-file example that does not preallocate a matrix (page D-34).
squares2.m	M-file example that preallocates a matrix (page D-35).
squibo.m	M-file to calculate "squibonacci" numbers. Compile squibo.m into a MEX-file.
squibo2.m	M-file example that contains a %#realonly pragma.
tridi.m	M-file to solve a tridiagonal system of equations. Compile tridi.m into a MEX-file.
yovector.m	M-file example that demonstrates the influence of vectorization (page D-36).

<matlab>/toolbox/compiler

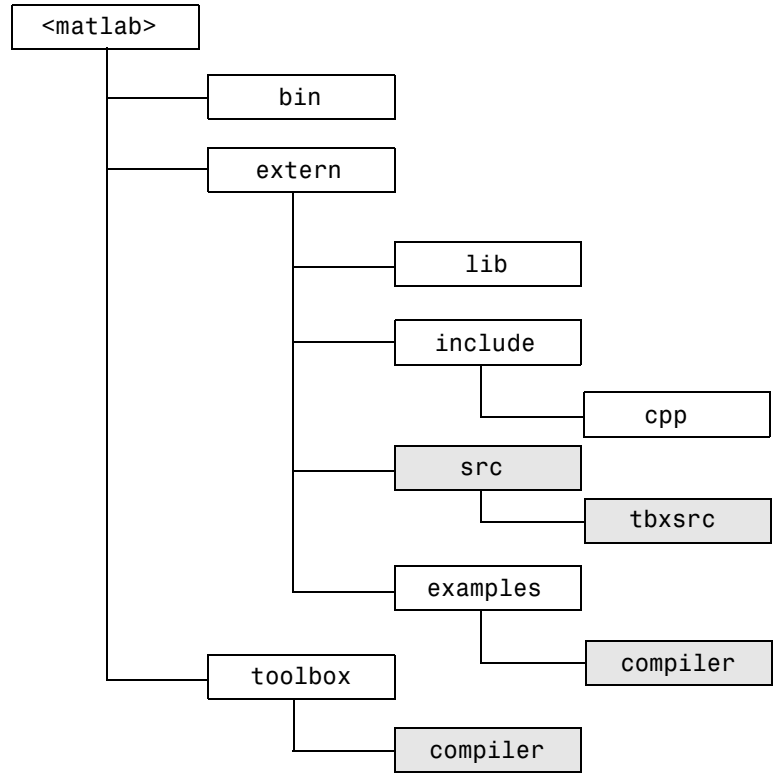
The <matlab>/toolbox/compiler directory contains the MATLAB Compiler's additions to the MATLAB command set.

Contents.m	List of M-files in this directory.
inbounds.m	Help file for the %#inbounds pragma.
ivdep.m	Help file for the %#ivdep pragma.
mbchar.m	Assert variable is a MATLAB character string.
mbcharscalar.m	Assert variable is a character scalar.
mbcharvector.m	Assert variable is a character vector, i.e., a MATLAB string.
mbint.m	Assert variable is integer.
mbintscalar.m	Assert variable is integer scalar.
mbintvector.m	Assert variable is integer vector.
mbreal.m	Assert variable is real.

`mbrealscalar.m`	Assert variable is real scalar.
`mbrealvector.m`	Assert variable is real vector.
`mbscalar.m`	Assert variable is scalar.
`mbuild.m`	Build stand-alone applications from MATLAB command prompt.
`mbvector.m`	Assert variable is vector.
`mcc.<mex>`	Invoke the MATLAB Compiler.
`mccexec.<mex>`	MATLAB Compiler internal routine.
`mccsavepath.m`	Duplicate MATLAB's search path for use with Compiler 2.0.
`reallog.m`	Natural logarithm for nonnegative real inputs.
`realonly.m`	Help file for the `%#realonly` pragma.
`realpow.m`	Array power function for real-only output.
`realsqrt.m`	Square root for nonnegative real inputs.

Directory Organization on Microsoft Windows

You must install MATLAB prior to installing the MATLAB Compiler. Installing the MATLAB Compiler places many new files into directories already created by the MATLAB installation. In addition, installing the MATLAB Compiler creates several new directories. This figure illustrates the Microsoft Windows directories into which the MATLAB Compiler installation places files.



In the illustration, `<matlab>` symbolizes the top-level folder where MATLAB is installed on your system.

<matlab>

The <matlab> directory, in addition to containing many other directories, can contain one MATLAB Compiler document.

Compiler_Readme	Optional document that describes configuration details, known problems, workarounds, and other useful information.

<matlab>\bin

The <matlab>\bin directory contains the files required to build stand-alone applications and MEX-files.

This table lists the library for the MATLAB Compiler.

compiler.dll	Kernel code for the MATLAB Compiler product shared between the MEX and stand-alone versions of the Compiler.
libmccmx.dll	MATLAB Compiler Library for MEX-files. Contains the mcc and mcm routines required for building MEX-files (–V1.2 mode only).
mcc.exe	Executable that compiles M code to C code and optionally invokes mex or mbuild.

This table lists the libraries for the MATLAB C Math Library, a separate product, and the Compiler options files.

libmatlbmx.dll	Shared library that contains the Math Library API for use by MEX-files.
libmmfile.dll	MATLAB M-File Math Library. Contains callable versions of the M-files in the MATLAB Toolbox. Needed for building stand-alone applications that require MATLAB toolbox functions.

`libmcc.dll`	MATLAB Compiler Library for stand-alone applications. Contains the `mcc` and `mcm` routines required for building stand-alone applications (`–V1.2` mode only).
`libmatlb.dll`	MATLAB Math Built-In Library. Contains callable versions of MATLAB built-in math functions and operators. Required for building stand-alone applications.
`mex.bat`	Batch file that builds C files into MEX-files. See the *Application Program Interface Guide* for more details on `MEX.BAT`.
`mbuild.bat`	Batch file that builds C files into stand-alone C or C++ applications with math libraries.
`mexopts.bat`	Default options file for use with `mex.bat`. Created by `mex –setup`.
`compopts.bat`	Default options file for use with `mbuild.bat`. Created by `mbuild –setup`.
Options files for `mex.bat`	Options and settings for C and C++ compilers to create MEX-files, e.g., `bccopts.bat` for use with Borland C++ and `watcopts.bat` for use with Watcom C/C++, Version 10.x.
Options files for `mbuild.bat`	Options and settings for C and C++ compilers to create stand-alone applications, e.g., `msvccompp.bat` for use with Microsoft Visual C and `bcc52compp.bat` for use with Borland C/C++.

All DLLs are in WIN32 format.

<matlab>\extern\lib

The MATLAB C++ Math Library, a separate product, has three separate, compiler-specific libraries. Each library contains callable versions of MATLAB

built-in math functions and operators. The MATLAB C++ Math Library is required for building stand-alone C++ applications.

`libmatpb50.lib`	MATLAB C++ Math Library for Borland Compiler V5.0.
`libmatpb52.lib`	MATLAB C++ Math Library for Borland Compiler V5.2.
`libmatpb53.lib`	MATLAB C++ Math Library for Borland Compiler V5.3.
`libmatpm.lib`	MATLAB C++ Math Library for Microsoft Visual C++ (MSVC) Compiler.
`libmatpw106.lib`	MATLAB C++ Math Library for Watcom Compiler V10.6.
`libmatpw11.lib`	MATLAB C++ Math Library for Watcom Compiler V11.

<matlab>\extern\include

The `<matlab>\extern\include` directory contains the header files that come with MATLAB-based, software development products.

This table lists the header file for the MATLAB Compiler 1.2.

`mcc.h`	Header file for MATLAB Compiler Library.

This table lists the header file for the MATLAB C Math Library, a separate product.

`matlab.h`	Header file for MATLAB Math Library.
`libmatlb.h`	Header file for MATLAB Built-In Library.
`libmmfile.h`	Header file for M-File Math Library.
`libsgl.h`	Header file for C/C++ Graphics Library API.

This table lists the relevant header files from MATLAB.

`mat.h`	Header file for programs accessing MAT-files. Contains function prototypes for `mat` routines; installed with MATLAB.
`matrix.h`	Header file containing a definition of the `mxArray` type and function prototypes for matrix access routines; installed with MATLAB.
`mex.h`	Header file for building MEX-files. Contains function prototypes for `mex` routines; installed with MATLAB.

The following `.def` files are used by the MSVC and Borland compilers. The `lib*.def` files are used by MSVC and the `_lib*.def` files are used by Borland.

`libmmfile.def` `_libmmfile.def`	Contains names of functions exported from the MATLAB M-File Math Library DLL.
`libmcc.def` `_libmcc.def`	Contains names of functions exported from the MATLAB Compiler Library for stand-alone applications.
`libmatlb.def` `_libmatlb.def`	Contains names of functions exported from the MATLAB Math Built-In Library.
`libmatlbmx.def` `_libmatlbmx.def`	Contains the Math Library API for use by MEX-files.
`libmccmx.def` `_libmccmx.def`	Contains names of functions exported from `libmccmx`.
`libmx.def` `_libmx.def`	Contains names of functions exported from `libmx.dll`.

<matlab>\extern\include\cpp

This table lists the header file for the MATLAB C++ Math Library, a separate product.

`matlab.hpp`	Header file for MATLAB C++ Math Library.

Note There are numerous other files included by `matlab.hpp` in this directory.

<matlab>\extern\src\tbxsrc

The `<matlab>\extern\src\tbxsrc` directory contains the MATLAB Compiler-compatible M-files.

<matlab>\extern\examples\compiler

The `<matlab>\extern\examples\compiler` directory holds sample M-files, C functions, and batch files described in earlier chapters of this book. For some examples, the online version may differ from the version in this book. In those cases, assume that the online versions are correct.

`earth.m`	M-file that accesses a global variable (page 6-53).
`fibocert.m`	M-file that explores assertions (page D-19).
`fibocon.m`	M-file used to show MEX-file source code (page D-38).
`fibomult.m`	M-file that explores helper functions (page D-29).
`gasket.m`	M-file that determines the coordinates of a Sierpinski Gasket (page 3-3).
`hello.m`	M-file that displays `Hello, World`.

houdini.m	Script M-file that cubes each element of a 2-by-2 matrix (page 3-17).
lu2.m	M-file example that benefits from %#ivdep.
README	Short description of each file in this directory.
main.m	M-file "main program" that calls mrank (page 4-34).
mrank.m	M-file to calculate the rank of magic squares (page 4-33).
mrankp.c	POSIX-compliant C function "main program" that calls mrank. Demonstrates how to write C code that calls a function generated by the MATLAB Compiler. Input for this function comes from the standard input stream and output goes to the standard output stream (page 4-43).
mrankwin.c	Windows version of mrankp.c.
multarg.m	M-file that contains two input and two output arguments (page 4-44).
multargp.c	C function "main program" that calls multarg. Demonstrates how to write C code that calls a function generated by the MATLAB Compiler (page 4-44).
mycb.m	M-file example used to study callbacks (page D-29).
mydep.m	M-file example used to show a case when %#ivdep generates incorrect results.
myfunc.m	M-file that explores helper functions (page D-18).
myph.c	C function print handler initialization (page 5-72).

