MATLAB®
Compiler

Computation

Visualization

Programming

User's Guide

Version 1.2

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 Revised for Version 1.2

Contents

1 Introducing the MATLAB Compiler

2 Installation and Configuration

3 Getting Started

4 Optimizing Performance

5 Stand-Alone External Applications

6 The Generated Code

7 Directory Organization

8 Reference

9 MATLAB Compiler Library

1

Introducing the MATLAB Compiler

Introduction

This book describes version 1.2 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 external applications. Stand-alone external 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 C/C++ Math Libraries to create stand-alone, external applications that rely on the core math and data analysis capabilities of MATLAB.
- C code S-functions for use with Simulink® and Real-time Workshop®.

Note: Version 1.2 of the MATLAB Compiler is a compatibility release that brings the MATLAB Compiler into compliance with MATLAB 5. Although the Compiler works with MATLAB 5, it does not support many of the new features of MATLAB 5. In addition, code generated with Version 1.2 of the MATLAB Compiler is not guaranteed to be forward compatible.

This chapter takes a closer look at these categories of C and C++ source code, explains the value of compiled code, and finishes by discussing actual examples that use the Compiler.

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

Creating MEX-Files

The MATLAB Compiler (`mcc`) can translate M-files into C MEX-file source code. By default, the MATLAB Compiler then invokes the `mex` utility, which builds the C MEX-file source code into 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.

If you do not have the MATLAB Compiler, you must write the source code for MEX-files "by hand" in either Fortran or C. The *Application Program Interface Guide* explains the fundamentals of this process. To write MEX-files by hand, you have to know how MATLAB represents its supported data types and the MATLAB external interface (i.e., the application program interface).

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.

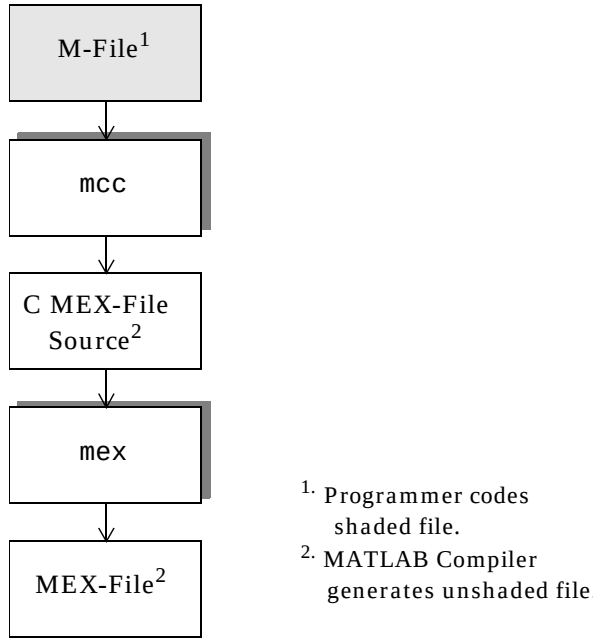


Figure 1-1: Developing MEX-Files

Creating Stand-Alone External Applications

C Stand-Alone External Applications

The MATLAB Compiler, when invoked with the appropriate option flag (`-e` or `-m`), translates input M-files into C source code suitable for your own stand-alone external applications. After compiling this C source code, the resulting object file is linked with the following libraries:

- The MATLAB M-File Math Library, which contains compiled versions of most MATLAB M-file math routines.
- The MATLAB Compiler Library, which contains specialized routines for manipulating certain data structures.
- The MATLAB Math Built-In Library, which contains compiled versions of most MATLAB built-in math routines.
- The MATLAB Application Program Interface Library, which contains the array access routines.
- The MATLAB Utility Library, which contains the utility routines used by various components in the background.
- The ANSI C Math Library.

The first three libraries come with the MATLAB C Math Library product and the next two come with MATLAB. The last library comes with your ANSI C compiler.

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

C++ Stand-Alone External Applications

The MATLAB Compiler, when invoked with the appropriate option flag (`-p`), translates input M-files into C++ source code suitable for your own stand-alone external applications. After compiling this C++ source code, link the resulting object file with the six C object libraries listed above, as well as the MATLAB C++ Math Library, which contains C++ versions of MATLAB functions. Link the MATLAB C++ Math Library first, then the C object libraries listed above.

Creating 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 part of it in C or C++ and part of it as one or more M-files. Figure 1-2 outlines this development process.

See Chapter 5 for complete details regarding stand-alone external applications.

Figure 1-2 illustrates the process of developing a typical stand-alone C external application. Use the same basic process for developing stand-alone C++ external applications, but use the `-p` flag instead of the `-e` flag with the MATLAB Compiler, a C++ compiler instead of a C compiler, and the MATLAB C++ Math Library in addition to the MATLAB C Math Library.

Note: The MATLAB Compiler contains a tool, `mbuild`, which simplifies much of this process. Chapter 5, "Stand-Alone External Applications," describes the `mbuild` tool.

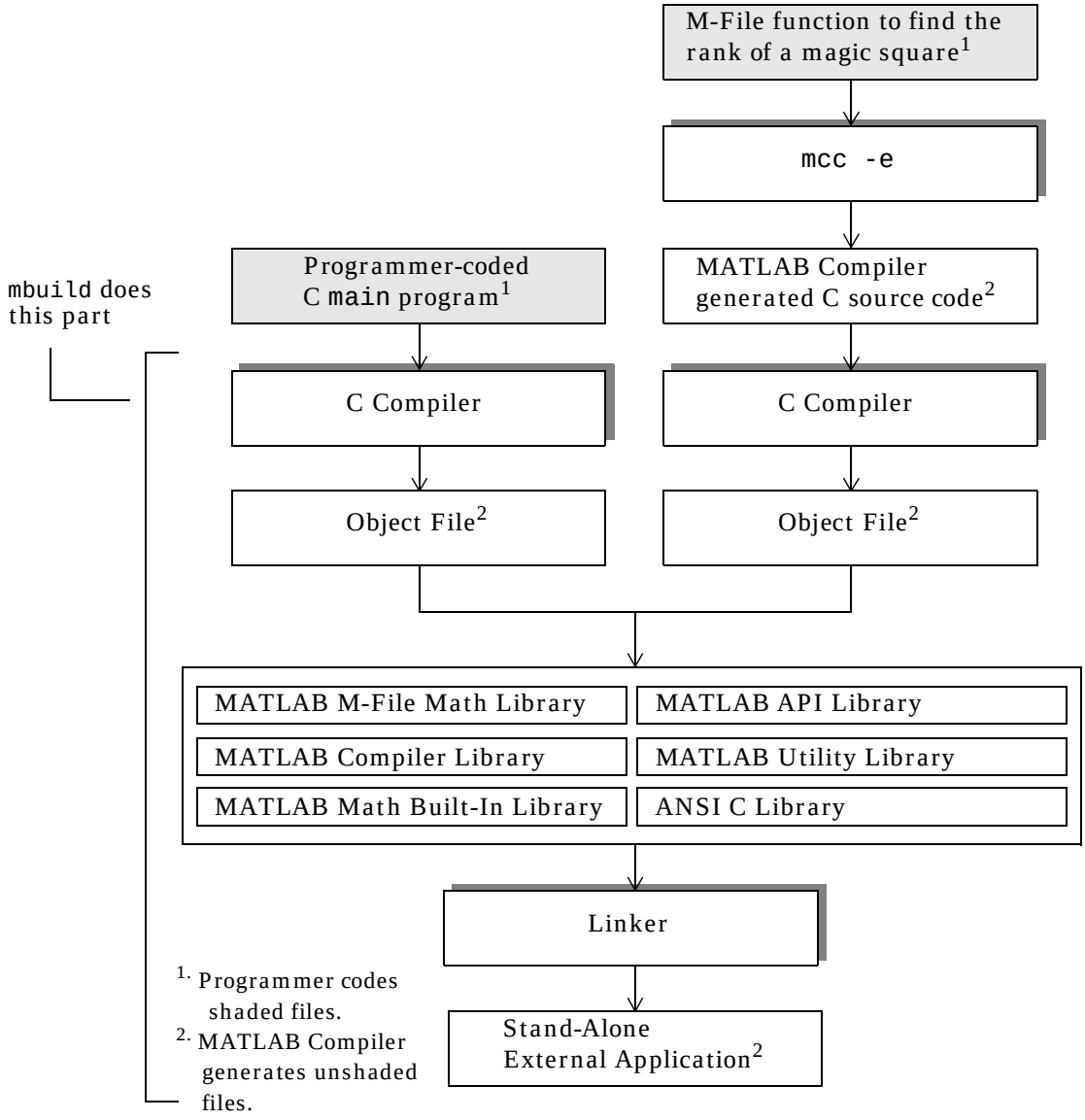
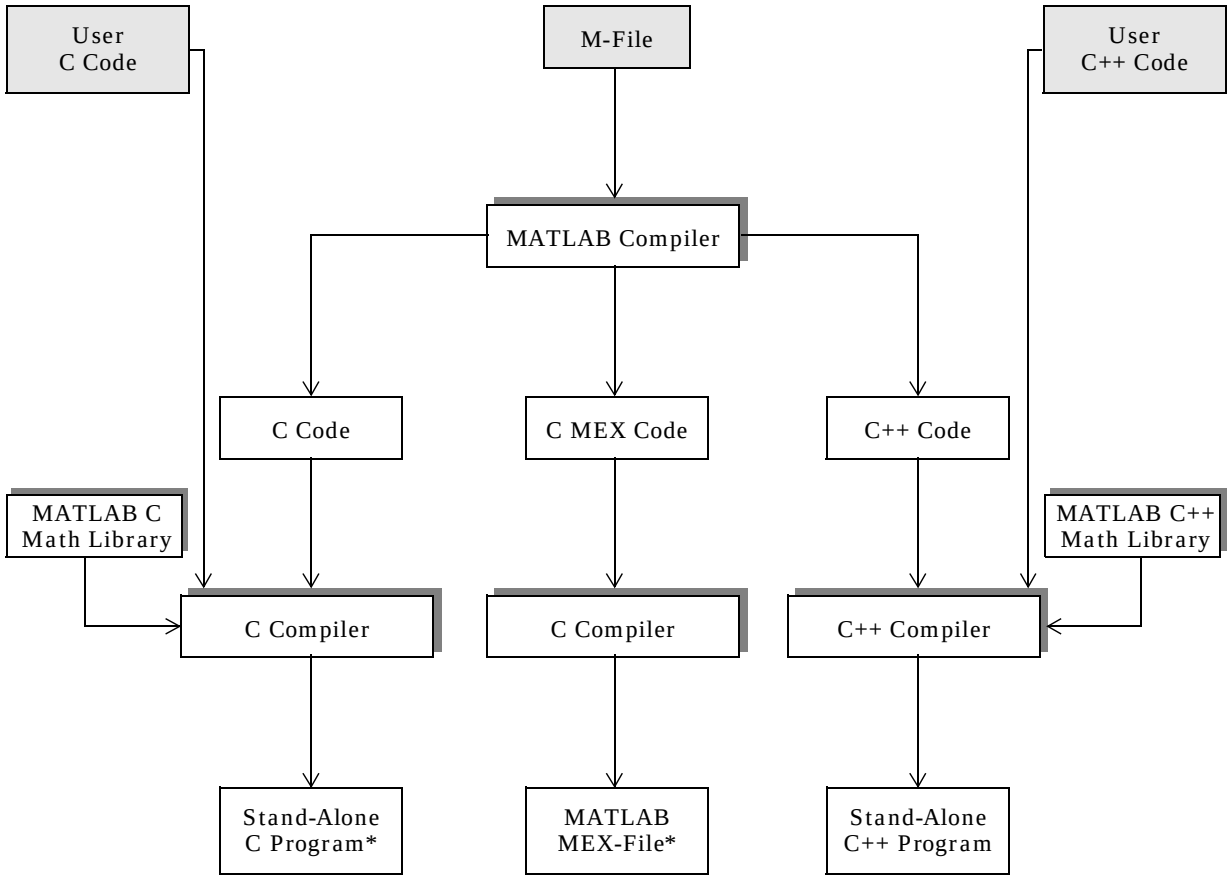


Figure 1-2: Developing a Typical Stand-Alone C External Application

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.