`mypoly.m`	M-file example used to study callbacks (page D-31).
`novector.m`	M-file example that demonstrates the influence of vectorization (page D-36).
`plot1.m`	M-file example that calls `feval` (page D-33).
`plotf.m`	M-file example that calls `feval` multiple times (page D-33).
`squares1.m`	M-file example that does not preallocate a matrix (page D-34).
`squares2.m`	M-file example that preallocates a matrix (page D-35).
`squibo.m`	M-file to calculate “squibonacci” numbers. Compile `squibo.m` into a MEX-file.
`squibo2.m`	M-file example that contains a `%#realonly` pragma.
`tridi.m`	M-file to solve a tridiagonal system of equations. Compile `tridi.m` into a MEX-file.
`yovector.m`	M-file example that demonstrates the influence of vectorization (page D-36).

<matlab>\toolbox\compiler

The `<matlab>\toolbox\compiler` directory contains the MATLAB Compiler’s additions to the MATLAB command set.

`Contents.m`	List of M-files in this directory.
`inbounds.m`	Help file for the `%#inbounds` pragma.
`ivdep.m`	Help file for the `%#ivdep` pragma.
`mbchar.m`	Assert variable is a MATLAB character string.

mbcharscalar.m	Assert variable is a character scalar.
mbcharvector.m	Assert variable is a character vector, i.e., a MATLAB string.
mbint.m	Assert variable is integer.
mbintscalar.m	Assert variable is integer scalar.
mbintvector.m	Assert variable is integer vector.
mbreal.m	Assert variable is real.
mbrealscalar.m	Assert variable is real scalar.
mbrealvector.m	Assert variable is real vector.
mbscalar.m	Assert variable is scalar.
mbuild.m	Build stand-alone applications from MATLAB command prompt.
mbvector.m	Assert variable is vector.
mcc.dll	Invoke the MATLAB Compiler 2.0.
mccexec.dll	MATLAB Compiler 1.2 internal routine.
mccsavepath.m	Duplicate MATLAB's search path for use with Compiler 2.0.
reallog.m	Natural logarithm for nonnegative real inputs.
realonly.m	Help file for the `%#realonly` pragma.
realpow.m	Array power function for real-only output.
realsqrt.m	Square root for nonnegative real inputs.

D

Using Compiler 1.2

Introduction

This appendix provides information on Compiler 1.2. To use this Compiler, you must explicitly include the `–V1.2` option on the `mcc` command line.

Why Use Compiler 1.2?

The primary reason for using Compiler 1.2 is if you need code that is optimized for performance. If you don't require optimized code, then you should use Compiler 2.0.

About This Appendix

In addition to discussing the optimization techniques available with Compiler 1.2, this appendix contains information on the code generated by Compiler 1.2, and the limitations and restrictions for this version of the Compiler.

Note All references to the MATLAB Compiler and `mcc` in this Appendix refer to version 1.2 of the Compiler.

Limitations and Restrictions

This section summarizes restrictions on MATLAB code that can be used with the Compiler. It also describes differences between the MATLAB Compiler 1.2 and the MATLAB interpreter. Finally, this section discusses the restrictions that exist on stand-alone applications generated by the Compiler.

Type Imputation

Type imputation is the process that the MATLAB Compiler uses to analyze how variables in M-files are assigned values and how these values are used. This analysis helps produce more efficient C code from the M-files.

Optimization

The optimization sections of this appendix explain how to improve the performance of code generated by the MATLAB Compiler 1.2.

You can optimize performance for C code by:

- Supplying appropriate MATLAB Compiler options (page D-10).
- Avoiding callbacks to MATLAB (page D-27).
- Preallocating matrices in your M-file (page D-34).

You can optimize performance for C *and* C++ code by:

- Type imputation (page D-8).
- Specifying assertions (page D-17).
- Specifying pragmas (page D-21).
- Avoiding complex calculations (page D-26).
- Vectorizing your M-file (page D-35).

The *MATLAB C++ Math Library User's Guide* provides additional information about how to optimize C++ applications that use the MATLAB C++ Math Library.

Note In this release of MATLAB Compiler 2.0, optimization requires the `–V1.2` option, making MATLAB Compiler 1.2 the working version.

The Generated Code

This section describes the code generated by Compiler 1.2, in particular:

- The C code that the `mcc` command generates (MEX-files)
- The C code that the `mcc –V1.2 –m` command generates (stand-alone)
- The C++ code that `mcc –V1.2 –p` command generates (stand-alone)

Limitations and Restrictions

Version 1.2 of the MATLAB Compiler was a compatibility release that brought the Compiler into compliance with MATLAB 5. Although Compiler 1.2 works with MATLAB 5, it does not support many of the newer features of MATLAB 5. In addition, code generated with Compiler 1.2 is not guaranteed to be forward compatible.

The remainder of this section describes the limitations of Compiler 1.2.

MATLAB Compiler 1.2

MATLAB Code

There are some limitations and restrictions on the kinds of MATLAB code with which the MATLAB Compiler can work. The MATLAB Compiler Version 1.2 cannot compile:

- Script M-files.
- M-files containing `eval` or `input`. These functions create and use internal variables that only the MATLAB interpreter can handle.
- M-files that use the explicit variable `ans`.
- M-files that create or access sparse matrices.
- Built-in MATLAB functions (functions such as `eig` have no M-file, so they can't be compiled), however, calls to these functions are okay.
- Functions that are only MEX functions.
- Functions that use variable argument lists (`varargin`).
- M-files that use `feval` to call another function defined within the same file. (Note: In stand-alone C and C++ modes, a new pragma (`%#function <name-list>`) is used to inform the MATLAB Compiler that the specified function will be called through an `feval` call. See "Using feval" in Chapter 5 for more information.)

- Calls to `load` or `save` that do not specify the names of the variables to load or save. The `load` and `save` functions *are* supported in compiled code for lists of variables only. For example, this is acceptable:

```
load( filename, 'a', 'b', 'c' );      % This is OK and loads the
                                      % values of a, b, and c from
                                      % the file.
```

  However, this is *not* acceptable:

```
load( filename, var1, var2, var3 );   % This is not allowed.
```

  There is *no* support for the `load` and `save` options `–ascii`, `–mat`, `–v1.2`, and `–append`, and the variable wildcard `(*)`.
- M-files that use multidimensional arrays, cell arrays, structures, or objects.

Variable Names Ending with Underscores. The MATLAB Compiler generates a warning message for M-files that contain variables whose names end with an underscore (_) or an underscore followed by a single digit. For example, if your M-file contains the variables `result_` or `result_8`, the Compiler would generate a warning message because these names can conflict with the Compiler-generated names and may cause errors.

MATLAB Compiler-Compatible M-Files. Since Compiler 1.2 cannot handle return results from functions that are MATLAB 5 objects (cell arrays, structures, objects), handling of the ODE solvers in MEX mode is problematic. To properly handle `odeget()` and `odeset()`, a set of MATLAB Compiler-compatible M-files that produce the same results is included with the MATLAB Compiler. These M-files must be on the path in MEX mode because Compiler 1.2 will generate calls to them and the MATLAB 5 versions will return cell arrays causing a runtime error.

The directories containing the MATLAB Compiler-compatible M-files are:

- On UNIX, `<matlab>/extern/src/tbxsrc`
- On Windows, `<matlab>\extern\src\tbxsrc`

This stipulation also applies to the return structure of `polyval` used as input to `polyfit`. Calls to Handle Graphics functions will produce errors in stand-alone mode if you do not have the optional MATLAB C/C++ Graphics Library.

Differences Between the MATLAB Compiler 1.2 and Interpreter

In addition, there are several circumstances where the behavior of the MATLAB Compiler 1.2 is slightly different from the MATLAB interpreter's:

- The MATLAB interpreter permits complex operands to <, <=, >, and >= but ignores any imaginary components of the operands. The MATLAB Compiler does not permit complex operands to <, <=, >, and >=. In fact, the MATLAB Compiler assumes that any operand to these operators is real.
- The MATLAB Compiler assumes all arguments to the iterator operator (`:`) are scalars.
- The MATLAB Compiler forces all arguments to `zeros`, `ones`, `eye`, and `rand` to be integers. The MATLAB interpreter allows noninteger arguments to these functions but issues a warning indicating that noninteger arguments may not be supported in a future release.
- The MATLAB interpreter stores all numerical data as double-precision, floating-point values. By contrast, the MATLAB Compiler sometimes stores numerical data in integer data types. Integer results (the results of adding, subtracting, or multiplying integers) must fit into integer variables. A very large integer might overflow an integer variable but be represented correctly in a double-precision, floating-point variable.
- The MATLAB Compiler treats the assertion functions (i.e., the functions whose names begin with `mb`, such as `mbintvector`) as type declarations. The MATLAB Compiler does not use these functions for type verification. Assertion functions will not appear in the output generated by the compiler.

Restrictions on Stand-Alone Applications

The restrictions and limitations noted in the previous section also apply to stand-alone applications. In addition, stand-alone applications cannot access:

- MATLAB debugging functions, such as `dbclear`
- MATLAB graphics functions, such as `surf`, `plot`, `get`, and `set`
- MATLAB `exists` function
- Calls to MEX-file functions because the MATLAB Compiler needs to know the signature of the function
- Simulink functions

Although the MATLAB Compiler 1.2 *can* compile M-files that call these functions, the MATLAB C and C++ Math libraries do not support them. Therefore, unless you write your own versions of the unsupported routines, the linker will report unresolved external reference errors.

Type Imputation

The MATLAB interpreter views all variables in M-files as a single object type, the MATLAB array. All MATLAB variables, including scalars, vectors, matrices, strings, cell arrays, and structures are stored as MATLAB arrays. The `mxArray` declaration corresponds to the internal data structure that MATLAB uses to represent arrays. The MATLAB array is the C language definition of a MATLAB variable.

To generate efficient C code from an M-file, the MATLAB Compiler analyzes how the variables in the M-files are assigned values and how these values are used. The goal of this analysis is to determine which variables can be downsized to a smaller data type (a C `int` or a C `double`). This analysis process is called *type imputation*; the MATLAB Compiler *imputes* a data type for a variable. For example, if the MATLAB Compiler sees

```
[m,n] = size(a);
```

then the MATLAB Compiler probably imputes the C `int` type for variables `m` and `n` because in MATLAB the result of this operation always yields integer scalars.

In some cases, the MATLAB Compiler has to read several lines of code in order to impute properly. For example, if the MATLAB Compiler sees

```
[m,n] = size(a);
m = m + .25;
```

then the MATLAB Compiler imputes the C `double` data type for variable `m`.

Note Specifying assertions and pragmas, as described later in this chapter, can greatly assist the type imputation process.

Type Imputation Across M-Files

If an M-file calls another M-file function, the MATLAB Compiler reads the entire contents of the called M-file function as part of the type imputation

analysis. For example, consider an M-file function named `profit` that calls another M-file function `getsales`:

```
function p = profit(inflation)
revenue = getsales(inflation);
...
p = revenue – costs;
```

To impute the data types for variables `p` and `revenue`, the MATLAB Compiler reads the entire contents of the file `getsales.m`.

Suppose you compile `getsales.m` to produce `getsales.mex`. When invoked, `profit.mex` calls `getsales.mex`. However, the MATLAB Compiler reads `getsales.m`. In other words, the runtime behavior of `profit.mex` depends on `getsales.mex`, but type imputations depend on `getsales.m`. Therefore, unless `getsales.m` and `getsales.mex` are synchronized, `profit.mex` may run peculiarly.

To ensure the files are synchronized, recompile every time you modify an M-file.

Optimization Techniques

Optimizing with Compiler Option Flags

Some MATLAB Compiler option flags optimize the generated code; other option flags generate compilation or runtime information. The two most important optimization option flags are –i (suppress array boundary checking) and –r (generate real variables only).

Consider the squibo M-file:

```
function g = squibo(n)
% The first n "squibonacci" numbers.
g = zeros(1,n);
g(1) = 1;
g(2) = 1;
for i = 3:n
   g(i) = sqrt(g(i–1)) + g(i–2);
end
```

We compiled squibo.m with various combinations of performance option flags on a Pentium Pro 200 MHz workstation running Linux. Then, we ran the resulting MEX-file 10 times in a loop and measured how long it took to run. Table D-1 shows the results of the squibo example using n equal to 10,000 and executing it 10 times in a loop.

Table D-1: Performance for n=10000, run 10 times

Compile Command Line	Elapsed Time (in sec.)	% Improvement
squibo.m (uncompiled)	5.7446	--
mcc –V1.2 squibo	3.7947	33.94
mcc –V1.2 –r squibo	0.4548	92.08
mcc –V1.2 –i squibo	2.7815	51.58
mcc –V1.2 –ri squibo	0.0625	98.91

As you can see from the performance table, –r and –i have a strong influence on elapsed execution time.

In order to understand how –r and –i improve performance, you need to look at the MEX-file source code that the MATLAB Compiler generates. When examining the generated code, focus on two sections:

- The comment section that lists the MATLAB Compiler's assumptions.
- The code that the MATLAB Compiler generates for loops. Most programs spend the vast majority of their CPU time inside loops.

An Unoptimized Program

Compiling `squibo.m` without any optimization option flags produces a MEX-file that runs only about 34% faster than the M-file. The MATLAB Compiler can do a lot better than that. To determine what's slowing things down, examine the MEX-file source code that the MATLAB Compiler generates.

Type Imputations for Unoptimized Case. After analyzing `squibo.m`, the MATLAB Compiler imputations in `squibo.c` are:

```
/***************** Compiler Assumptions ***************
 *
 *       CO_            complex scalar temporary
 *       IO_            integer scalar temporary
 *       g              complex vector/matrix
 *       i              integer scalar
 *       n              integer scalar
 *       sqrt           <function>
 *       squibo         <function being defined>
 *       zeros          <function>
 ******************************************************/
```

The MATLAB interpreter uses the MATLAB `mxArray` to store all variables. However, the MATLAB Compiler generates variables having additional data types like integer scalars. The MATLAB Compiler scrutinizes the M-file code and option flags for places where it can downsize a variable to a scalar; a scalar variable requires less accompanying code and memory.

The assumptions list for `squibo.c` shows variables (`CO_` and `IO_`) that do not appear in `squibo.m`. The MATLAB Compiler uses these variables to hold intermediate results. Although it is hard to make definitive judgments about intermediate variables, you generally want to keep them to a minimum. You

can sometimes make a program run faster by writing code (or applying option flags) that eliminates the need for some of them.

By default, the MATLAB Compiler generates code to handle all possible circumstances. Therefore, the MATLAB Compiler tends to impute that most matrices are complex. Complex matrices require more supporting code than do real matrices. MEX-files that manipulate complex vector/matrices run slower than those that manipulate real vector/matrices. The MATLAB Compiler imputes the complex vector/matrix type for variable `g`. Applying certain optimizations (described in this chapter) to `squibo.m` causes the MATLAB Compiler to impute the real vector/matrix type for variable `g`. When `g` is a real vector/matrix, `squibo` runs significantly faster.

The Generated Loop Code. Here is the C MEX-file source code that the MATLAB Compiler generates for the loop:

```
/* for i=3:n */
      for (IO_ = 3; IO_ <= n; IO_ = IO_ + 1)
      {
         i = IO_;
         /* g(i) = sqrt(g(i-1)) + g(i-2); */
         Mprhs_[0] = mccTempVectorElement(&g, (i - 1));
         Mplhs_[0] = 0;
         mccCallMATLAB(1, Mplhs_, 1, Mprhs_, "sqrt", 7);
         C0__r = mccImportScalar(&C0__i, 0, 0, Mplhs_[ 0 ],
                                 " (squibo, line 7): C0_");
         mccSetVectorElement(&g, mccRint(i),
                     (C0__r + (mccGetRealVectorElement(&g,
                     mccRint((i - 2))))),
                     (C0__i + mccGetImagVectorElement(&g,
                     mccRint((i - 2)))));
         /* end */
      }
```

The body of the M-file loop contains only one statement. The MATLAB Compiler expands this one statement into six C code statements. The most expensive of these six statements in terms of processing time is the callback to MATLAB (`mccCallMATLAB`). The MATLAB Compiler cannot impute any type or bounds information so it generates code to handle complex values and perform subscript checking. If you need this behavior, then the extra C code is

necessary. If you do not need this behavior, instruct the MATLAB Compiler to optimize the code further.

Optimizing with the -r Option Flag

Compiling an M-file with the –r option flag causes the MATLAB Compiler to impute the type real for all input, output, and temporary values and variables in the MEX-file. In other words, the generated MEX-file does not contain any code to handle complex numbers. Handling complex numbers normally requires a significant amount of code in a MEX-file, so eliminating this code can make a MEX-file run faster. Note that if the Compiler detects a complex number when compiling with the –r option, it will generate an error message.

Compiling with the –r option makes `squibo.mex` run about 92% faster than `squibo.m`. Obviously, the –r option has made a significant impact. To find out why, examine the MATLAB Compiler imputations and the generated loop code.

Type Imputations for -r. The –r option flag forces the MATLAB Compiler to assume that no variables are complex. With the –r option flag, the MATLAB Compiler assumptions are:

```
/***************** Compiler Assumptions ***************
 *
 *       IO_             integer scalar temporary
 *       RO_             real scalar temporary
 *       g               real vector/matrix
 *       i               integer scalar
 *       n               integer scalar
 *       realsqrt        <function>
 *       squibo          <function being defined>
 *       zeros           <function>
 ******************************************************/
```

Notice that variable `g` is now real. (Without –r, variable `g` is complex.)

Is it safe to compile with the –r option flag? Yes, because `g(i–1)` cannot become negative. As long as `g(i–1)` is nonnegative, `sqrt(g(i–1))` is always real.

The Generated Loop Code for -r. Since the MATLAB Compiler no longer has to generate code to handle complex values, the code within the loop reduces to only three C code statements. Here is the entire loop:

```
/* for i=3:n */
for (I0_ = 3; I0_ <= n; I0_ = I0_ + 1)
{
   i = I0_;
   /* g(i) = sqrt(g(i–1)) + g(i–2); */
   R0_ = sqrt((mccGetRealVectorElement(&g, mccRint((i – 1)))));
   mccSetRealVectorElement(&g, mccRint(i),
                     (R0_ + (mccGetRealVectorElement(&g,
                     mccRint((i – 2))))));
   /* end */
}
```

This code calculates square roots by calling the `sqrt` function in the standard C library. If you compile without the `–r` option, the resulting code makes a callback to MATLAB to calculate square roots. Since callbacks are relatively slow, eliminating the callback makes the program run much faster. The `sqrt` function in the standard C library only handles positive real input and only produces real output. The MATLAB Compiler can generate a call to the `sqrt` function of the standard C library because the `–r` option assures the MATLAB Compiler that `g(i–1)` is real and that the result of `sqrt(g(i–1))` is also real. Without the `–r` option, the MATLAB Compiler has to allow for the possibility that `g(i–1)` is complex or negative.

The `mccSetRealVectorElement` routine assigns a value (the sum of `sqrt(g(i–1))` and `g(i–2)`) to the `i`-th element of `g`. Before doing that assignment, the `mccSetRealVectorElement` routine checks the value of `i`:

- If `i` is zero or negative, `mccSetRealVectorElement` issues an error and terminates the program.
- If `i` is positive, `mccSetRealVectorElement` checks to see if `i` is less than or equal to the number of elements in `g`. If `i` is larger than the current number of elements in `g`, then `mccSetRealVectorElement` grows `g` until it is large enough to hold the new element (just as the MATLAB interpreter does).

Optimizing with the -i Option Flag

The –i option flag generates code that:

- Does *not* allow matrices to grow larger than their starting size.
- Does *not* check matrix bounds.

The MATLAB interpreter allows arrays to grow dynamically. If you do not specify –i, the MATLAB Compiler also generates code that allows arrays to grow dynamically. However, dynamic arrays, for all their flexibility, perform relatively slowly.

If you specify –i, the generated code does not permit arrays to grow dynamically. Any attempts to access an array beyond its fixed bounds will cause a runtime error. Using –i reduces flexibility but also makes array access significantly cheaper.

To be a candidate for compiling with –i, an M-file *must* preallocate all arrays. Use the `zeros` or `ones` function to preallocate arrays. (Refer to the "Optimizing by Preallocating Matrices" section later in this chapter.)

Caution If you fail to preallocate an array and compile with the –i option, your system will behave unpredictably and may crash.

If you forget to preallocate an array, the MATLAB Compiler cannot detect the mistake; the errors do not appear until runtime. If your program crashes with an error referring to:

- Bus errors
- Memory exceptions
- Phase errors
- Segmentation violations
- Unexplained application errors

then there is a good chance that you forgot to preallocate an array.

The –i option makes some MEX-files run faster, but generally, you have to use –i in combination with –r in order to see real speed advantages. For example, compiling `squibo.m` with –i does not produce any speed advantages, but

compiling `squibo.m` with a combination of `–i` and `–r` creates a very fast MEX-file.

Optimizing with a Combination of -r and -i Flags

Compiling programs with a combination of `–r` and `–i` produces code with all the speed advantages of both option flags. Compile with both option flags only if your M-file:

- Contains no complex values or operations
- Preallocates all arrays, and then never changes their size

Compiling `squibo.m` with `–ri` produces an extremely fast version of `squibo.mex`. In fact, the resulting `squibo.mex` runs more than 98% faster than `squibo.m`.

Type Imputations for -ri . When compiling with `–r` and `–i`, the MATLAB Compiler type imputations are:

```
/***************** Compiler Assumptions ***************
 *
 *       IO_             integer scalar temporary
 *       RO_             real scalar temporary
 *       g               real vector/matrix
 *       i               integer scalar
 *       n               integer scalar
 *       realsqrt        <function>
 *       squibo          <function being defined>
 *       zeros           <function>
 ******************************************************/
```

The MATLAB Compiler's type imputations for `–ri` are identical to the imputations for `–r` alone. Additional performance improvements are due to the generated loop code.