*You can also produce Simulink S-functions (MEX or stand-alone).
See the "Generating Simulink S-Functions" section in Chapter 3 for details.

Figure 1-3: MATLAB Compiler Uses

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 external applications.

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++ 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.
- 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` switch, which controls this, is not available in C++.)
- C or C++ can avoid unnecessary memory allocation overhead that the MATLAB interpreter performs.

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.

Compilation is most likely to speed up M-file functions that:

- Contain loops.
- Contain variables that the MATLAB Compiler views as integer or real scalars.
- Operate on real data only.

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.

2

Installation and Configuration

This chapter explains:

- The hardware and software 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 all three MATLAB Compiler platforms, UNIX, Windows, and Macintosh.

For latest information about the MATLAB Compiler, see the *MATLAB Late-Breaking News*.

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 ANSI compiler vendor.

Unsupported MATLAB Platforms

The MATLAB Compiler and the MATLAB C Math Library support all platforms that MATLAB 5 supports, *except* for:

- VAX/VMS and OpenVMS

The MATLAB C++ Math Library supports all platforms that MATLAB 5 supports, *except* for:

- VAX/VMS and OpenVMS
- Macintosh

Overview

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 the ANSI C/C++ compiler.
3. Configure `mex` to create MEX-files.
4. Verify that `mex` can generate MEX-files.
5. Verify that the MATLAB Compiler can generate MEX-files.

Figure 2-1 shows the sequence on all 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, however, you still must perform the steps given in this chapter first. Chapter 5, "Stand-Alone External Applications," provides the details about the additional installation and configuration steps necessary for creating stand-alone applications.

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

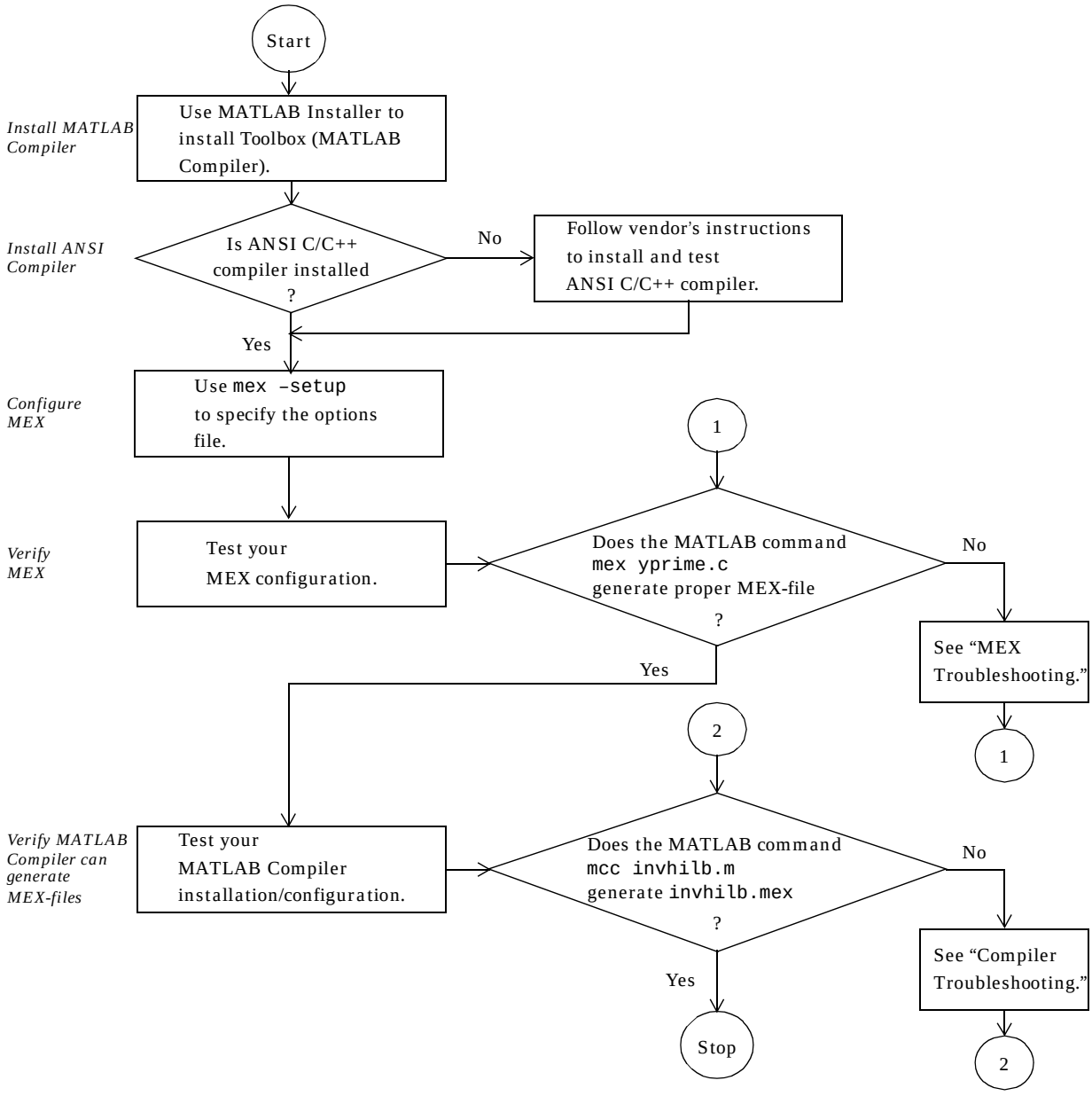


Figure 2-1: MATLAB Compiler Installation Sequence for Creating MEX-Files

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 5.2 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 (less than 2 MB).

MEX-Files

To create MEX-files with the MATLAB Compiler, you must install and configure an ANSI C compiler. The MATLAB Compiler supports the GNU C compiler, gcc, (except on HP and SGI64) and the system's native ANSI compiler.

Note: The C compiler that Sun provides with SunOS 4.1.X is *not* an ANSI C compiler. However, ANSI C compilers for this operating system are available from several vendors.

Stand-Alone C Applications

To create stand-alone C external applications with the MATLAB Compiler, you must install and configure an ANSI C compiler. The MATLAB Compiler supports the GNU C compiler, gcc, (except on HP and SGI64) and the system's native ANSI compiler. You must also install the MATLAB C Math Library, which is separately sold.

Stand-Alone C++ Applications

To create stand-alone C++ external applications, you must install and configure an ANSI C++ compiler. The MATLAB Compiler supports the system's native ANSI compiler on all UNIX platforms and the GNU C++ compiler, g++, on SunOS 4.1.x and Linux. You must also install the MATLAB C++ Math Library, which is a separately sold product.

Installation

MATLAB Compiler

To install the MATLAB Compiler on UNIX systems, follow the instructions in the *MATLAB Installation Guide for UNIX*. The MATLAB Compiler will appear as one of the installation choices that you can select as you proceed through the installation screens.

Before you can install the MATLAB Compiler, you will require an appropriate FEATURE line in your License File. If you do not have the required FEATURE line, contact The MathWorks immediately:

- Via e-mail at `service@mathworks.com`
- Via telephone at 508-647-7000, ask for Customer Service
- Via fax at 508-647-7001

MATLAB Access members can obtain the necessary license data via the Web (`www.mathworks.com`). Click on the MATLAB Access icon and log in to the Access home page. MATLAB Access membership is free of charge.

ANSI 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 ANSI 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 compiler.

Note: On some UNIX platforms, an ANSI 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 an ANSI C/C++ compiler on your system.

Description	Comment
Determine which ANSI compiler is installed on your system.	See your system administrator.
Determine the path to your ANSI compiler.	See your system administrator.
The C compiler that Sun provides with SunOS 4.1.X is *not* an ANSI C compiler.	ANSI C compilers for this operating system are available from several vendors; see your system administrator.

MEX Configuration

To create MEX-files on UNIX, you must 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 2-1.

Table 2-1: MEX-File Extensions for UNIX

Platform	MEX-File Extension
SunOS 4.x	`mex4`
Solaris	`mexsol`
HP 9000 PA-RISC	`mexhp7`
DEC Alpha	`mexaxp`
SGI	`mexsg`
SGI 64	`mexsg64`

Table 2-1: MEX-File Extensions for UNIX (Continued)

Platform	MEX-File Extension
IBM RS/6000	`mexrs6`
Linux	`mexlx`

Specifying an Options File

On UNIX, if you are not using the system native ANSI compiler, you must specify an options file for your compiler. The preconfigured options files that are included with MATLAB are:

Compiler	Options File
System ANSI Compiler	`mexopts.sh`
GCC	`gccopts.sh`
System C++ Compiler	`cxxopts.sh`

Using setup. You can use the `setup` switch to specify the default options file for your C or C++ compiler. Run the `setup` option from the UNIX or MATLAB prompt; it can be called anytime to configure the options file.

```
mex –setup
```

Executing the setup option presents a list of options files currently included in the bin subdirectory of MATLAB:

```
mex –setup

Using the 'mex –setup' command selects an options file that is
placed in ~/matlab and used by default for 'mex' when no other
 options file is specified on the command line.

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/cxxopts.sh :
         Template Options file for building C++ MEXfiles
  2: /matlab/bin/gccopts.sh :
         Template Options file for building gcc MEXfiles
  3: /matlab/bin/mexopts.sh :
         Template Options file for building MEXfiles using the
         native 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, if it does not already exist, in your individual home directory and copies the appropriate options file to the directory. 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.

Changing Compilers. If you want to change compilers, use the `mex -setup` command and make the desired changes.

MEX Verification

C source code for example `yprime.c` is included in the `<matlab>/extern/examples/mex` directory. 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.

Note: The MATLAB Compiler library (`libmccmx.`*`ext`*, where *`ext`* is the extension that corresponds to the specific UNIX platform) is a shared library on all UNIX platforms except Sun4. If you plan to share a compiler-generated MEX-file with another user on any UNIX platform other than Sun4, you must provide the `libmccmx.`*`ext`* library. The user must locate the library along the `LD_LIBRARY_PATH` environment variable.

The values of `.`*`ext`* are: `.a` on IBM RS/6000 and Sun4; `.so` on Solaris, Alpha, Linux, and SGI; and `.sl` on HP 700.

MATLAB Compiler Verification

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 invhilb
```

After displaying the message:

```
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.
```

this command should complete. 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:

```
tic; invhilb(200); toc
```

The `tic` and `toc` commands measure how long a command takes to run: the command should complete in less than a second.

Note that this only tests the compiler's ability to make MEX-files. If you want to create stand-alone applications, refer to Chapter 5, "Stand-Alone External Applications," for additional details.

Microsoft Windows

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

System Requirements

You cannot install the MATLAB Compiler unless MATLAB 5.2 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 (less than
2 MB).

MEX-Files and Stand-Alone C/C++ Applications

To create MEX-files or stand-alone C/C++ applications 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 (DLL) or Windows-NT applications:

- Watcom C version 10.6 or later.
- Borland C++ version 5.0 or later.
- MSVC (Microsoft Visual C++) version 4.2 or later.

To create stand-alone applications, you also need the MATLAB C or C++ Math Libraries, which are sold separately.

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

Installation

MATLAB Compiler

To install the MATLAB Compiler on a PC, follow the instructions in the *MATLAB Installation Guide for PC and Macintosh*. The MATLAB Compiler will appear as one of the installation choices that you can select as you proceed through the installation screens.

Before you can install the MATLAB Compiler, you will require an appropriate FEATURE line in your License File (networked PC users) or an appropriate Personal License Password (non-networked PC users). If you do not have the required FEATURE line or Personal License Password, contact The MathWorks immediately:

- Via e-mail at service@mathworks.com
- Via telephone at 508-647-7000, ask for Customer Service
- Via fax at 508-647-7001

MATLAB Access members can obtain the necessary license data via the Web (www.mathworks.com). Click on the MATLAB Access icon and log in to the Access home page. MATLAB Access membership is free of charge.