The Generated Loop Code for -ri. The MATLAB Compiler generates the loop code:

```
/* for i=3:n */
for (IO_ = 3; IO_ <= n; IO_ = IO_ + 1)
{
   i = IO_;
   /* g(i) = sqrt(g(i-1)) + g(i-2); */
   RO_ = sqrt((mccPR(&g)[((i-1)-1)]));
   mccPR(&g)[(i-1)] = (RO_ + (mccPR(&g)[((i-2)-1)]));
   /* end */
}
```

This loop is very short and contains no callbacks to MATLAB. The `–ri` loop is more efficient than the `–r` loop because subscript checking is eliminated. In addition, the `–ri` loop code gains some speed by using the C assignment operator (=) to assign values. By contrast, the `–r` loop code assigns values by calling the relatively expensive `mccSetRealVectorElement` function.

Optimizing Through Assertions

By adding *assertions* to an M-file, you can guide the MATLAB Compiler's type imputations. Assertions help the MATLAB Compiler recognize where it can generate simpler data types (and the associated simpler code).

Assertions are M-file functions (installed by the MATLAB Compiler) whose names begin with the letters `mb`, which stands for "must be."

Function	Assertions
`mbscalar(x)`	x must be a scalar.
`mbvector(x)`	x must be a vector.
`mbint(x)`	x must be an integer.
`mbchar(x)`	x must be a character string.
`mbreal(x)`	x must be real (not complex).
`mbcharscalar(x)`	x must be a character scalar.
`mbintscalar(x)`	x must be an integer scalar.

Function	Assertions
`mbrealscalar(x)`	x must be a real scalar.
`mbcharvector(x)`	x must be a vector of characters.
`mbintvector(x)`	x must be a vector of integers.
`mbrealvector(x)`	x must be a vector of real numbers.

The MATLAB Compiler and MATLAB interpreter both recognize assertions and issue error messages if a variable does not satisfy a particular assertion. For example, if variable `x` holds the value 4.5, then

```
mbint(x)
```

triggers an error message in both the MATLAB Compiler and interpreter because `x` is not an integer.

The MATLAB Compiler, unlike the MATLAB interpreter, uses assertions to guide type imputations. Therefore, the MATLAB Compiler imputes the C `int` data type for variable `x`. Since the MATLAB interpreter does not support different C data types, the `mbint` assertion does not influence the MATLAB interpreter. However, since the assertions are M-files that check the value of their input, the MATLAB interpreter executes extra code, which will cause an error if the value of the variable cannot be represented in the specified C type.

Although you can use assertions on any variable in an M-file, you typically use assertions to constrain the data types of input arguments. For example, `mbintscalar` forces the input argument, `n`, to `myfunc` to be an integer scalar:

```
function y = myfunc(n)
mbintscalar(n);
```

A single assertion can have a fairly wide ranging influence. For example, if you assert that variable `a` is real, then the MATLAB Compiler also assumes that the variables to which you assign `a` are also real. For instance, in the code

```
mbreal(a);
b = a + 2;
```

the `mbreal` assertion allows the MATLAB Compiler to impute the real type for both `a` and `b`.

Note that the MATLAB Compiler does not automatically impute that all variables that interact with variable a are real. For example, although the MATLAB Compiler imputes the real type for a

```
mbreal(a);
b = a + 2i;
```

the MATLAB Compiler still imputes the complex type for variable b.

An Assertion Example

To explore assertions, consider the M-file

```
function [g,h] = fibocert(a,b)

% $Revision: 1.1 $

% Part 1 contains an assertion
mbreal(a);   % Assert that "a" contains only real numbers.
n = max(size(a));
g = zeros(1,n);
g(1) = a(1);
g(2) = a(2);
for c = 3:n
   g(c) = g(c – 1) + g(c – 2) + a(c);
end

% Part 2 contains no assertions
n = max(size(b));
h = zeros(1,n);
h(1) = b(1);
h(2) = b(2);
for c = 3:n
   h(c) = h(c – 1) + h(c – 2) + b(c);
end
```

This M-file consists of two parts labeled Part 1 and Part 2. Both parts are identical except that Part 1 contains the assertion

```
mbreal(a);
```

`fibocert` accepts real data into argument `a` and either real or complex data into argument `b`. Compiling `fibocert.m`

```
mcc –V1.2 fibocert
```

generates a telling list of imputations, among them:

```
*       a               real vector/matrix
*       b               complex vector/matrix
*       c               integer scalar
*       g               real vector/matrix
*       h               complex vector/matrix
```

The MATLAB Compiler imputes the complex vector/matrix type for variable `b`. The `mbreal` assertion forces the MATLAB Compiler to impute the real vector/matrix type for variable `a`. If you remove the `mbreal` assertion, the MATLAB Compiler imputes the complex vector/matrix type for variable `a`.

Since variable `b` is complex, the MATLAB Compiler imputes the complex vector/matrix type for variable `h`. The side effect of asserting that variable `a` is real is that the MATLAB Compiler imputes that variable `g` is also real.

Note the difference between the `–r` MATLAB Compiler option and the `mbreal` assertion. The `–r` option tells the MATLAB Compiler to assume that there are no complex variables anywhere in the file. The `mbreal` assertion gives the MATLAB Compiler advice about a particular variable. If compiled with the `–r` option, the resulting MEX-file does not accept any complex data, for example:

```
mcc –V1.2 –r fibocert
f1 = 1:0.5:1000;
f2 = f1 + 4i;
[fibor,fiboc] = fibocert(f1,f2);
??? Runtime Error: Encountered a complex value where a real was
expected
(compiling with –l may give line number)
```

Optimizing with Pragmas

The MATLAB Compiler provides three *pragmas* that affect code optimization. You can use these pragmas to send optimization information to the MATLAB Compiler. The three optimization pragmas are:

- `%#inbounds`
- `%#realonly`
- `%#ivdep`

All pragmas begin with a percent sign (`%`) and thus appear as comments to the MATLAB interpreter. Therefore, the MATLAB interpreter ignores all pragmas.

The `%#inbounds` and `%#realonly` pragmas are the equivalent of the `–i` and `–r` MATLAB Compiler option flags, respectively. Placing `%#inbounds` in an M-file causes the MATLAB Compiler to generate the same source code as compiling with the `–i` option flag. You can place `%#inbounds` and `%#realonly` anywhere within an M-file; these pragmas affect the whole file.

The `%#ivdep` pragma tells the MATLAB Compiler to ignore vector dependencies in the assignment statement that immediately follows it. Using `%#ivdep` can speed up some assignment statements, but using it incorrectly causes assignment errors.

Unlike `%#inbounds` and `%#realonly`, the `%#ivdep` pragma has no option flag equivalent. Also unlike `%#inbounds` and `%#realonly`, the placement of `%#ivdep` within an M-file is critical. A `%ivdep` pragma only influences the statement in the M-file that immediately follows it. If that statement happens to be an assignment statement, `%#ivdep` may be able to optimize it. If that statement is not an assignment statement, `%#ivdep` has no effect. You can place multiple `%#ivdep` pragmas inside an M-file.

%#inbounds

`%#inbounds` is the pragma version of the MATLAB Compiler option flag `–i`. The syntax of the inbounds pragma is:

```
%#inbounds
```

This pragma has no effect on C++ generated code (i.e., if the `–p` MATLAB Compiler option flag is used). Placing the pragma anywhere inside an M-file

has the same effect as compiling that file with –i. The %#inbounds pragma (or –i) causes the MATLAB Compiler to generate C code that:

- Does not check array subscripts to determine if array indices are within range.
- Does not reallocate the size of arrays when the code requests a larger array. For example, if you preallocate a 10-element vector, the generated code cannot assign a value to the 11th element of the vector.
- Does not check input arguments to determine if they are real or complex.

The %#inbounds pragma can make a program run significantly faster, but not every M-file is a good candidate for %#inbounds. For instance, you can only specify %#inbounds if your M-file preallocates all arrays. You typically preallocate arrays with the zeros or ones functions.

Note If an M-file contains code that causes an array to grow, then you cannot compile with the %#inbounds option. Using %#inbounds on such an M-file produces code that fails at runtime.

Using %#inbounds means you guarantee that your code always stays within the confines of the array. If your code does not, your compiled program will probably crash.

The %#inbounds pragma applies only to the M-file in which it appears. For example, suppose %#inbounds appears in alpha.m. Given the command:

```
mcc –V1.2 alpha beta
```

the %#inbounds pragma in alpha.m has no influence on the way the MATLAB Compiler compiles beta.m.

%#ivdep

The %#ivdep pragma tells the MATLAB Compiler to ignore vector dependencies in the assignment statement that immediately follows it. The syntax of the Ignore-vector-dependencies (ivdep) pragma is:

```
%#ivdep
```

This pragma has no effect on C++ generated code (i.e., if the –p MATLAB Compiler option flag is used). Since the `%#ivdep` pragma only affects a single line of an M-file, you can place multiple `%#ivdep` pragmas into an M-file. Using `%#ivdep` can speed up some assignment statements, but using `%ivdep` incorrectly causes assignment errors.

The `%#ivdep` pragma borrows its name from a similar feature in many vectorizing C and Fortran compilers.

This is an M-file function that does not (and should not) contain any `%#ivdep` pragmas:

```
function a = mydep
a = 1:8;
a(3:6) = a(1:4);
```

Compiling this program and then running the resulting MEX-file yields the correct answer, which is:

```
mydep
ans =
     1     2     1     2     3     4     7     8
```

The assignment statement

```
a(3:6) = a(1:4)
```

accesses values on the right side of the assignment that have been changed earlier by the left side of the assignment. (This is the “vector dependency” referred to in the name.) The MATLAB Compiler deals with this problem by creating a temporary matrix for such a computation, in effect doing the assignment in two steps:

```
TEMP = A(1:4);
A(3:6) = TEMP;
```

If you mistakenly place an `%#ivdep` pragma in the M-file

```
function a = mydep
a = 1:8;
%#ivdep
a(3:6) = a(1:4);
```

then the resulting MEX-file does not create a temporary matrix and consequently calculates the wrong answer:

```
mydep
ans =
     1     2     1     2     1     2     7     8
```

The MATLAB Compiler creates a temporary matrix to handle many assignments. In some situations, the temporary matrix is not needed and causes MEX-files to run more slowly. Use `%#ivdep` to flag those assignment statements that do not need a temporary matrix. Using `%#ivdep` typically results in faster code *but the code may not be correct if there are vector dependencies*.

For example, this M-file benefits from `%#ivdep`.

```
function [A,p] = lu2(A)
[m,n] = size(A);
p = (1:m)';

for k = 1:min(m,n)-1
  q = min(find(abs(A(k:m,k)) == max(abs(A(k:m,k))))) + k-1;
  if q ~= k
     p([k q]) = p([q k]);
     A([k q],:) = A([q k],:);
  end
  if A(k,k) ~= 0
     A(k+1:m,k) = A(k+1:m,k)/A(k,k);
     for j = k+1:n;
         %# ivdep
         A(k+1:m,j) = A(k+1:m,j) - A(k+1:m,k)*A(k,j);
     end
  end
end
```

The `%#ivdep` pragma tells the MATLAB Compiler that the elements being referenced on the right side are independent of the elements on the left side. Therefore, the MATLAB Compiler can create a MEX-file that calculates the

correct answer without generating a temporary matrix. Table D-2 shows that the `%#ivdep` pragma had a significant impact on the performance of `lu2`.