ANSI Compiler

To install your C/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 DGB files	For the purposes of the MATLAB Compiler, it is not necessary to install DBG (debugger) files. However, you may need them for other purposes.
MFC	Microsoft Foundation Classes (MFC) are not required.

Description	Comment
16-bit DLL/executables	Not required.
ActiveX	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 let it do so.
Recording the root directory of your C/C++ compiler	Record the complete path to where your C/C++ compiler has been installed, for example, `C:\msdev`. You will need to enter the path when you configure MEX in the next section.

MEX Configuration

To create MEX-files on Windows 95 or NT machines, you must first specify an options file that corresponds to your C/C++ compiler.

Specifying an Options File

The preconfigured options files that are included with MATLAB are:

Compiler	Options File
Microsoft C/C++, Version 4.2 Microsoft C/C++, Version 5.0	`msvcopts.bat` `msvc50opts.bat`
Watcom C/C++, Version 10.6 Watcom C/C++, Version 11.0	`watcopts.bat` `wat11copts.bat`
Borland C++, Version 5.0	`bccopts.bat`

Using setup. You can use the `setup` switch to configure the default options file for your system C/C++ compiler. You can run the `setup` option from either the MATLAB or DOS command prompt; it can be called anytime to configure the options file.

Executing the `setup` option presents a list of compilers whose options files are currently shipped in the `bin` subdirectory of MATLAB. This example shows how to select the Microsoft Visual C++ compiler:

```
Welcome to the utility for setting up compilers
for building external interface files.


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

Fortran compilers
[4] Microsoft PowerStation (version 4.0)
[5] DIGITAL Visual Fortran (version 5.0)

[0] None

compiler: 2
```

If the selected compiler has more than one options file (due to more than one version of the compiler), you are asked for a specific version. For example,

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

version: 1
```

You are then asked to enter the root directory of your compiler installation:

```
Please enter the location of your C/C++ compiler: [c:\msdev]
```

Note: Some compilers create a directory tree under their root directory when you install them. You must respond to this prompt with the root directory only. For example, if the compiler creates directories `bin`, `lib`, and `include` under `c:\msdev`, you should enter only the root directory, which is `c:\msdev`.

Finally, you are asked to verify your choices.

```
Please verify your choices:

Compiler: Microsoft Visual C++ 4.2
Location: c:\msdev
Are these correct?([y]/n): y

Default options file is being updated...
```

Changing Compilers. If you want to change compilers, use the `mex -setup` command and select the desired compiler.

MEX Verification

There is C source code for an example, `yprime.c` included in the `<matlab>\extern\examples\mex` directory, where `<matlab>` represents the top-level directory where MATLAB is installed on your system. 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 MEX-file called `yprime.dll`. MEX-files created on Windows 95 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.

Note: The MATLAB Compiler Library is shipped in DLL format (`libmccmx.dll`). If you plan to share a compiler-generated MEX-file with another user, you must provide the `libmccmx.dll` library. The user must locate the library in the `<matlab>\bin` directory.

MATLAB Compiler Verification

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 invhilb
```

After displaying the message:

```
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.
```

this command should complete. 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:

```
tic; invhilb(200); toc
```

The `tic` and `toc` commands measure how long a command takes to run: the command should complete in less than a second.

Note that this only tests the compiler's ability to make MEX-files. If you want to create stand-alone applications, refer to Chapter 5, "Stand-Alone External Applications," for additional details.

Macintosh

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

System Requirements

You cannot install the MATLAB Compiler unless MATLAB 5.2 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 (less than 2 MB).

MEX-Files

To create MEX-files with the MATLAB Compiler, you must install and configure a supported C/C++ compiler. The supported C/C++ compilers are:

Power Macintosh

- Metrowerks CodeWarrior C/C++ Pro Compiler (Version 12).
- Metrowerks CodeWarrior C/C++ Compiler (Versions 10 & 11).
- MrC Compiler; the MrC compiler comes with the MPW development environment (ETO 21, 22, & 23).

68K Macintosh

- Metrowerks CodeWarrior C/C++ Compiler (Versions 10 & 11).

Stand-Alone C Applications

To create stand-alone C external applications, you must install one of the supported compilers:

- Metrowerks CodeWarrior C/C++ Pro Compiler for Power Macintosh (Version 12).
- Metrowerks CodeWarrior C/C++ Compiler for Power Macintosh or 68K Macintosh systems (Versions 10 & 11).
- MrC Compiler for Power Macintosh (ETO 21, 22, & 23); this compiler comes with the MPW development environment.

You must also install the MATLAB C Math Library, which is sold separately.

Installation

MATLAB Compiler

To install the MATLAB Compiler on a Macintosh, follow the instructions in the *MATLAB Installation Guide for PC and Macintosh*. The MATLAB Compiler will appear as one of the installation choices that you can select as you proceed through the installation screens.

Before you can install the MATLAB Compiler, you will require an appropriate Personal License Password. If you do not have the required Personal License Password, contact The MathWorks immediately:

- Via e-mail at `service@mathworks.com`
- Via telephone at 508-647-7000, ask for Customer Service
- Via fax at 508-647-7001

MATLAB Access members can obtain the necessary license data via the Web (`www.mathworks.com`). Click on the MATLAB Access icon and log in to the Access home page. MATLAB Access membership is free of charge.

ANSI Compiler

To install your C/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. Typically, the compiler vendor provides some test procedures. The following section, "Things to Be Aware of," contains

Macintosh-specific details regarding the installation and configuration of your ANSI 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.

MEX Configuration

Before you can create MEX-files on Macintosh systems, you must perform several configuration steps.

Specifying an Options File

MATLAB includes preconfigured options files; you must configure your system to use the one that corresponds to your C/C++ compiler. The included options files are:

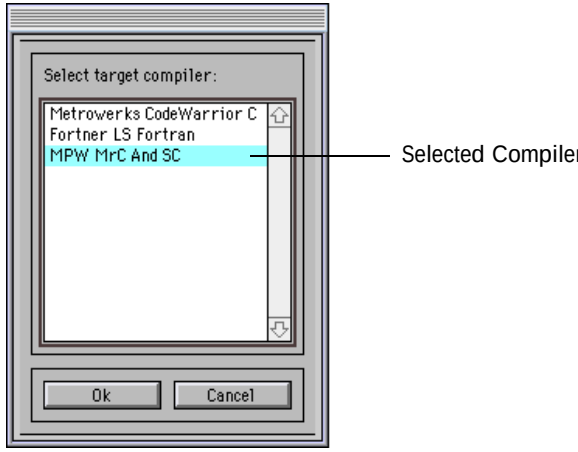
Compiler	Options File
Metrowerks CodeWarrior C/C++	`mexopts.CW`
Metrowerks CodeWarrior C/C++ Pro	`mexopts.CWPRO`
MPW MrC	`mexopts.MPWC`

Using setup. You can use the `setup` switch to specify the default options file for your system C compiler. It can be run at anytime to configure the options file.

Run the `setup` option from the MATLAB prompt.

```
mex –setup
```

Executing `setup` displays a dialog with a list of compilers whose options files are currently shipped in the `<matlab>:extern:scripts:` folder. (Your dialog may differ from this one.) This figure shows MPW MrC selected as the desired compiler.



Click **Ok** to select the compiler. If you previously selected an options file, you are asked if you want to overwrite it. If you do not have an options file in your `<matlab>:extern:scripts:` folder, `setup` creates the appropriate options file for you.

Note: If you select MPW, `setup` asks you if you want to create `UserStartup•MATLAB_MEX` and `UserStartupTS•MATLAB_MEX`, which configure MPW and ToolServer for building MEX-files.

Changing Compilers. To change the C compiler used by the `mex` script, run `mex -setup` and select the desired compiler.

MEX Verification

There is C source code for an example, `yprime.c` included in the `<matlab>:extern:examples:mex:` folder. 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 MEX-file called `yprime.mex`. MEX-files created on Macintosh systems always have the extension `mex`.

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.

Notes: If you plan to share a compiler-generated MEX-file with another user, you must provide the MATLAB Compiler library (`libmccmx`) for Power Macintosh users. No additional files are necessary for 68K Macintosh users.

For Power Macintosh shared libraries such as `libmccmx` to be found by the Macintosh operating system at runtime, the shared libraries need to appear in either:

- The `System Folder:Extensions:` folder, or
- The same folder as the application that uses the shared libraries.

The MATLAB installer automatically puts an alias to `<matlab>:extern:lib:PowerMac:` (where the shared libraries are stored) in the `System Folder:Extensions:` folder and names the alias `MATLAB Shared Libraries`.

MATLAB Compiler Verification

Testing ToolServer

(*MPW Users*) To compile MATLAB Compiler-generated MEX-functions automatically, the `mcc` function uses the MPW ToolServer. Therefore, to use the auto-compile features of `mcc`, ToolServer must be available. To test if the MPW ToolServer is accessible from MATLAB, type

```
!toolserver echo "Pass"
```

at the command prompt. The MPW ToolServer utility should start up, and the word `Pass` should display in the MATLAB Command Window. For more information on installing ToolServer, see the documentation included with ToolServer.

Note: If you have more than one copy of ToolServer installed on your system, you should manually launch the correct ToolServer prior to Compiler use.

Testing the MATLAB Compiler

Assuming that 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 invhilb
```

After displaying the message:

```
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.
```

this command should complete. 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.mex`. Finally, at the MATLAB prompt, type:

```
tic; invhilb(200); toc
```

The `tic` and `toc` commands measure how long a command takes to run: the command should complete in less than a second.

Note that this only tests the compiler's ability to make MEX-files. If you want to create stand-alone applications, refer to Chapter 5, "Installation and Configuration," for additional details.

Special Considerations

The first time you run the `mex` script, dialogs may appear that ask you to find and select either the CodeWarrior IDE application or the ToolServer application. This information is saved in the `<matlab>:extern:scripts:` folder. Be sure you have write privileges enabled for that folder.

Special Considerations for CodeWarrior 10 and 11 Users

There are several cases when CodeWarrior users may have to perform some additional steps to use the `mex` script.

Note: The file, `PPCstationery_pro.proj`, works with CodeWarrior 12 (CodeWarrior Pro) without modification.

Updating Project. While using the `mex` script with CodeWarrior on a Power Macintosh, you may get a warning dialog that reads:

```
This project was created by an older version of CodeWarrior. Do
you wish to update it?
```

To update, do the following:

1. Click on the **Cancel** button to dismiss the dialog.
2. From the Finder, select the file `<matlab>:extern:src:PPCstationery.proj`.
3. Choose **Get Info** from the **File** menu.

4 Uncheck **Stationery pad** in the **PPCstationery.proj Info** window.

5 Switch applications to CodeWarrior.

6 From CodeWarrior, open the `PPCstationery.proj` file using **Open** from the **File** menu.

7 When the `Do you wish to update it?` dialog appears, click **OK**.

8 Close the project by selecting **Close** from the **File** menu.

9 Switch back to the Finder.

10 Again, select the `PPCstationery.proj` file from the Finder and choose **Get Info** from the **File** menu.

11 Recheck the **Stationery pad** check box.

12 Close the **PPCstationery.proj Info** window by selecting **Close Window** from the **File** menu.

If you get the same warning dialog on a 68K Macintosh, repeat steps 2 through 12 using the file `<matlab>:extern:src:68Kstationery.proj`.

You will now be able to use the `mex` script without getting the warning dialog shown above.

Access Path Message. The CodeWarrior project file, `PPCstationery.proj`, included with MATLAB 5.2 was built with CodeWarrior 10. If you get a message that says:

```
The following access path cannot be found
<CodeWarrior>:Metrowerks CodeWarrior:(Project
Stationery):Project Stationery Support:
```

you must edit your project settings.

1 Choose **Project Settings** from the **Edit** menu.

2 Remove the line
`{compiler ƒ}:(Project Stationery):Project Stationery Support:`

3 Click **OK**.

Special Considerations for MPW

You must install the ToolServer application (included with MPW) in your MPW folder. For more information on installing ToolServer, see the documentation included with ToolServer.

Troubleshooting

This section identifies some of the more common problems that may occur when installing and configuring the MATLAB Compiler. The "MEX Troubleshooting" section focuses on problems creating MEX-files; the "Compiler Troubleshooting" section focuses on problems involving the MATLAB Compiler.

MEX Troubleshooting

Non-ANSI Compiler on UNIX

A common configuration problem in creating C MEX-files on UNIX involves using a non-ANSI C/C++ compiler. You must use an ANSI C/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 a problem with either the MATLAB Compiler or improper use of the `-i` option. If the problem is with the MATLAB Compiler, you should send the pertinent information via e-mail to `bugs@mathworks.com`. For more information on the `-i` option, see "Optimizing with the -i Option" in Chapter 4.

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 `bugs@mathworks.com`.

MEX Works from Shell But Not from MATLAB (UNIX)

If the command

```
mex 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`.