Table D-2: Performance for lu2(magic(100)), run 20 times

MEX-File	Elapsed Time (in sec.)
`lu2` containing `%#ivdep`	1.8610
`lu2` omitting `%#ivdep`	2.6102

%#realonly

`%#realonly` is the pragma version of the MATLAB Compiler option flag `–r`. The syntax of the real only pragma is:

```
%#realonly
```

This pragma has no effect on C++ generated code (i.e., if the `–p` MATLAB Compiler option flag is used). Placing the pragma anywhere inside an M-file has the same effect as compiling that file with `–r`. Specifying both `–r` and `%#realonly` has the same effect as specifying only `%#realonly`.

`%#realonly` tells the MATLAB Compiler to generate source code with the assumption that all input data, output data, and temporary data in the M-file are real. Since all data are real, all operations are also real.

In this example, the function `squibo2` contains a `%#realonly` pragma.

```
function g = squibo2(n)
#%realonly
g = ones(1,n);
for i=4:n
   g(i) = sqrt(g(i–1)) + g(i–2) + g(i–3);
end
```

The `%#realonly` pragma forces the MATLAB Compiler to generate real-only data and operations.

```
mcc –V1.2 squibo2
```

An alternative way of ending up with the same code is to omit the `%#realonly` pragma and to compile with

```
mcc –V1.2 –r squibo2
```

The %#realonly pragma applies only to the M-file in which it appears. For example, suppose %#realonly appears in alpha.m. Given the command

```
mcc –V1.2 alpha beta
```

the %#realonly pragma in alpha.m has no influence on the way the MATLAB Compiler compiles beta.m.

Optimizing by Avoiding Complex Calculations

The MATLAB Compiler adds three special functions to MATLAB—reallog, realpow, and realsqrt—that are real-only versions of the log, .^ (array power), and sqrt functions. The three real-only functions accept only real values as input and return only real values as output. Because they do not have to handle complex values, the three real-only functions execute faster than log, .^, and sqrt.

Function	Description
Y = reallog(X)	Return the natural logarithm of the elements of X, if X is positive. Otherwise signal an error.
Z = realpow(X,Y)	Return the elements of X raised to the Y power. If X is negative and Y is not an integer, signal an error.
Y = realsqrt(X)	Return the square root of the elements of X, if X is nonnegative. Otherwise return an error.

For example, consider the simple M-file function:

```
function h = powwow1(a,b)
h = a .^ b;
```

This coding of powwow1 is appropriate if there is even a slight possibility that a, b, or h is complex. (Note that h can be complex even if a and b are both real, e.g., a = –1 and b = 0.5.) On the other hand, if you are certain that a, b, and h are always going to be real, then a better way to write the M-file is:

```
function h = powwow2(a,b)
h = realpow(a,b);
```

If you invoke `powwow2` and mistakenly specify a complex value for `a` or `b`, then the function issues an error message and halts execution.

Effects of the Real-Only Functions

The MATLAB Compiler assumes that all input and output arguments to a real-only function are real. For example, since `powwow2` calls `realpow`, the MATLAB Compiler imputes the real type for all of `realpow`'s input and output arguments (`a`, `b`, and `h`). Since all variables in `powwow2` are real, the MATLAB Compiler generates no code to handle complex data.

Automatic Generation of the Real-Only Functions

If you compile with the `–r` option flag, the MATLAB Compiler automatically converts `log`, `sqrt`, and `.^` to their real versions. For example, compiling `powwow1` with the `–r` option flag generates the same code as compiling `powwow2` without the `–r` option flag.

If the result of a call to `log` or `sqrt` is guaranteed to be real, the MATLAB Compiler often imputes the function call to be real only. For example, since the number 2 is both real and positive, the MATLAB Compiler generates code for

```
a = sqrt(2);
```

as if the code were written

```
a = realsqrt(2);
```

As another example, suppose an M-file contains:

```
a = sqrt(b);
mbreal(a);
```

Since variable `a` is guaranteed to be real, the MATLAB Compiler converts `sqrt` to `realsqrt` and further imputes the real type for variable `b`.

For more information about `reallog`, `realpow`, and `realsqrt`, see their corresponding reference pages in Chapter 6.

Optimizing by Avoiding Callbacks to MATLAB

Callbacks to the MATLAB interpreter slow down a MEX-file's performance. The MATLAB Compiler generates callbacks to handle some MATLAB functions, particularly the more complicated and time-consuming ones. The

MATLAB Compiler handles simple functions without making a callback. For example, to compile the call to the relatively simple `ones` function

```
a = ones(5,7)
```

the MATLAB Compiler does not generate a callback to the MATLAB interpreter. The MATLAB Compiler generates a call to the `mccOnesMN` routine contained in the MATLAB Compiler Library:

```
mccOnesMN(&a, 5, 7);
```

On the other hand, the `lu` call is a complicated function. Therefore, the MATLAB Compiler translates

```
X = lu(A);
```

into the callback:

```
mccCallMATLAB(1, Mplhs_, 1, Mprhs_, "lu", 2);
```

This `mccCallMATLAB` routine asks the MATLAB interpreter to compute the `lu` decomposition.

Whenever possible, you should try to produce code that minimizes these callbacks. Here are a few suggestions:

- Identify when callbacks are occurring.
- Avoid calling other M-files that themselves call built-in functions.
- Compile referenced M-files along with the target M-file.

This section takes a closer look at these suggestions.

Identifying Callbacks

There are two ways to find callbacks to MATLAB:

- Study the MEX-file source code the MATLAB Compiler generates and look for calls to `mccCallMATLAB`.
- Invoke the MATLAB Compiler with the `–w` option flag.

To study callbacks, consider the function M-file named `mycb`:

```
function g = mycb(n)
g = ones(1,n);
for i = 4:n
   temp1 = log(g(i–1) + g(i–2));
   temp2 = tan(g(i–3));
   g(i) = temp1 + temp2;
end
log10(g);
```

Compiling `mycb` with the `–w` option flag shows that `mycb` makes a number of callbacks to built-in functions:

```
mcc –V1.2 –w mycb
Warning: MATLAB callback of 'tan' will be slow (line 5)
Warning: MATLAB callback of 'log10' will be slow (line 8)
```

This output tells you that the MATLAB Compiler generates `mccCallMATLAB` calls for the `tan` and `log10` functions. Notice that the `ones` and `log` functions do not appear in this list of callbacks. That is because the MATLAB Compiler handles `ones` by generating a call to `mccOnesMN` and `log` by generating a call to `mcmLog` (both routines are in the MATLAB Compiler Library). The calls to `mccOnesMN` and `mcmLog` are resolved at link time.

Compiling Multiple M-Files into One MEX-File

When M-files call other M-files, which in turn may call additional M-files, the called files are called *helper functions*. If your M-file uses helper functions, you can often improve performance by building all accessed M-files into a single MEX-file. The MATLAB Compiler always compiles all functions that appear in the same M-file into the resulting application as helper functions.

Consider the function `fibomult.m`

```
function g = fibomult(n)
g = ones(1,n);
for i = 3:n
   g(i) = myfunc(g(i–1)) + g(i–2);
end
```

where myfunc is defined as:

```
function z = myfunc(x)
temp1 = x .* 10 .* sin(x);
z = round(temp1);
```

If you compile fibomult by itself, the resulting code has to do a callback to MATLAB in order to find myfunc. Since callbacks to MATLAB are slow, performance is not optimal. The problem is compounded by the fact that this callback happens one time for each iteration of the loop. The results on our machine are:

```
mcc –V1.2 fibomult
tic; fibomult(10000); toc
elapsed_time =
   16.4851
```

If you compile fibomult.m and myfunc.m together, the resulting code does not contain any callbacks to MATLAB and performance is significantly better:

```
mcc –V1.2 fibomult myfunc
tic; fibomult(10000); toc
elapsed_time =
   0.0690
```

Compiling two M-files on the same mcc command line produces only one MEX-file. The resulting MEX-file has the same root filename as the first M-file on the compilation command line. For example,

```
mcc –V1.2 fibomult myfunc
```

creates fibomult.mex (not myfunc.mex).

Note You can build several M-files into one MEX-file. However, no matter how many input M-files there are, MEX-files still offer only one entry point. Thus, a MEX-file is different from a library of C routines, each of which can be called separately.

Using the -h Option. You can also compile multiple M-files into a single MEX-file or stand-alone application by using the –h option of the mcc command. The –h option compiles all helper functions into a single MEX-file or stand-alone

application. In this example, you can compile `fibomult.m` and `myfunc.m` together into the single MEX-file, `fibomult.mex`, by using:

```
mcc –V1.2 –h fibomult
```

Using the `–h` option is equivalent to listing the M-files explicitly on the `mcc` command line.

The `–h` option purposely does not include built-in functions or functions that appear in the MATLAB M-File Math Library portion of the C/C++ Math Libraries. This prevents compiling functions that are already part of the C/C++ Math Libraries. If you want to compile these functions as helper functions, you should specify them explicitly on the command line. For example, use

```
mcc –V1.2 minimize_it fmins
```

instead of

```
mcc –V1.2 –h minimize_it
```

Note Due to Compiler restrictions, some of the MATLAB 5 versions of the M-files for the C and C++ Math Libraries do not compile as is. The MathWorks has rewritten these M-files to conform to the Compiler restrictions. The modified versions of these M-files are in `<matlab>/extern/src/tbxsrc`, where `<matlab>` represents the top-level directory where MATLAB is installed on your system.

Compiling MATLAB Provided M-Files. Callbacks sometimes appear in unexpected places. For example, consider the function M-file:

```
function g = mypoly(n)
m = magic(n);
m = m / 5;
g = poly(m);
```

MATLAB implements `poly` as an M-file rather than as a built-in function. Compiling `poly.m` along with your own M-file does not improve the performance of `mypoly.m`.

The –w option flag reveals the problem:

```
mcc –V1.2 –w mypoly poly
Warning: MATLAB callback of 'magic' will be slow (line 2)
Warning:
You are compiling a copyrighted M-file. You may use the resulting
copyrighted C source code,  object code, or linked binary in your
own work,  but you may not distribute,  copy,  or sell it without
permission from The MathWorks or other copyright holder.

... in function 'poly'
Warning: MATLAB callback of 'eig' will be slow
  ... in function 'poly', line 24
Warning: MATLAB callback of 'isfinite' will be slow
  ... in function 'poly', line 32
Warning: MATLAB callback of 'sort' will be slow
  ... in function 'poly', line 42
Warning: MATLAB callback of 'conj' will be slow
  ... in function 'poly', line 42
Warning: MATLAB callback of 'sort' will be slow
  ... in function 'poly', line 42
Warning: MATLAB callback of 'isequal' will be slow
  ... in function 'poly', line 42
```

In other words, `poly` itself calls many MATLAB built-in functions. Consequently, compiling `poly` does not increase its execution speed.

Caution If you compile a copyrighted M-file, you may use the resulting copyrighted C source code, object code, or linked binary in your own work, but you may not distribute, copy, or sell it without permission from The MathWorks, Inc. or other copyright holder.

Compiling M-Files That Call feval

The first argument to the `feval` function is the name of another function. The MATLAB interpreter allows you to specify the name of this input function at runtime. In a similar manner, the code generated by the MATLAB Compiler allows you to specify the name of this input function at runtime. However, the MATLAB Compiler also lets you specify the name of this input function at

compile time. Specifying the name of this input function at compile time can improve performance.