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.

Compiler Troubleshooting

Stack Overflow on Macintosh

If, while converting large M-files to C code, you experience crashes of a seemingly random nature (often a System Error #28 or #11, but not always), the problem may be that the MATLAB stack is overflowing. To increase the size of the MATLAB stack, follow these steps:

1. Make a copy of the MATLAB binary (referred to as "`MATLAB copy`" below).
2. Open the copy using ResEdit, a resource editor available from Apple and packaged with most Macintosh C compilers.
3. Double-click on the STR icon in the ResEdit window titled "`MATLAB copy`".
4. Find the STR resource named "`PowerMacStackSize`" and double-click on it.
5. The field labeled "`The String`" contains a number that represents the number of kilobytes of stack space reserved by MATLAB. The default value is 128 on the Power Macintosh. Enter a higher number in this field (exactly

how high depends on the size of your M-file, but we recommend a minimum of 512K).

6 Quit ResEdit. When ResEdit asks if you want to save "MATLAB copy" before closing, choose **Yes**.

Double-click on the "MATLAB copy" icon to start MATLAB. Convert the M-file to C code by running `mcc` as before. You may need to repeat the previous steps, experimenting with stack size until you find a value high enough to allow compiling of large M-files, but small enough that the stack doesn't significantly reduce MATLAB's free memory (i.e., heap space). When you are satisfied with your stack size setting, you can delete the original MATLAB and rename "MATLAB copy" to "MATLAB".

MATLAB Compiler Cannot Generate MEX-File

If the previous solution does not address your difficulty with the MATLAB Compiler, contact Technical Support at The MathWorks at `support@mathworks.com` or 508 647-7000.

3

Getting Started

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.
- Time the performance of M-files and MEX-files.
- Invoke MEX-files.
- Generate Simulink S-functions.

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

A Simple Example

Consider a simple M-file function called `squibo.m`.

```
function g = squibo(n)
% This function calculates the first n "squibonacci" numbers.
% $Revision: 1.1 $
%
g = zeros(1,n);
g(1)=1;
g(2)=1;

for i=3:n
   g(i) = sqrt(g(i–1)) + g(i–2);
end
```

The name `squibo` is an amalgam of square root and Fibonacci. The traditional Fibonacci sequence grows too rapidly for our purposes.

`squibo.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. However, the MATLAB Compiler generates loop code that runs very quickly.

Invoking the M-File

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

Note: The timings listed in this book were recorded on a Pentium Pro 200 MHz PC running Linux. 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.

```
tic; for i = 1:10; squibo(10000); end; toc
elapsed_time =
     5.7446
```

On the Pentium Pro 200, the M-file took about six seconds of CPU time to calculate the first 10,000 “squibonacci” numbers ten times.

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 squibo
```

This `mcc` command generates:

- A file named `squibo.c` containing MEX-file C source code.
- A MEX-file named `squibo.mex`. (The actual filename extension of the executable MEX-file varies depending on your platform.)

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

Invoking the MEX-File

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

```
squibo(10000);
```

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

Timing the MEX-file on our machine produces:

```
tic; for i = 1:10; squibo(10000); end; toc
elapsed_time =
    3.7947
```

The MEX-file runs about 33% faster than the M-file version. To get better results, you typically have to give the MATLAB Compiler some hints on how it can optimize the code. The MATLAB Compiler provides many ways to optimize code including option flags, assertions, pragmas, and special optimization functions. Optimization is such a rich topic that Chapter 4 is devoted to it.

Optimizing

As an example of optimization, consider what happens when you specify the `–r` (all arguments are real; no complex data) and `–i` (suppress array boundary checking) MATLAB Compiler option flags:

```
mcc –ri squibo
tic; for i = 1:10; squibo(10000); end; toc
elapsed_time =
    0.0625
```

Now the optimized version runs about 100 times faster than the original (unoptimized) M-file. Does this imply that all optimized MEX-files run 100 times faster than their corresponding M-files? No, absolutely not. You only see significant performance gains in certain situations, typically in those M-files that contain loops. Many MEX-files run no faster than their M-file counterparts. On the other hand, some M-files that contain nested loops may experience performance gains greater than a factor of 100.

Generating Simulink S-Functions

You can use the MATLAB Compiler to generate Simulink C language S-functions. This allows you to speed up Simulink models that contain MATLAB M-code that is referenced from a MATLAB Fcn block. Although using the C code in real-time implementations is not recommended, this capability of the MATLAB Compiler allows subsequent code generation through Real-Time Workshop, thus speeding up your entire Simulink model.

For more information about Simulink, see the *Using Simulink* manual; in particular, see the chapter on S-functions. For more information about Real-Time Workshop, see the *Real-Time Workshop User's Guide*.

Simulink-Specific Options

By using Simulink-specific options with the MATLAB Compiler, you can generate a complete S-function that is compatible with the Simulink 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`). Do not confuse it with the lowercase `-s` option that translates MATLAB `global` variables to `static` C (local) variables.

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

`mfilename.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.

Real-Time Applications

Code generated by the MATLAB Compiler is not necessarily suitable for real-time applications. In real-time applications, it is generally desirable for code to be very fast and efficient. The Real-Time Workshop for Simulink is designed to produce such code. The MATLAB Compiler on the other hand, is designed to handle many different aspects of the MATLAB language. MATLAB is a very flexible environment that allows many different capabilities within its M-code language. The MATLAB Compiler must understand all of these nuances in the language when converting M-code to C code. These factors contribute to making it difficult to create efficient real-time executable code from a MATLAB M-file. Therefore, we do not recommend that you use the S-function output from the MATLAB Compiler in real-time applications.

Using -e Option

If you are simply interested in speeding up your Simulink diagrams that contain MATLAB code referenced in a MATLAB Fcn block, the capability described above may help. If you plan to generate code for a nonreal-time simulation with Real-Time Workshop and include S-functions generated by the

MATLAB Compiler, you must create the S-functions with the Compiler's `-e` option. The `-e` option lets you generate C code for stand-alone external applications.

```
mcc -S -e mfilename
```

After you generate your code with Real-Time Workshop, you must ensure that you have the MATLAB C Math Library installed on whatever platform you want to run the generated code.

Note: The MATLAB Compiler `-S` option does *not* support the passing of parameters that is normally available with Simulink S-functions.

Specifying S-Function Characteristics

Sample Time

Similar to the MATLAB Fcn block, the automatically generated S-function has an inherited sample time. To specify a different sample time for the generated code, edit the C code file after the MATLAB Compiler generates it and set the sample time through the `ssSetSampleTime` function. See the *Using Simulink* manual for a description of the sample time settings.

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. You can use the MATLAB Compiler assertions `mbrealvector`, `mbrealscalar`, and `mbreal` to guarantee that the resulting C code uses the correct data types. For more information on assertions, see "Optimizing Through Assertions" in Chapter 4.

Limitations and Restrictions

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. (See page 3-12 for further details.)
- 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`, `-v4`, 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.

Calls to mlf Functions. The MATLAB Compiler cannot generate calls to `mlf` functions when creating a MEX-file. Support for that capability would require that *all* MATLAB Compiler users also have the optional MATLAB C or C++ Math Library installed; since some users do not have either of those optional libraries installed, this restriction applies to all MATLAB Compiler users.

MATLAB Compiler-compatible M-Files. Since the Compiler cannot handle return results from functions that are V5 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 produces the same results is included with the MATLAB Compiler. These M-files must be on the path in MEX mode because the Compiler will generate calls to them and the V5 versions will return cell arrays causing a runtime error.

The directories containing the MATLAB Compiler-compatible M-files are:

- On UNIX, `<matlab>/extern/src/math/tbxsrc`
- On Windows, `<matlab>\extern\src\math\tbxsrc`
- On Macintosh, `<matlab>:extern:src:math:tbxsrc:`

This stipulation also applies to the return structure of `polyval` used as input to `polyfit`.

Differences Between the MATLAB Compiler and Interpreter

In addition, there are several circumstances where the MATLAB Compiler's behavior 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 External Applications

The restrictions and limitations noted in the previous section also apply to stand-alone external applications. In addition, stand-alone external 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 *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.

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 global in the base workspace; you can access these variables from the MATLAB interpreter.

The MATLAB Compiler can only compile function M-files. That is, the MATLAB Compiler cannot compile script M-files. Furthermore, the MATLAB Compiler cannot 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(2); % Assign 2x2 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(2); % Assign 2x2 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

Optimizing Performance

This chapter explains how to improve the performance of code generated by the MATLAB Compiler.

You can optimize performance for C code by

- Supplying appropriate MATLAB Compiler options (page 4-5).
- Avoiding callbacks to MATLAB (page 4-20).
- Preallocating matrices in your M-file (page 4-27).

You can optimize performance for C *and* C++ code by

- Type imputation (page 4-3).
- Specifying assertions (page 4-13).
- Specifying pragmas (page 4-17).
- Avoiding complex calculations (page 4-18).
- Vectorizing your M-file (page 4-29).

The *MATLAB C++ Math Library User's Guide* provides additional information about how to optimize C++ applications that use the MATLAB C++ Math Library.

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.

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 4-1 shows the results of the `squibo` example using `n` equal to 10,000 and executing it 10 times in a loop.

Table 4-1: Performance for n=10000, run 10 times

Compile Command Line	Elapsed Time (in sec.)	% Improvement
`squibo.m` (uncompiled)	5.7446	--
`mcc squibo`	3.7947	33.94
`mcc -r squibo`	0.4548	92.08
`mcc -i squibo`	2.7815	51.58
`mcc -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 ***************
 *
 *       C0_            complex scalar temporary
 *       I0_            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 `I0_`) 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 (I0_ = 3; I0_ <= n; I0_ = I0_ + 1)
      {
         i = I0_;
         /* 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 ****************
 *
 *       I0_             integer scalar temporary
 *       R0_             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

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

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 ****************
 *
 *       I0_             integer scalar temporary
 *       R0_             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 (I0_ = 3; I0_ <= n; I0_ = I0_ + 1)
{
   i = I0_;
   /* g(i) = sqrt(g(i-1)) + g(i-2); */
   R0_ = sqrt((mccPR(&g)[((i-1)-1)]));
   mccPR(&g)[(i-1)] = (R0_ + (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.
`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 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 -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. See the `%#ivdep` reference page in Chapter 8 for complete details.

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.

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`.

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 –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. (See Chapter 9 for more details on the MATLAB Compiler Library.)

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 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 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 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 –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 minimize_it fmins
```

instead of

```
mcc –h minimize_it
```

Note: Due to Compiler restrictions, some of the V5 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/math/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 –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 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 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 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 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. The execution times for the interpreted and compiled versions of the two M-files are:

Table 4-2: 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);
```

The execution times for the interpreted and the compiled versions of the two M-files are:

Table 4-3: 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.

5

Stand-Alone External Applications

Introduction

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

To *build* stand-alone external C applications as described in this chapter, MATLAB, the MATLAB Compiler, a C compiler, and the MATLAB C Math Library must be installed on your system. To build stand-alone external C++ applications, MATLAB, the MATLAB Compiler, a C++ compiler, and the MATLAB C++ Math Library must be installed on your system. If you have the MATLAB C++ Math Library installed, you can build both C and C++ external applications.

Note: The MATLAB Compiler cannot compile calls to MEX-functions in stand-alone mode.

MEX-files and stand-alone external 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 external C or C++ applications run independently of MATLAB.

The source code for a stand-alone external 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 (–e), translates input M-files into C source code suitable for your own stand-alone

external 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 Compiler Library (`libmcc`), which contains specialized routines for manipulating certain data structures.
- The MATLAB Math Built-In Library (`libmatlb`), which contains compiled versions of most MATLAB built-in math routines.
- The MATLAB Application Program Interface 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 ANSI C Math Library.

The `libmmfile`, `libmcc`, and `libmatlb` libraries come with the MATLAB C Math Library product and the `libmx` and `libut` libraries come with MATLAB. 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 C/C++ Math Library installed, the system will not be able to find the appropriate libraries and the compilation will fail.

The MATLAB Compiler, when invoked with the appropriate option flag (`-p`), translates input M-files into C++ source code suitable for your own stand-alone external 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.

Building Stand-Alone External C/C++ Applications

This section explains how to build C and C++ stand-alone external applications on UNIX, Microsoft Windows, and Macintosh systems.

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

Note: This chapter assumes that you have installed and configured the MATLAB Compiler, and that you can use it to create MEX-files. If this is not the case, follow the instructions in Chapter 2, "Installation and Configuration," so that you can create MEX-files on your system.

Overview

On all three operating systems, the sequence of steps you use to build C and C++ stand-alone external applications is:

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

Figure 5-1 shows the sequence on all platforms. The sections following the flowchart provide more specific details for the individual platforms.

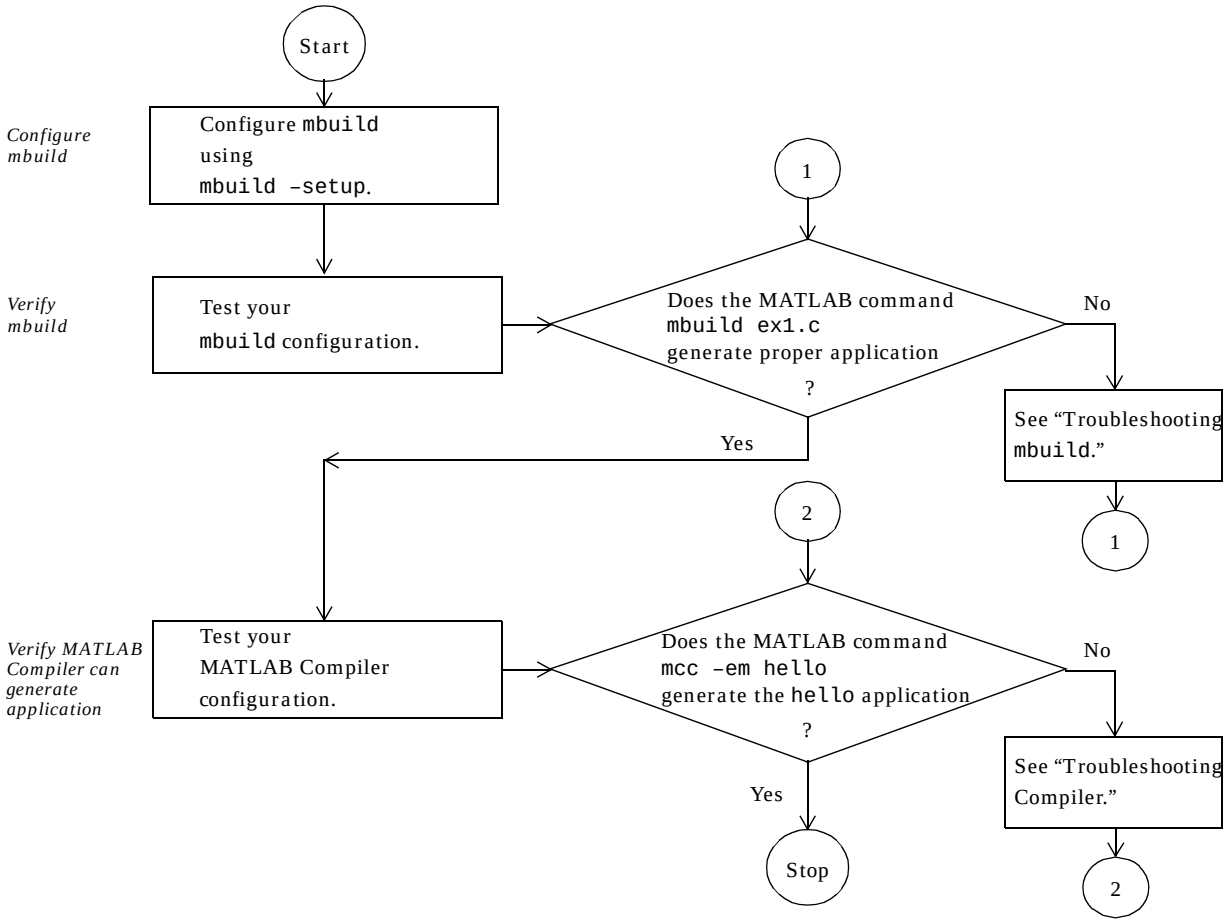


Figure 5-1: Sequence for Creating Stand-Alone C/C++ Applications

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 against. The necessary shared libraries vary by platform and are listed within the individual UNIX, Windows, and Macintosh sections that follow.

Getting Started

Before you can create stand-alone external applications, you must provide your ANSI compiler with the correct set of compiler and linker switches. The set of switches and other parameters are included in the options file for your ANSI compiler. Once you provide this information, `mbuild` is ready to build the application.

Note: Before you can create stand-alone external C applications, you must install the MATLAB C Math Library on your system. Before you can create stand-alone external C++ applications, you must install the MATLAB C++ Math Library on your system. (The MATLAB C++ Math Library includes the MATLAB C Math Library.) The C and C++ Math Libraries are separately sold products available from The MathWorks, Inc.

The MATLAB C++ Math Library makes use of both templates and exceptions. Make sure your C++ compiler supports these C++ language features; if it does not, you will be unable to use the MATLAB C++ Math Library.

Introducing mbuild

The MathWorks provides a utility, `mbuild`, on all platforms that 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 -e` or `mcc -p` invokes `mbuild` only when a `main` is present. The MATLAB Compiler determines that a `main` is present if:

- `-m` is specified on the command line (e.g., `mcc -em` *filename*).
- The file is `main.m`.
- A `.c` file is specified on the command line.

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

Building on UNIX

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

Configuring mbuild

The `mbuild` script provides a convenient and easy way to configure your ANSI compiler with the proper switches to create an application. To configure your compiler, at the UNIX prompt type:

```
mbuild –setup
```

The `setup` switch creates a user-specific options file for your ANSI compiler.

Note: The default C compiler that comes with SunOS 4.1.X workstations is not an ANSI C compiler.

Executing the setup option 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' when no other
options file is specified on the command line.

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

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

Select the proper options file for creating a stand-alone C or C++ application 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 MATLAB directory. If an options file already exists in your MATLAB directory, the system prompts you to overwrite it.

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

Changing Compilers. If you want to change compilers or switch between C and C++, use the `mbuild –setup` command and make the desired changes.

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 `cd` 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/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`, `sol2`, or `sun4`)

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 (except Sun4), the compiler library is shipped as a shared object (.so) file. Any compiler-generated, stand-alone application must be able to locate the C/C++ libraries along the LD_LIBRARY_PATH environment variable 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 –em 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.

The mbuild Script

The mbuild script supports various switches that allow you to customize the building and linking of your code. The only required option that all users must execute is setup; the other options are provided for users who want to customize the process. The mbuild syntax and options are:

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

Table 5-1: 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> as the options file; <file> is a full pathname if it is not in current directory. . (Not necessary if you use the –setup option.)
-F <file>	Use <file> as the options file. (Not necessary if you use the –setup option.) <file> is searched for in the following manner: The file that occurs first in this list is used: • ./<filename> • $HOME/matlab/<filename> • $TMW_ROOT/bin/<filename>
–g	Build an executable with debugging symbols included.

Table 5-1: mbuild Options on UNIX (Continued)

Option	Description
-h[elp]	Help; prints a description of mbuild and the list of options.
-I<pathname>	Include <pathname> in 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.
<name>=<def>	Override options file setting for variable <name>.
-n	No execute flag. This option causes the commands used to compile and link the target to display without executing them.
-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 switch 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 (-g, -v, and -f) are available on the mcc command line and are passed along to mbuild.

Customizing mbuild

If you need to see which switches `mbuild` passes to your compiler and linker, use the verbose switch, `-v`, as in:

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

to generate a list of all the current compiler settings. If you need to change the switches, use an editor to make changes to your options file, which is in your local MATLAB directory. You can also embed the settings obtained from the verbose switch into an integrated development environment (IDE) or makefile that you need to maintain outside of MATLAB.

Note: Any changes made to the local options file will be overwritten if you execute `mbuild -setup`.

Distributing Stand-Alone UNIX 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 against. This package of files includes:

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

where `.`*`ext`* is

`.a` on IBM RS/6000 and Sun4; `.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`, `libmcc.so`, `libmx.so`, `libut.so`, and `libmatpp.so`. Remember to locate the shared libraries along the `LD_LIBRARY_PATH` environment variable so that they can be found and loaded.

Building on Microsoft Windows

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

Shared Libraries

All the libraries (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 external applications are WIN32 Dynamic Link Libraries (DLLs). Before running a stand-alone external application, you must ensure that the directory containing the DLLs is on your path.

Configuring mbuild

The `mbuild` script provides a convenient and easy way to configure your ANSI compiler with the proper switches to create an application. To configure your compiler, use

```
mbuild –setup
```

Run `mbuild` with the `setup` option from either the MATLAB or DOS command prompt. The `setup` switch creates an options file for your ANSI compiler.

You *must* run `mbuild –setup` before you create your first stand-alone application; otherwise, when you try to create a stand-alone application, you will get the message

```
Sorry! No options file was found for mbuild. The mbuild script
must be able to find an options file to define compiler flags and
other settings.  The default options file is
$script_directory\\$OPTFILE_NAME.

To fix this problem, run the following:

mbuild –setup

This will configure the location of your compiler.
```

Executing the setup option presents a list of compilers whose options files are currently included in the bin subdirectory of MATLAB. This example shows how to select the Microsoft Visual C++ compiler:

```
mbuild –setup
Welcome to the utility for setting up compilers
for building math library applications files.

Choose your default Math Library:
[1] MATLAB C Math Library
[2] MATLAB C++ Math Library

Math Library: 1

Choose your C/C++ compiler:
[1] Borland C/C++        (version 5.0)
[2] Microsoft Visual C++ (version 4.2 or version 5.0)
[3] Watcom C/C++         (version 10.6 or version 11.0)

[0] None

compiler: 2
```

If we support more than one version of the compiler, you are asked for a specific version. For example,

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

version: 2
```

Next, you are asked to enter the root directory of your ANSI compiler installation:

```
Please enter the location of your C/C++ compiler: [c:\msdev]
```

Finally, you must verify that the information is correct:

```
Please verify your choices:

Compiler: Microsoft Visual C++ 5.0
Location: c:\msdev
Library: C math library

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

Default options file is being updated...
```

If you respond to the verification question with `n` (no), you get a message stating that no compiler was set during the process. Simply run `mbuild -setup` once again and enter the correct responses for your system.

Changing Compilers. If you want to change your ANSI (system) compiler, make other changes to its options file (e.g., change the compiler's root directory), or switch between C and C++, use the `mbuild -setup` command and make the desired changes.