For example, consider a function M-file named `plot1` containing a call to `feval`:

```
function plot1(fun,x)
y = feval(fun,x)
plot(y);
```

If you compile `plot1` in the usual manner

```
mcc –V1.2 plot1
```

you must invoke `plot1` like this

```
plot1('myfun',7);
```

`plot1.mex` makes a callback to MATLAB to find the function `myfun`. To avoid the expense of the callback, specify on the compilation command line the name of the function that you want to pass to `feval`. For example, to pass `myfun` as the argument to `feval`, invoke the MATLAB Compiler as:

```
mcc –V1.2 plot1 fun=myfun
```

If an M-file contains multiple `feval` calls, you can pass the names of none, some, or all of the input function names at compile time. For example, consider an M-file named `plotf.m` that contains two calls to `feval`:

```
function plotf(fun1,fun2,x)
hold on
y = feval(fun1,x);
plot(x,y,'g+');
z = feval(fun2,x);
plot(x,z,'b');
```

The fastest possible code for `plotf` is generated by specifying the names of both input functions on the compilation command line. For example, the command line

```
mcc –V1.2 plotf fun1=orange fun2=lemon
```

causes the MATLAB Compiler to generate code in `plotf.c` that explicitly calls `orange` and `lemon`. Using the `fun=`*`feval_arg`* syntax creates faster runtime performance; however, this syntax eliminates the inherent flexibility of `feval`.

No matter what you specify as the first and second arguments to `plotf`, for instance:

```
plotf('dumb','dumber',0:pi/100:pi);
```

`plotf.mex` still calls `orange` and `lemon`.

Many MATLAB functions are themselves M-files. Many of these M-files (for example `fzero` and `ode23`) call `feval`. For example, consider an M-file named `ham.m` containing the line:

```
y = fzero(fun,8);
```

Since `fzero` calls `feval`, you can tell the MATLAB Compiler which function `fzero` must evaluate, for example:

```
mcc –V1.2 ham fun=greenegg
```

Note The facility described in this section is supported for backwards compatibility only and may be removed in the future. For additional information on `feval`, see "Using feval" in Chapter 5 .

Optimizing by Preallocating Matrices

You should preallocate matrices (or vectors) whenever possible. Preallocating matrices eliminates the need for costly memory reallocations. For example, consider the M-file:

```
function myarray = squares1(n)
for i = 1:n
     myarray(i) = i * i;
end
```

The `squares1` function runs relatively slowly because the MATLAB interpreter must grow the size of `myarray` with each pass through the loop. To grow `myarray`:

- The MATLAB interpreter must ask the operating system to allocate more memory.
- In some cases, the MATLAB interpreter must copy the previous contents of `myarray` to a new region of memory.

Growing `myarray` is expensive, particularly when `i` becomes large.

A better approach is to allocate a row vector of `n` elements prior to the beginning of the loop, typically by calling the `zeros` or `ones` function

```
function myarray = squares2(n)
myarray = zeros(1,n);
for i = 1:n
     myarray(i) = i * i;
end
```

The `zeros` function in `squares2` allocates enough space for all `n` elements of the vector. Thus, `squares2` does not have to reallocate space at every iteration of the loop. `squares2` runs significantly faster than `squares1`, in both the interpreted and compiled forms. Table D-3 shows the execution times for the interpreted and compiled versions of the two M-files.

Table D-3: Performance for n=5000, run 10 times

M-file Function	Form	Elapsed Time (sec.)
`squares1` (not preallocated)	Interpreted	9.0298
	Compiled	1.1537
`squares2` (preallocated)	Interpreted	2.1329
	Compiled	0.0622

Optimizing by Vectorizing

The MATLAB interpreter runs vectorized M-files faster than M-files that contain loops. In fact, the MATLAB interpreter runs vectorized M-files so efficiently that compiling vectorized M-files into MEX-files rarely brings big performance improvements. (See *Using MATLAB* for more information on how to vectorize M-files.)

To demonstrate the influence of vectorization, consider a nonvectorized M-file containing an unnecessary `for` loop:

```
function h = novector(stop)
for angle = 1:stop
   radians = (angle ./ 180) .* pi;
   h(angle) = sin(radians);
end
```

Vectorizing the `angle` variable eliminates the `for` loop:

```
function h = yovector(stop)
angle = 1:stop;
radians = (angle ./ 180) .* pi;
h = sin(radians);
```

Table D-4 shows the execution times for the interpreted and the compiled versions of the two M-files.

Table D-4: Performance for n=19200

M-file	Form	Elapsed Time (sec.)
`novector`	Interpreted	23.1750
(not vectorized)	Compiled	2.5108
`yovector`	Interpreted	0.0256
(vectorized)	Compiled	0.0231

As expected, the vectorized form of the program runs significantly faster than the nonvectorized form. However, compiling the vectorized version has no significant impact on performance.

The Generated Code

MEX-File Source Code Generated by mcc

The contents of MEX-files produced by the MATLAB Compiler depend, of course, on the contents of the M-files being compiled. However, most MEX-files produced by the MATLAB Compiler share this common format:

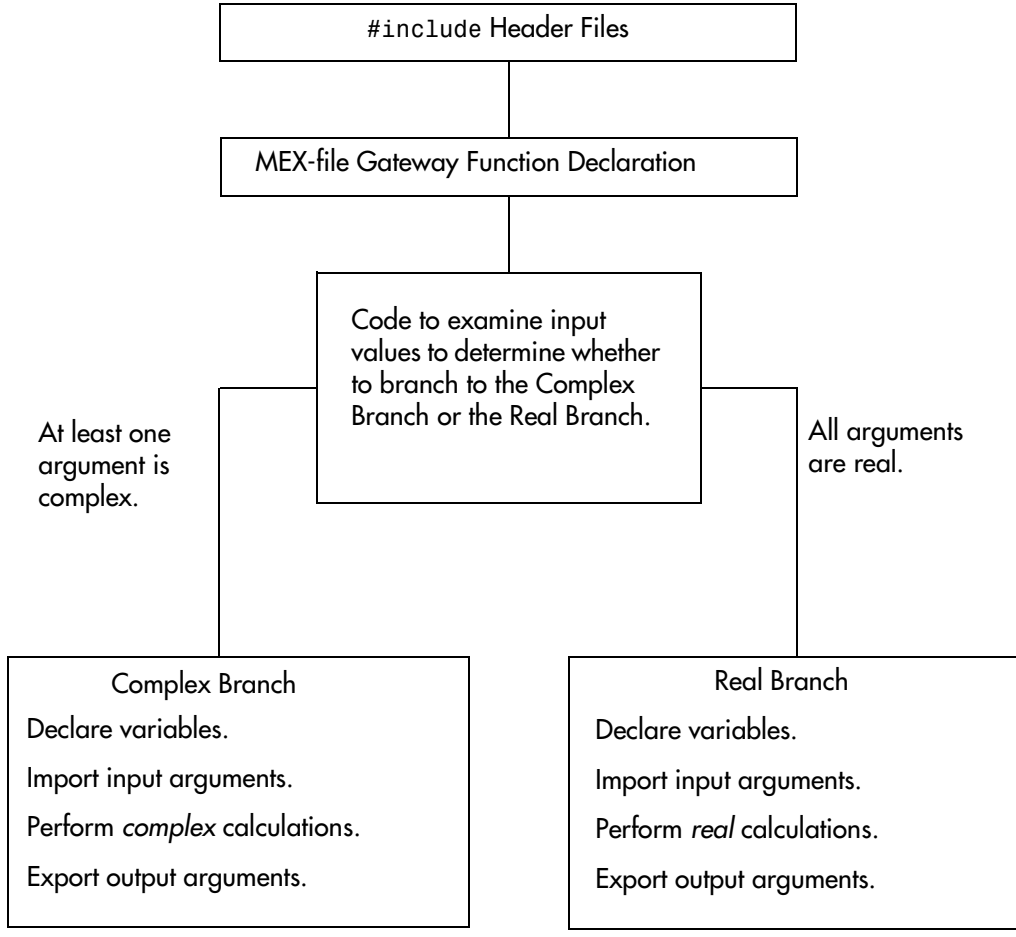


Figure D-1: Structure of MEX-File Source Code Generated by Compiler

For example, consider the simple M-file `fibocon.m`:

```
function g = fibocon(n,x)
g(1)=1; g(2)=1;
for i=3:n
   g(i) = g(i – 1) + g(i – 2) + x;
end
```

Compiling `fibocon.m`

```
mcc –V1.2 fibocon
```

yields the MEX-file source code that this section details.

Header Files

Invoking the MATLAB Compiler without the `–e` option flag causes the MATLAB Compiler to include these header files inside the MEX-file source code:

```
#include <math.h>
#include "mex.h"
#include "mcc.h"
```

This table lists the included header files.

Header File	Contains Declarations and Prototypes
`math.h`	ANSI/ISO C Math Library
`mex.h`	MEX-files
`mcc.h`	MATLAB Compiler Library

MEX-File Gateway Function

All MEX-files, whether generated by the MATLAB Compiler or written by a programmer, must contain the standard MEX-file gateway routine. The gateway routine is the entry point for all MEX-files. The gateway routine may in turn call other routines in the MEX-file.

The name of the gateway routine is always `mexFunction`, which takes the form of:

```
void
mexFunction(
    int nlhs_,
    mxArray *plhs_[],
    int nrhs_,
    const mxArray *prhs_[]
)
```

`mexFunction` has the same prototype regardless of the number of arguments that the MEX-file expects to receive or to return. See the *Application Program Interface Guide* for details on `mexFunction`.

Complex Argument Check

The MATLAB Compiler generates the following input test code to determine if the input data contains any imaginary parts:

```
int ci_;
int i_;
/*  Argument Checking  */
for (ci_=i_=0; i_<nrhs_; ++i_)
{
   if ( mccPI(prhs_[i_]) )
   {
     ci_ = 1;
     break;
   }
}
if (ci_)
{
  /* Complex Branch */
  ...
}
else
{
  /* Real Branch */
  ...
}
```

The input test code examines each input argument to determine whether the argument has an imaginary component. If any input variable has an imaginary component, the input test code sets the flag variable `ci_` to 1. If `ci_` is set to 1, the program executes the Complex Branch. If `ci_` has not been set to 1, then the program executes the Real Branch. See the *Application Program Interface Guide* for details on the `pi` field of the `mxArray` structure.

If the MATLAB Compiler determines that none of the input arguments is complex, the MATLAB Compiler does not generate a Complex Branch. Similarly, compiling with the `–r` option flag forces the MATLAB Compiler to suppress generating the Complex Branch. If you pass complex input to a function that has no Complex Branch, then the function returns an error message and exits.

If you compile with both the `–r` and `–i` option flags, the input test code is not present. Consequently, the MEX-file does not check input arguments. If you pass complex input to a function having no input test code, the function may produce incorrect results.