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 or NT always have the extension exe. The created application is a 32-bit MS-DOS console application.

You can now run your stand-alone application by launching it from 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 –em 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.

The mbuild Script

The `mbuild` script supports various switches that allow you to customize the building and linking of your code. The only required option that all users must

execute is setup; the other options are provided for users who want to customize the process. The mbuild syntax and options are:

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

Table 5-2: mbuild Options on Windows

Option	Description
-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 it is not in current directory. . (Not necessary if you use the -setup option.)
-F <file>	Use <file> as the options file. (Not necessary if you use the -setup option.) <file> is searched for in the current directory first and then in the same directory as mbuild.bat.
-g	Build an executable with debugging symbols included.
-h[elp]	Help; prints a description of mbuild and the list of options.
-I<pathname>	Include <pathname> in the compiler include search path.
-n	No execute flag. This option causes the commands used to compile and link the target to display without executing them.
-output <name>	Create an executable named <name>. (An appropriate executable extension is automatically appended.)

Table 5-2: mbuild Options on Windows (Continued)

Option	Description
-O	Build an optimized executable.
-setup	Set up default options file. This switch 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 (-g, -v, and -f) are available on the mcc command line and are passed along to mbuild.

Customizing mbuild

If you need to see which switches mbuild passes to your compiler and linker, use the verbose switch, -v, as in:

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

to generate a list of all the current compiler settings. If you need to change the switches, 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 switch into an integrated development environment (IDE) or makefile that you need to maintain outside of MATLAB.

Note: If you want to use an IDE to create your applications, you should look at the project template files included in the following compiler-specific directory that corresponds to your compiler:

```
<matlab>\extern\examples\cppmath\borland
<matlab>\extern\examples\cppmath\watcom
<matlab>\extern\examples\cppmath\msvc
```

Distributing Stand-Alone Windows 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 against. This package of files includes:

- Application (executable)
- `libmmfile.dll`
- `libmatlb.dll`
- `libmcc.dll`
- `libmx.dll`
- `libut.dll`

For example, to distribute the Windows version of the `ex1` example, you need to include `ex1.dll`, `libmmfile.dll`, `libmatlb.dll`, `libmcc.dll`, `libmx.dll`, and `libut.dll`.

Building on Macintosh

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

Note: For Power Macintosh shared libraries to be found by the Macintosh operating system at runtime, the shared libraries need to appear in either

- The `System Folder:Extensions:` folder, or
- The same folder as the application that uses the shared libraries.

The MATLAB installer automatically puts an alias to `<matlab>:extern:lib:PowerMac:` (where the shared libraries are stored) in the `System Folder:Extensions:` folder and names the alias `MATLAB Shared Libraries`.

Configuring mbuild

The `mbuild` script provides a convenient and easy way to configure your ANSI compiler with the proper switches to create an application. To configure your compiler, use

```
mbuild –setup
```

Note: You must run `mbuild –setup` before you create your first stand-alone application; otherwise, when you try to create a stand-alone application, you will get the error

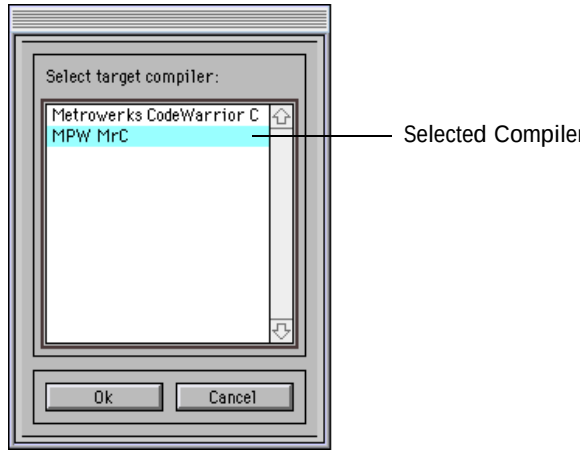
```
An mbuildopts file was not found or specified.  Use
"mbuild –setup" to configure mbuild for your compiler.
```

Run the `setup` option from the MATLAB prompt.

```
mbuild –setup
```

Executing `mbuild` with the `setup` option displays a dialog with a list of compilers whose options files are currently included in the

<matlab>:extern:scripts: folder. (Your dialog may differ from this one.) This figure shows MPW MrC selected as the desired compiler.



Click **Ok** to select the compiler. If you previously selected an options file, you are asked if you want to overwrite it. If you do not have an options file in your <matlab>:extern:scripts: folder, the setup option creates the appropriate options file for you.

Note: If you select MPW, mbuild –setup asks you if you want to create UserStartup•MATLAB_MEX and UserStartupTS•MATLAB_MEX, which configures MPW and ToolServer for building MEX-files and stand-alone compiler applications.

When this message displays, mbuild is configured properly.

```
MBUILD –setup complete.
```

Changing Compilers. If you want to change your current compiler, use the mbuild –setup command.

Verifying mbuild

There is C source code for an example, ex1.c included in the <matlab>:extern:examples:cmath: directory. 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`, a stand-alone application. You can now run your stand-alone application by double-clicking its icon. The results should be:

```
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 –em hello.m
```

This command should complete without errors. To run the stand-alone application, `hello`, invoke it as you would any other Macintosh console application, by double-clicking its icon. 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.

The mbuild Script

The `mbuild` script supports various switches that allow you to customize the building and linking of your code. The only required option that all users must

execute is setup; the other options are provided for users who want to customize the process. The mbuild syntax and options are:

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

Table 5-3: mbuild Options on Macintosh

Option	Description
–c	Compile only; do not link.
–D<name>[=<def>]	Define C preprocessor macro <name> [as having value <def>.]
–f <file>	Use <file> as the options file. (Not necessary if you use the –setup option.) If <file> is specified, it is used as the options file. If <file> is not specified and there is a file called mbuildopts in the current directory, it is used as the options file. If <file> is not specified and mbuildopts is not in the current directory and there is a file called mbuildopts in the directory <matlab>:extern:scripts:, it is used as the options file. Otherwise, an error occurs.
–g	Build an executable with debugging symbols included.
–h[elp]	Help; prints a description of mbuild and the list of options.
–I<pathname>	Include <pathname> in the compiler include search path.
<name>=<def>	Override options file setting for variable <name>.
–n	No execute flag. This option causes the commands used to compile and link the target to display without executing them.
–output <name>	Create an executable named <name>.

Table 5-3: mbuild Options on Macintosh (Continued)

Option	Description
`-O`	Build an optimized executable.
`-setup`	Set up default options file. This switch should be the only argument passed.
`-v`	Verbose; print all compiler and linker settings.

Note: Some of these options (`-g`, `-v`, and `-f`) are available on the `mcc` command line and are passed along to `mbuild`.

Customizing mbuild

If you need to customize the application building process, use the verbose switch, `-v`, as in:

```
mbuild -v filename.m [filename1.m filename2.m ...]
```

to generate a list of all the current compiler settings. After you determine the desired changes that are necessary for your purposes, use an editor to make changes to the options file that corresponds to your compiler. You can also use the settings obtained from the verbose switch to embed them into an IDE or makefile that you need to maintain outside of MATLAB.

Distributing Stand-Alone Macintosh 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 against. These lists show which files should be included on the Power Macintosh and 68K Macintosh systems:

Power Macintosh

- Application (executable)
- `libmmfile`
- `libmatlb`
- `libmcc`
- `libmx`
- `libut`

68K Macintosh

- Application (executable)
- `libmmfile.o`
- `libmatlb.o`
- `libmcc.o`
- `libmx.o`
- `libut.o`

For example, to distribute the Power Macintosh version of the `ex1` example, you need to include `ex1`, `libmmfile`, `libmatlb`, `libmcc`, `libmx`, and `libut`. To distribute the 68K Macintosh version of the `ex1` example, you need to include `ex1`, `libmmfile.o`, `libmatlb.o`, `libmcc.o`, `libmx.o`, and `libut.o`.

Troubleshooting mbuild

This section identifies some of the more common problems that might occur when configuring `mbuild` to create stand-alone external applications.

Options File Not Writable

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 writable, the process will terminate and you will not be able to use `mbuild` to create your applications.

Directory or File Not Writable

If a destination directory or file is not writable, 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 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 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 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`.

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 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 when you try to access the Compiler. 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 the previous solution does not address your difficulty with the MATLAB Compiler, contact Technical Support at The MathWorks at `support@mathworks.com` or 508 647-7000.

Coding External Applications

This section begins with some important information regarding memory usage. It then examines how to code applications as M-files only. Later on, the chapter explains how to code part of the application as M-files and part as C or C++ functions.

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."

Reducing Memory Usage

The MATLAB M-File Math Library (`libmmfile`) is very large. `libmmfile` contains the compiled versions of every math M-file included in the C Math library. (For example, `rank`, `gradient`, and `hadamard` are all implemented as M-files and are therefore part of `libmmfile`.) Since the average application calls only a small subset of the routines in `libmmfile`, dynamically linking against `libmmfile` typically uses excess memory. An alternative to dynamically linking against the entire `libmmfile` is to compile (with the MATLAB Compiler) only those function M-files that your application needs.

To compile only the required function M-files:

- Add the MATLAB Compiler-compatible M-files directory to your path. The directories containing these M-files are:
  - On UNIX, `<matlab>/extern/src/math/tbxsrc`
  - On Windows, `<matlab>\extern\src\math\tbxsrc`
  - On Macintosh, `<matlab>:extern:src:math:tbxsrc:`
- Compile the M-file with the MATLAB Compiler.
- Edit your `mbuild` options file so that it does not link with `libmmfile`. This means that you will receive `link undefined` errors when you have not compiled all functions that you are using from `libmmfile`.

Note: Do not compile functions to C++ that are in `libmmfile` because the C++ Math Library implements the function by calling the C version of the compiled M-file. This means that recompiling the C version is sufficient for use with the C++ Math Library. You may receive errors if you attempt to compile the C++ version of these included files.

Coding with M-Files Only

One way to create a stand-alone external 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 external application.

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.

`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 a stand-alone external application, invoke the MATLAB Compiler twice, one time for each M-file:

```
mcc -ec main
mcc -ec mrank
```

The `-e` option flag causes the MATLAB Compiler to generate C source code suitable for external applications. For example, the MATLAB Compiler generates C source code files `main.c` and `mrank.c`. `main.c` contains a C function

named `main`; `mrank.c` contains a C function named `mlfMrank`. (The `-c` option flag inhibits invocation of `mbuild`.)

Note that the MATLAB Compiler treats M-file functions named `main` differently from all other M-file functions. `main` is the only function that retains its name; for all other M-file functions, the MATLAB Compiler affixes the `mlf` prefix to the front of the function name.

To build an executable application, compile and link these files using `mbuild`. You can automate the entire process of invoking the MATLAB Compiler two times, using `mbuild` to compile the files with your ANSI C compiler, and linking the code by using the command

```
mcc -e main mrank
```

Figure 5-2 illustrates the process of building a stand-alone, external C application from two M-files. The commands to compile and link depend on the operating system being used. See the "Building Stand-Alone External C/C++ Applications" section for details.

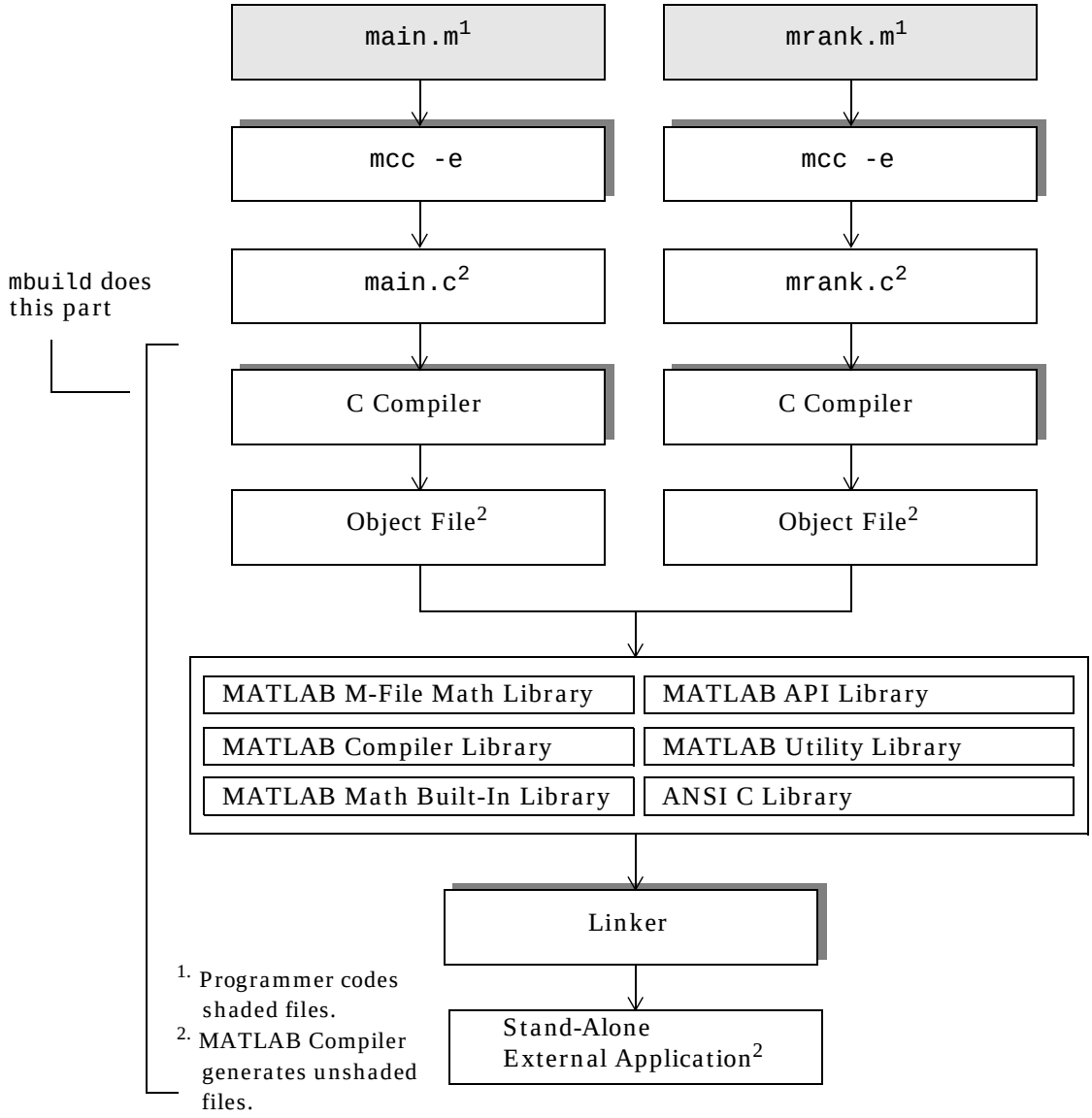


Figure 5-2: Building Two M-Files into a Stand-Alone C External Application

For C++ code, you use the `mcc -p` command instead of `mcc -e`, a C++ compiler instead of a C compiler, and 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 use `mcc -p` for C++ instead of `mcc -e` 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 external 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 -ec rank
```

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

```
mcc -ec main
mcc -ec mrank
```

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

```
mbuild main.c rank.c mrank.c
```

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

Compiling mrank.m and rank.m as Helper Functions

Another way of building the `mrank` external 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 –e main mrank rank
```

or

```
mcc –e –h main
```

For C++:

```
mcc –p main mrank rank
```

or

```
mcc –p –h main
```

These commands create a single file (`main.c`) of C or C++ source code. Files built with helper functions run slightly faster.

Note: Each of these commands automatically invoke `mbuild` because `main.m` is explicitly included on the command line.

Print Handlers

A print handler is a routine that controls how your application displays the output generated by `mlf` calls.

If you do not register a print handler, the system provides a default print handler for your application. The default print handler writes output to the standard output stream. You can override this behavior, however, by registering an alternative print handler. In fact, if you are coding a stand-alone external application with a GUI, then you *must* register an alternative print handler. This makes it possible for application output to be displayed inside a GUI mechanism, such as a Windows message box or a Motif Label widget.

If you create a print handler routine, you must register its name at the beginning of your stand-alone external application.

The way you establish a print handler depends on whether or not your source code is written entirely as function M-files.

Note: The print handlers discussed in this section work for C++ as well as C applications. However, we recommend that you use different print handlers for C++ applications. See the *MATLAB C++ Math Library User's Guide* for details about C++ print handlers.

Source Code Is Not Entirely Function M-Files

If some (or all) of your stand-alone external application is coded in C (as opposed to being written entirely as function M-files), then you must

- Register the print handler.
- Write a print handler.

To register a print handler routine, call `mlfSetPrintHandler` as the first executable line in `main` (or `WinMain`). For example, the first line of `mrankwin.c` (a Microsoft Windows program) registers a print handler routine named `WinPrint` by calling `mlfSetPrintHandler` as

```
mlfSetPrintHandler(WinPrint);
```

Next, you must write a print handler routine. The print handler routine in `mrankwin.c` is

```
static int totalcnt  = 0;
static int upperlim  = 0;
static int firsttime = 1;
char *OutputBuffer;

void WinPrint( char *text)
{
        int cnt;

        if (firsttime) {
                OutputBuffer = (char *)mxCalloc(1028, 1);
                upperlim += 1028;
                firsttime = 0;
        }

        cnt = strlen(text);
        if (totalcnt + cnt >= upperlim) {
                char *TmpOut;
                TmpOut = (char *)mxCalloc(upperlim + 1028, 1);
                memcpy(TmpOut, OutputBuffer, upperlim);
                upperlim += 1028;
                mxFree(OutputBuffer);
                OutputBuffer = TmpOut;
        }
        strncat(OutputBuffer, text, cnt);
}
```

Whenever an `mlf` function wants to write data, your application automatically intercepts that request and causes a call to `WinPrint`. In fact, the application calls `WinPrint` one time for every line of data to be output. For example, by default, vector `Matrix R` contains 12 lines of data. Therefore, the call

```
mlfPrintMatrix(R);
```

causes the application to call `WinPrint` 12 times, each time passing the next line of data. `WinPrint` allocates enough dynamic memory (in `OutputBuffer`) to

hold the printable contents of `Matrix R`. When `OutputBuffer` is filled with all 12 lines of data, `WinMain` calls `WinFlush`.

```
void WinFlush(void)
{
        MessageBox(NULL, OutputBuffer, "MRANK", MB_OK);
        mxFree(OutputBuffer);
}
```

`WinFlush` instantiates a `MessageBox`, which displays the 12 lines of data in a Microsoft Windows output window.

For more details on `mlfPrintMatrix`, see the *MATLAB C Math Library User's Guide*.

Source Code Is Entirely Function M-Files

If your stand-alone external application source code is written entirely as function M-files, you create a print handler by

- Calling a print handler initialization routine as the first executable line in the first M-file to be executed.
- Writing a print handler in C or C++.
- Writing a one-line (dummy) function M-file.

For example, suppose that `mr.m` contains the source code of the first M-file to be called in a Windows external application:

```
function mr(m)
r = mr(m);
r
```

that calls a print handler initialization routine as the first executable line in `mr.m`, for example:

```
function mr(m)
initprnt; % Call a print handler initialization routine
r = mrank(m);
r
```

The next step is to write a print handler in C or C++. Your print handler must consist of two functions:

- A print handler initialization function.
- A print handler function.

You must give the print handler initialization function a name beginning with `mlf` and ending with the name you specified in the M-file. For example, since the name specified in `mr.m` is `initprnt`, you must name the print handler initialization function `mlfInitprnt`. (Note: Unlike MATLAB, C is case sensitive, therefore, the first letter following `mlf` must be capitalized.)

All print handler initialization functions must register the name of the print handler function. To do so, all print handler initialization functions must call `mlfSetPrintHandler`. (See the previous section for more information on `mlfSetPrintHandler`.)

For example, the sample print handler file `myph.c` contains the two necessary functions to establish a simple print handler:

```
/* print handler initialization function */
void mlfInitprnt(void)
{
 /* Establish myPrintHandler as the print handler routine. */
   mlfSetPrintHandler(myPrintHandler);
}

/* print handler function */
static void myPrintHandler(const char *s)
{
   printf("%s", s);
}
```

You must compile `myph.c` with one of the supported C compilers. Then, you must ensure that the resulting object file is linked into the stand-alone external application.

Finally, you must write the dummy M-file. If you omit the dummy M-file, the MATLAB Compiler cannot compile the call to the print handler initialization routine. Give the dummy M-file the same name as the print handler initialization routine specified in the first M-file to be executed. For example,

the name of the print handler initialization routine in `mr.h` is `initprnt`; therefore, name the dummy M-file `initprnt.m`.

The dummy M-file should contain the keyword `function` followed by the name of the print handler initialization routine specified in the first M-file to be executed. For example, the name of the print handler initialization routine in `mr.h` is `initprnt`; therefore, the dummy M-file should simply contain:

```
function initprnt
```

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). If you do not identify to the Compiler which functions will be called through `feval` and the function is not contained in the C/C++ Math Library, the Compiler will produce a runtime error.

If you are using the `%#function` pragma to define functions that are not available in M-code, you must write a dummy M-file 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-file that contains a function line, such as:

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

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-file.

Note: All of the functions in the C/C++ Math Libraries are automatically available to the `feval` function without using the `%#function` pragma.

Stand-Alone C Mode

In stand-alone C mode, any function that will be called with `feval` must be a separately compiled function with the standard `mlfxxx()` interface. The function can be written by hand in C code or generated using the MATLAB Compiler.

Stand-Alone C++ Mode

In stand-alone C++ mode, any function can be called through `feval`, but the function must be mentioned with the pragma.

Mixing M-Files and C or C++

Another way to create a stand-alone external 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 external 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.

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

```
#include <stdio.h>
#include <math.h>
#include "matlab.h"

/* Prototype for mlfMrank */
extern mxArray *mlfMrank( mxArray * );

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

    /* 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);

}
```

To build this stand-alone external application, use:

```
mcc –e mrank mrankp.c
```

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

The MATLAB Compiler provides three different online versions of `mrankp.c`:

- `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`.
- `mrankmac.c` contains a Macintosh version of `mrankp.c`.