Computation Section — Complex Branch and Real Branch

The computation section performs the computations defined in the M-file. As noted earlier, the computation section typically contains both a Complex Branch and a Real Branch; however, only one of these gets executed at runtime. Both branches have the same basic organization, which is:

- A list of commented MATLAB Compiler assumptions and then the variable declarations themselves
- Code to import the passed argument values to variables
- Code to perform the calculations
- Code to return the results back to MATLAB

Declaring Variables . Each branch begins with a commented list of MATLAB Compiler assumptions and an uncommented list of variable declarations. For example, the imputations for the Complex Branch of `fibocon.c` are:

```
/***************** Compiler Assumptions ****************
 *
 *       IO_             integer scalar temporary
 *       fibocon         <function being defined>
 *       g               complex vector/matrix
 *       i               integer scalar
 *       n               complex vector/matrix
 *       x               complex vector/matrix
 ******************************************************/
```

Variable declarations follow the assumptions. For example, the variable declarations of `fibocon.c` are:

```
mxArray g;
mxArray n;
mxArray x;
int i = 0;
int IO_ = 0
```

This table shows the mapping between commented MATLAB Compiler assumptions and the actual C variable declarations.

Assumption	C Type
Integer scalar	`int`
Real scalar	`double`
Complex vector/matrix	`mxArray`
Real vector/matrix	`mxArray`

All vectors and matrices, whether real or complex, are declared as `mxArray` structures. All scalars are declared as simple C data types (either `int` or `double`). An `int` or `double` consumes far less space than an `mxArray` structure.

The MATLAB Compiler tries to preserve the variable names you specify inside the M-file. For instance, if a variable named `jam` appears inside an M-file, then the analogous variable in the compiled version is usually named `jam`.

The MATLAB Compiler typically generates a few "temporary" variables that do not appear inside the M-file. All variables whose names end with an underscore (`_`) are temporary variables that the MATLAB Compiler creates in order to help perform a calculation.

When coding an M-file, you should pick variable names that do not end with an underscore. For example, consider the two variable names:

```
hoop = 7;     % Good variable name
hoop_ = 7;    % Bad variable name
```

In addition, you should pick variable names that do not match reserved words in the C language; for example:

```
switches = 7;% Good variable name
switch = 7;  % Bad variable name because switch is a C keyword
```

Importing Input Arguments. When you invoke a MEX-file, MATLAB automatically assigns the passed input arguments to the `prhs` parameter of the `mexFunction` routine. For example, invoking `fibocon` as

```
fibocon(20,5)
```

causes MATLAB to assign the first input argument (`20`) to `*prhs[0]` and the second argument (`5`) to `*prhs[1]`. The generated MEX-file source code copies the relevant values that `prhs` points to into variables with more intuitive names. For example, `fibocon.c` uses this code to copy the contents of `*prhs[0]` to variable `n` and the contents of `*prhs[1]` to variable `x_r`:

```
n = mccImportReal(&n_set_, 0, (nrhs_>0) ? prhs_[0] : 0, "n");
x_r = mccImportScalar(&x_i, &x_set_, 0, (nrhs_>1) ? prhs_[1] : 0,
                      " (fibocon, line 1): x");
mccComplexInit(g);
```

Each element pointed to by the `prhs` array has the `mxArray` type. However, not all variables in the MEX-file source code have the `mxArray` type; some variables are `int` or `double`. The MEX-file source code must copy the relevant fields of each `mxArray` input argument into `int` and `double` variables. To do the copying, MEX-files rely on a family of *import routines* from the MATLAB Compiler Library. All import routines have names beginning with `mccImport`.

For example, `*prhs[0]` corresponds to variable `n`. However, `*prhs[0]` is an `mxArray` and variable `n` is a `double`. The `mccImportReal` import routine converts the `mxArray` into the `double` and assigns the `double` to variable `n`. Similarly, `mccImport` converts the second input `mxArray`, `*prhs[1]`, into the C `int` variable `x_r`.

The generated MEX-file source code uses some of the fields of the `mxArray` type differently than documented in the *Application Program Interface Guide*. To initialize these fields, the generated MEX-file source code calls one of its *matrix initialization routines*, such as `mccComplexInit`.

Performing Calculations . Following the variable declarations, the MATLAB Compiler generates the code to perform the calculations. For example, the calculations section of the real branch of `fibocon.c` is:

```
/* g(1)=1; g(2)=1; */
mccSetRealVectorElement(&g, 1, (double)1);
mccSetRealVectorElement(&g, 2, (double)1);
/* for i=3:n */
if( !n_set_)
{
   mexErrMsgTxt( "variable n undefined, line 3" );
}
for (I0_ = 3; I0_ <= n; I0_ = I0_ + 1)
{
   i = I0_;
   /* g(i) = g(i-1) + g(i-2) + x; */
   if( !x_set_)
   {
      mexErrMsgTxt( "variable x undefined, line 4" );
   }
   mccSetRealVectorElement(&g, i, (((mccGetRealVectorElement(&g,
           (i-1))) + (mccGetRealVectorElement(&g, (i-2)))) + x));
   /* end */
}
mccReturnFirstValue(&plhs_[0], &g);
```

Export Output Arguments. Each generated MEX-file must export its output variables into a form that the MATLAB interpreter understands. Exporting an output variable entails:

- Converting each output variable to the standard `mxArray` type. If an output variable is an `int` or a `double`, the MEX-file must convert the variable to an `mxArray`. If the output variable is an `mxArray` variable, the source code must still massage several fields in the `mxArray` structure because the MATLAB Compiler uses some of the fields of the `mxArray` in a nonstandard way.
- Copying each output variable to the appropriate fields of the `plhs` array.

To accomplish both objectives, generated MEX-files rely on a family of *export routines* from the MATLAB Compiler Library. All export routines have names that begin with `mccReturn`. For example, `mccReturnFirstValue` returns the value of variable `g`:

```
mccReturnFirstValue(&plhs_[0], &g);
```

Note In subsequent versions of the MATLAB Compiler, `mcc` routines may change or may disappear from the library. Avoid hand modifying these routines; let the MATLAB Compiler generate `mcc` calls. If you want a program to behave differently, modify the M-file and recompile.

Stand-Alone C Source Code Generated by mcc -e

The –e option flag causes the MATLAB Compiler to generate C code for stand-alone applications. This section explains the code. Most C source code generated by the `mcc –e` command shares this common format:

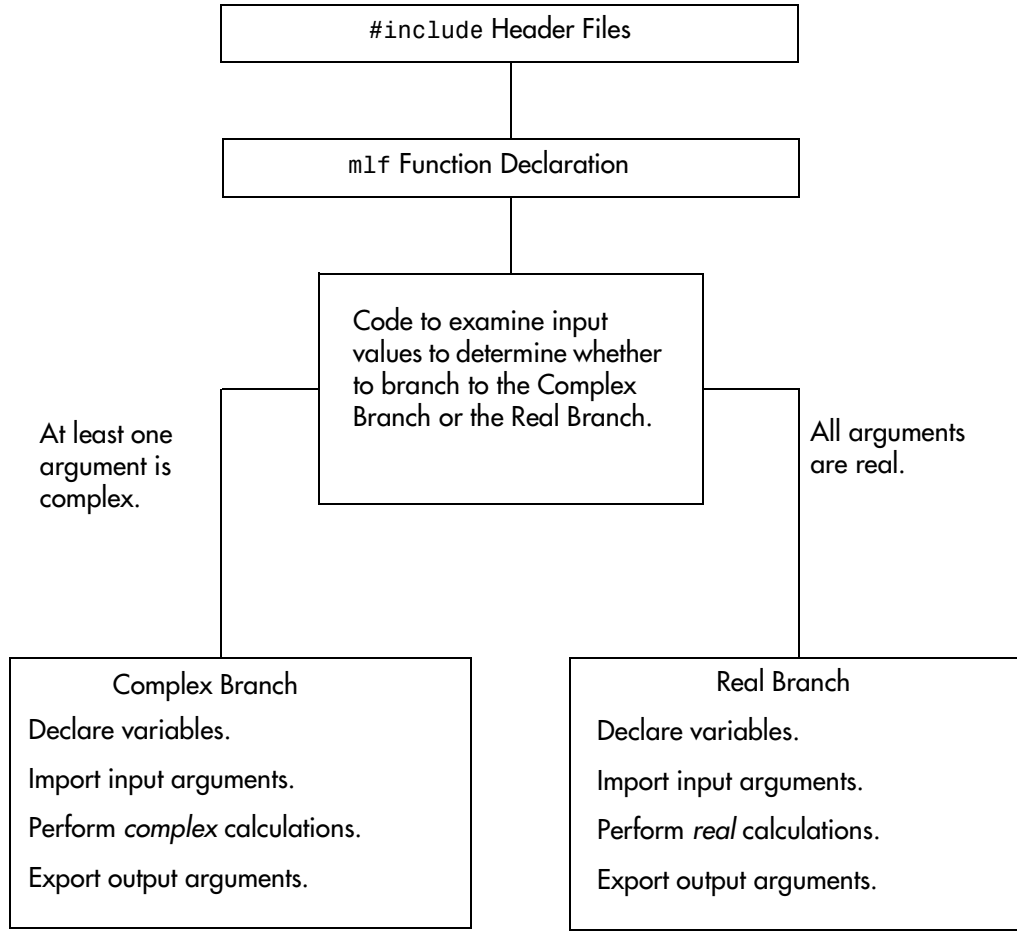


Figure D-2: Structure of Stand-Alone C Source Code Generated by Compiler

Header Files

The MATLAB Compiler generates these `#include` preprocessor statements:

```
#include <math.h>
#include "matrix.h"
#include "mcc.h"
#include "matlab.h"
```

This table lists the included header files.

Header File	Contains Declarations and Prototypes
`math.h`	ANSI C Math Library
`matrix.h`	Matrix Access Routines
`mcc.h`	MATLAB Compiler Library
`matlab.h`	MATLAB C Math Library

mlf Function Declaration

This section explains the function that the MATLAB Compiler generates when you specify the `–e` option flag. Note that the generated function is completely different than the MEX-file gateway function.

Name of Generated Function

The MATLAB Compiler assigns the name of a C function by placing the `mlf` prefix before the M-file function name. For example, compiling a function M-file named `squibo` produces a C function named `mlfSquibo`.

The MATLAB Compiler ignores the case of the letters in the input M-file function name. The output function name capitalizes the first letter and puts all remaining letters in lower case. For example, compiling an M-file function named `sQuIBo` produces a C function named `mlfSquibo`.

If the input M-file function is named `main`, then the MATLAB Compiler names the C function `main`, rather than `mlfMain`. Using the `–m` option flag also forces the MATLAB Compiler to name the C function `main`. (See the description of the `mcc` (Compiler 1.2) reference page in Chapter 6 for further details.)

Output Arguments. If an M-file function does not specify any output parameters, then the MATLAB Compiler generates a C function prototype having a return type of `mxArray *`. Upon completion, the generated C function passes back a null pointer to its caller.

If an M-file function defines only one output parameter, then the MATLAB Compiler maps that output parameter to the return parameter of the generated C function. For example, consider an M-file function that returns one output parameter:

```
function rate = unicycle();
```

The MATLAB Compiler maps the output parameter `rate` to the return parameter of C routine `mlfUnicycle`:

```
mxArray *mlfUnicycle( )
```

The return parameter always has the type `mxArray *`.

If an M-file function defines more than one output parameter, then the MATLAB Compiler still maps the first output parameter to the return parameter of the generated C function. The MATLAB Compiler maps subsequent M-file output parameters to C input/output arguments. For example, the M-file function prototype for `tricycle`

```
function [rate, price, weight]  = tricycle()
```

maps to this C function prototype:

```
mxArray *
mlfTricycle(mxArray **price_lhs_, mxArray **weight_lhs_)
```

The MATLAB Compiler generates C functions so that every input/output argument:

- Has the same type, namely, `mxArray **`
- Has a name ending with `_lhs_`

Input Arguments. The MATLAB Compiler generates one input argument in the C function prototype for each input argument in the M-file.

Consider an M-file function named `bicycle` containing two input arguments:

```
function rate = bicycle(speeds, gears)
```

The MATLAB Compiler translates `bicycle` into a C function named `mlfBicycle` whose function prototype is:

```
mxArray *
mlfBicycle(mxArray *speeds_rhs_, mxArray *gears_rhs_)
```

Every input argument in the C function prototype has the same data type, which is:

```
mxArray *
```

Notice that the MATLAB Compiler gives every input argument the `_rhs_` suffix.

Functions Containing Input and Output Arguments. Given a function containing both input and output arguments, for example,

```
function [g, h] = canus(a, b);
```

the resulting C function, `mlfCanus`, has the prototype

```
mxArray *
mlfCanus(mxArray **h_lhs_, mxArray *a_rhs_, mxArray *b_rhs_)
```

The M-file arguments map to the C function prototype as:

- M-file output argument `g` corresponds to the return value of `mlfCanus`.
- M-file output argument `h` corresponds to `h_lhs`.
- M-file input argument `a` corresponds to `a_rhs`.
- M-file input argument `b` corresponds to `b_rhs`.

In the C function prototype, the first input argument follows the last output argument. If there is only one output argument, then the first input argument becomes the first argument to the C function.

The Body of the mlf Routine

The code comprising the body of the generated `mlf` routine is similar to the code comprising the body of a MEX-file function. Like a MEX-file function, the body of the `mlf` routine typically starts by determining whether any arguments contain any complex values. If any arguments do, then the code branches to the Complex Branch. If all arguments are real, the code branches to the Real Branch.

The code within each branch appears in the order:

1. Declares variables.
2. Imports input arguments.
3. Performs calculations.
4. Exports output arguments.

The code for step 3, performing calculations, contains one significant difference from the way MEX-files perform calculations. The code generated by `mcc –e` never calls back to the MATLAB interpreter. For example, consider an M-file containing a call to the `eig` function:

```
eig(m);
```

If you invoke the MATLAB Compiler without the `–e` option flag, the MATLAB Compiler handles the call to `eig` by generating a callback to the MATLAB interpreter:

```
mccCallMATLAB(1, Mplhs_, 1, Mprhs_, "eig", 2);
```

However, if you invoke the MATLAB Compiler with the `–e` option flag, the MATLAB Compiler handles the call to `eig` by generating a call to the `mlfEig` function of the MATLAB C Math Library:

```
mccImport(&a, (mxArray *) mlfEig(0, &(rhs_[0]), 0), 1, 2);
```

Trigonometric Functions

If you compile an M-file that makes calls to trigonometric functions, you may notice that the MATLAB Compiler generates calls to both `mcc` and `mlf` functions. For example, compiling the M-file

```
function a = myarc(x)
a = acos(x);
```

generates code that contains calls to both `mccAcos` and `mlfAcos`. The reason for this is simple: efficiency. `mccAcos` handles real matrices and `mlfAcos` handles complex matrices. Computing the arc cosine of a complex matrix takes more time than computing the arc cosine of a real matrix. When the MATLAB Compiler is able to determine that a matrix is real, it generates a call to

mccAcos. The resulting code runs faster than it would if the MATLAB Compiler only generated calls to mlfAcos.

This behavior is not restricted to acos — all of the inverse trigonometric functions have both mcc and mlf counterparts.

Stand-Alone C++ Code Generated by mcc -p

This section explains the structure of the C++ code generated by the MATLAB Compiler. Unlike the MEX-file code and the stand-alone C code, the stand-alone C++ code is not divided into a complex branch and a real branch. The generated C++ mixes complex and real computations in a single body of code. This makes the code more concise and more readable.

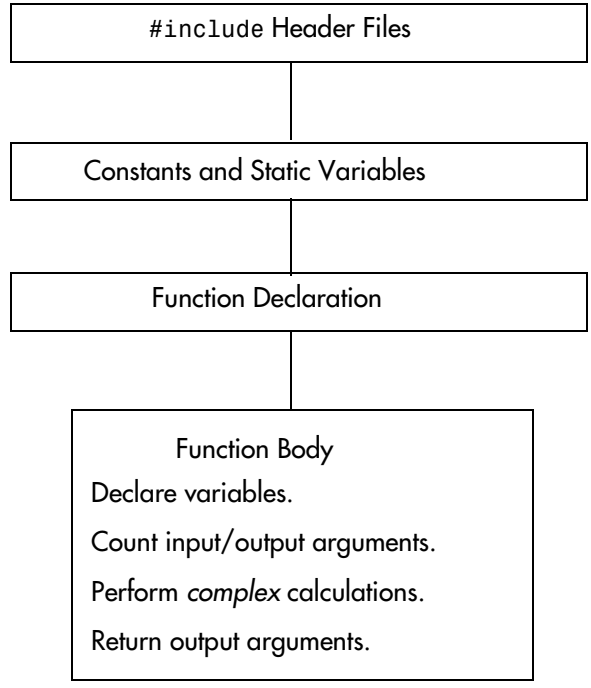


Figure D-3: Structure of Stand-Alone C++ Source Code Generated by Compiler

Header Files

All generated C++ functions include two header files:

```
#include <iostream.h>
#include "matlab.hpp"
```

This table lists the included header files.

Header File	Contains Declarations and Prototypes For
`iostream.h`	C++ input and output streams
`matlab.hpp`	MATLAB C++ Math Library

Constants and Static Variables

The MATLAB Compiler turns every matrix constant, string, or numeric in a MATLAB M-file into two static variables. The first of these variables is an array of doubles; this is the data corresponding to the MATLAB constant. The second variable is always an `mwArray`; it is built using the data in the first variable and is the value used in the generated code. String constants become arrays of ASCII numeric values.

Function Declaration

Name of Generated Function. The MATLAB Compiler uses the name of the M-file function as the name of the generated C++ function. For example, compiling a function M-file named `squibo` produces a C++ function named `squibo`.

When generating C++, the MATLAB Compiler ignores the case of the letters in the input M-file function name. The Compiler uses only lowercase letters in the generated function name. For example, compiling an M-file function named `sQuIBo` produces a C++ function named `squibo`.

If the input M-file function is named `main`, then the MATLAB Compiler names the C++ function `main`. Using the `–m` option flag also forces the MATLAB Compiler to name the C++ function `main`. (See the description of the `mcc` (Compiler 1.2) reference page in Chapter 6 for further details.)

Output Arguments. If an M-file function does not specify any output parameters, then the MATLAB Compiler generates a C++ function prototype having a return type of `void`.

If an M-file function defines only one output parameter, then the MATLAB Compiler maps that output parameter to the return parameter of the generated C++ function. For example, consider an M-file function that returns one output parameter:

```
function rate = unicycle( );
```

The MATLAB Compiler maps the output parameter `rate` to the return parameter of the C++ routine `unicycle`:

```
mwArray unicycle( )
```

The return parameter always has the type `mwArray`.

If an M-file function defines more than one output parameter, then the MATLAB Compiler still maps the first output parameter to the return parameter of the generated C++ function. The MATLAB Compiler maps subsequent M-file output parameters to C++ input/output arguments. For example, the M-file function prototype for `tricycle`

```
function [rate, price, weight]  = tricycle( )
```

maps to this C++ function prototype:

```
mwArray tricycle(mwArray *price, mwArray *weight)
```

The MATLAB Compiler generates C++ functions so that every input/output argument has the same:

- Type, namely, `mwArray *`
- Name as the corresponding MATLAB output argument

Input Arguments. The MATLAB Compiler generates one input argument in the C++ function prototype for each input argument in the M-file.

Consider an M-file function named `bicycle` containing two input arguments:

```
function rate = bicycle(speeds, gears)
```

The MATLAB Compiler translates `bicycle` into a C++ function named `mlfBicycle` whose function prototype is:

```
mwArray bicycle(mwArray speeds, mwArray gears)
```

Every input argument in the C++ function prototype has the same type, which is:

```
mwArray
```

Notice that the C++ input arguments have exactly the same names as the M-file input arguments.

Functions Containing Both Input and Output Arguments. Given a function containing both input and output arguments, for example:

```
function [g, h] = canus(a, b);
```

the resulting C++ function, `canus`, has the prototype:

```
mwArray canus(mwArray *h, mwArray a, mwArray b)
```

The M-file arguments map to the C++ function prototype as:

- M-file output argument `g` corresponds to the return value of `canus`.
- M-file output argument `h` corresponds to C++ input/output argument `h`.
- M-file input argument `a` corresponds to C++ input argument `a`.
- M-file input argument `b` corresponds to C++ input argument `b`.

In the C++ function prototype, the first input argument follows the last output argument. If there is only one output argument, then the first input argument becomes the first argument to the C++ function.

Functions with Optional Arguments. The MATLAB Compiler uses C++ default arguments to handle the optional input and output arguments of a MATLAB M-file function. In C++, arguments with default values need not be specified when the function is called. In C++, default arguments must be given a default value. The MATLAB Compiler uses 0 (zero) for optional output arguments and the special matrix `mwArray::DIN` for optional input arguments. The function declaration specifies which arguments have default values.

In MATLAB, functions with optional arguments use the two special variables `nargin` (number of inputs) and `nargout` (number of outputs) to determine the number of arguments they've been passed. When it compiles a function that uses `nargin` or `nargout`, the MATLAB Compiler generates code to count the number of input and output arguments that don't have the default values. It stores the results in two special variables `_nargin_count` and `_nargout_count`, which correspond exactly to the MATLAB variables `nargin` and `nargout`.

Function Body

In C, the code comprising the body of the generated routine is similar to the code comprising the body of a MEX-file function, with a real and complex branch. However, in C++, there is only one branch, which mixes real and complex calculations as necessary.

The C++ code has this structure:

1. Declares variables.
2. Counts input/output arguments. (Optional)
3. Performs calculations.
4. Returns output arguments.

The code for step 3, performing calculations, contains one significant difference from the way MEX-files perform calculations. The code generated by `mcc –p` never calls back to the MATLAB interpreter. For example, consider an M-file containing a call to the `eig` function:

```
eig(m);
```

If you invoke the MATLAB Compiler without the –p option flag, the MATLAB Compiler handles the call to `eig` by generating a callback to the MATLAB interpreter:

```
mccCallMATLAB(1, Mplhs_, 1, Mprhs_, "eig", 2);
```

However, if you invoke the MATLAB Compiler with the –p option flag, the MATLAB Compiler handles the call to `eig` by generating a call to the `eig` function of the MATLAB C++ Math Library:

```
eig(m);
```

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z