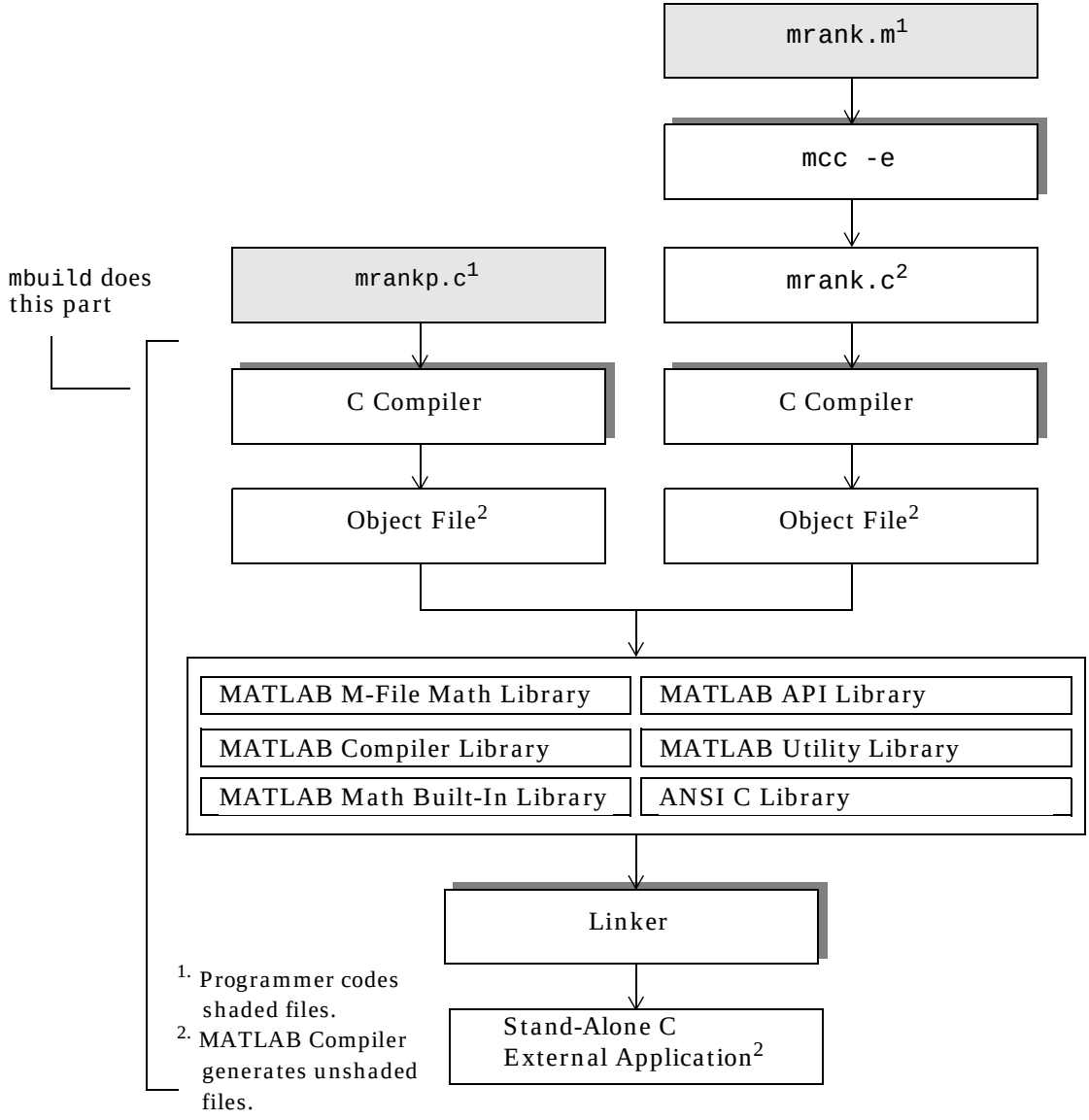


Figure 5-3: Mixing M-Files and C Code to Form a Stand-Alone Application

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.

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`, `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:

```
mxDestroyArray(N);
mxDestroyArray(R);
```

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 external 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"

/*
 * Function prototype; the MATLAB Compiler creates mlfMultarg
 *  from multarg.m
 */
mxArray * mlfMultarg( mxArray **, mxArray *, mxArray * );

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;

    /* 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);
    return(0);
}
```

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

```
mcc –e multarg
```

The program first displays the contents of a 2-by-2 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:

```
mxArray *mlfMultarg(mxArray **b_lhs_, mxArray *x_rhs_,
                    mxArray *y_rhs_)
```

This C function header shows two input arguments and two output arguments.

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.

This program solves a real-world problem. As a result, it is longer and more complex than the other examples. You do not need to understand this example to use the C++ Math Library successfully. You may want to skip this section on your first reading.

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

Note: Since the MATLAB Version 5 of the Image Processing Toolbox uses MEX-files for important parts of its functionality, this example includes the MATLAB Version 4 of the files used in the Image Processing Toolbox.

The 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.

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 C++ Math Library. Therefore, 10 M-files need to be compiled to build this example. However, since

we need to compile these M-files into stand-alone code, they need to be modified slightly. These modifications are necessary because the M-files used in this example make calls to Handle Graphics® routines, none of which exists in the MATLAB C++ Math Library. The code for the 10 M-files that are compiled is not included in this chapter; however, it is included in the `examples` directory.

You can generate `ex9.cpp` with the following command:

```
mcc -phm ex9
```

This command compiles the M-file `ex9.m` and all other M-files called by `ex9`, searching for them on the MATLAB path, and collects the output into a C++ file called `ex9.cpp`. The name of the output file derives from the name of the M-file, `ex9`, listed on the command line. Note that it is not necessary to specify the `.m` filename extension. The `-p` switch instructs the MATLAB Compiler to generate C++ code. The default output language (without the `-p` switch) is C. The `-h` switch instructs the compiler to find all the necessary helper functions automatically. Finally, the `-m` switch tells the compiler to produce a main program; the resulting C++ file can be compiled into a complete stand-alone application.

Note: The MATLAB Compiler can only compile calls to functions for which it either has an internal table entry or that it can find on the MATLAB path. Since the M-files in this example are modified versions of the MATLAB 4.2 Image Processing Toolbox, it is very important that you have the `examples` directory on your MATLAB path when you compile `ex9.m`; if you do not, it is possible that the MATLAB Compiler will either not find the required M-files, or that it will find files of the same name that will not work with this example.

`ex9.cpp` is broadly divided into three sections: declarations, compiled helper functions, and the main routine. The first section includes the required header

file and provides declarations of constants and functions defined in ex9.cpp. As shown below, the MATLAB Compiler declares constants in a two-step process.

```
/* static array S5_ (3 x 3) int, line 167 */
static int S5r_[] =
{
        0, 1, 0, 1, 0, 1, 0,
        1, 0,
};
static mwArray S5_ = mwArray( 3, 3, S5r_ );
```

The comment that precedes this constant indicates the constant is a 3-by-3 array of integers (note, however, that the MATLAB C++ Math Library stores integer arrays as arrays of double-precision, floating-point numbers) from line 167. The following code appears on line 167 of edge.m:

```
b = filter2([0 1 0;1 0 1;0 1 0],bb);
```

This constant represents the first argument in the call to filter2(). The first step in declaring the constant is to create a static C++ array that contains the data; as usual, this data is organized in column-major order. The second step is to initialize a static mwArray with the data from the static C++ array.

The MATLAB Compiler is very careful to declare all functions before their first use. Here is the declaration of the helper function edge():

```
static mwArray edge(mwArray *,mwArray =mwArray::DIN,
mwArray =mwArray::DIN, mwArray =mwArray::DIN,
mwArray =mwArray::DIN);
```

There are two interesting aspects of this declaration. Notice first that the Compiler declares edge() as a static function. This means that edge() can only be called from within this file. If you want to compile a function so that it can be called from more than one file, compile it by itself rather than as a helper function. Also note that this function has five arguments, but that the last four are optional. The MATLAB Compiler uses the special variable mwArray::DIN as the default value for optional arguments. Since all optional arguments have a special default value, the MATLAB Compiler can check the values of all the input arguments against mwArray::DIN and thereby count the number of arguments actually passed to the function. Just as MATLAB uses nargin and nargout to count the number of input and output variables passed to a function, the MATLAB Compiler uses the variables _nargin_count and _nargout_count.

The next section of ex9.cpp contains the compiled helper functions. These helper functions are the compiled versions of the M-files called by ex9(). In C++ code generated by the MATLAB Compiler, the only difference between a helper function and any other function is that every helper function is declared static, as shown above. This is the definition of the edge() function, minus most of the body, which is too long to reproduce here:

```
mwArray edge(mwArray *tol_out_, mwArray a, mwArray tol,
             mwArray method, mwArray k)
{
   // Code omitted

   // Begin nargin() counting code.
   int _nargin_count = 0;
   if (a != mwArray::DIN) _nargin_count++;
   else a = mwArray();
   if (tol != mwArray::DIN) _nargin_count++;
   else tol = mwArray();
   if (method != mwArray::DIN) _nargin_count++;
   else method = mwArray();
   if (k != mwArray::DIN) _nargin_count++;
   else k = mwArray();

  // More code omitted
}
```

Notice that there are five arguments and that the default values for optional arguments do not appear in the definition; C++ requires them in the declaration, so that they are known at the call site. However, the body of the function contains a block of code that counts the number of input arguments with nondefault values (i.e., those input arguments whose value is not equal to mwArray::DIN). The MATLAB Compiler detected the use of nargin in edge.m, and automatically generated C++ code to count the number of input arguments. If edge.m had used nargout, the MATLAB Compiler would have generated code to count the number of output arguments as well. Just as mwArray::DIN is the default value for optional input arguments, NULL is the default value for optional output arguments.

The final section of ex9.cpp is the main routine:

```
(1) int main(int argc, char *argv[])
    {

    #ifdef EXCEPTIONS_WORK
(2)    try {
    #endif

       mwArray TempMatrix_[32];
       mwArray infile;
       mwArray outfile;
(3)    mwArray x;
       mwArray map;
       mwArray inten;
       mwArray bw;

       infile = (argc >1) ? mwArray(argv[1]) : mwArray::DIN;
(4)    outfile = (argc >2) ? mwArray(argv[2]) : mwArray::DIN;
       // EX9 edge detection on Microsoft Windows bitmap files

       // EX9( infile, outfile )
       // Opens and reads the Windows bitmap stored in infile and
       // writes the resulting bitmap into outfile.

        // Copyright (c) 1997 by The MathWorks, Inc.
(5)    x = bmpread(&map,0,infile);
(6)    inten = ind2gray(x,map);
(7)    bw = edge(0,inten);
(8)    x = gray2ind(&map,bw);
(9)    bmpwrite(x,map,outfile);

    #ifdef EXCEPTIONS_WORK
(10)   } catch(mwException &ex) { cout << ex << endl; }
    #endif

       return 0;
    }
```

Notes

Each of the numbered steps is explained in more detail below. Steps 5 through 9 correspond to the five basic steps of the image processing program outlined at the beginning of this example.

1 Begin the `main()` function. `ex9` is called with two arguments: the name of the input file and the name of the output file. The `main()` function could just as easily be rewritten as a subroutine of a larger program.

2 Begin the `try` block. Because the code in the MATLAB C++ Math Library throws exceptions to indicate errors, a `try` block within the main routine must enclose all calls to the MATLAB C++ Math Library routines (even those routines called indirectly). Establishing a `try` block permits the program to catch any exceptions that the MATLAB C++ Math Library might throw. See the *MATLAB C++ Math Library User's Guide* for more details on exception handling.

3 Declare local variables that will store results from the edge detection steps. `x` stores the image data, `map` the colormap, `inten` the grayscale intensity image derived from the original bitmap, and `bw` the black and white intensity image returned by `edge()`. `infile` and `outfile` store the names of the input and output bitmap files, and `TempMatrix_` is a variable automatically generated by the MATLAB Compiler.

4 Obtain input and output filenames from the command line arguments. Since the first argument, stored in `argv[0]`, is always the name of the program being run, `argv[1]` contains the name of the input file, and `argv[2]` contains the name of the output file. Note that this code sets the filenames to a default value if the corresponding argument was not supplied. Later checks will generate errors if either the input file or the output file is omitted.

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

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.

10. Catch exceptions with a `catch` block. If an exception occurs anywhere in the program, control resumes here. Print out the message associated with the exception and then exit the program.

Now that you have a better understanding of the program, you can build and run it. See the earlier section, "Building Stand-Alone External C/C++ Applications," for details on how to build this example. Make sure the file `trees.bmp` is in the directory where you build the executable; copy it from the `examples` directory if necessary.

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. If all goes well, the program 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.

The Generated Code

This chapter details:

- The C code that the `mcc` command generates (MEX-files).
- The C code that the `mcc -e` command generates (stand-alone).
- The C++ code that `mcc -p` command generates (stand-alone).

Porting Generated Code to a Different Platform

The generated code is portable among platforms. However, if you build a MEX-file (say `foo.mex`) from `foo.m` on a PC, that same MEX-file will not run on a Macintosh or a UNIX system.

For example, you cannot simply copy `foo.mex` from a PC to a Sun system and expect the code to work, because binary formats are different on different platforms (MEX-files are binaries). However, you could copy either `foo.c` (generated C code) or `foo.m` from the PC to the Sun system. Then you could use `mex` or `mcc` (on the Sun) to produce a `foo.mex` that would work on the Sun system.

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:

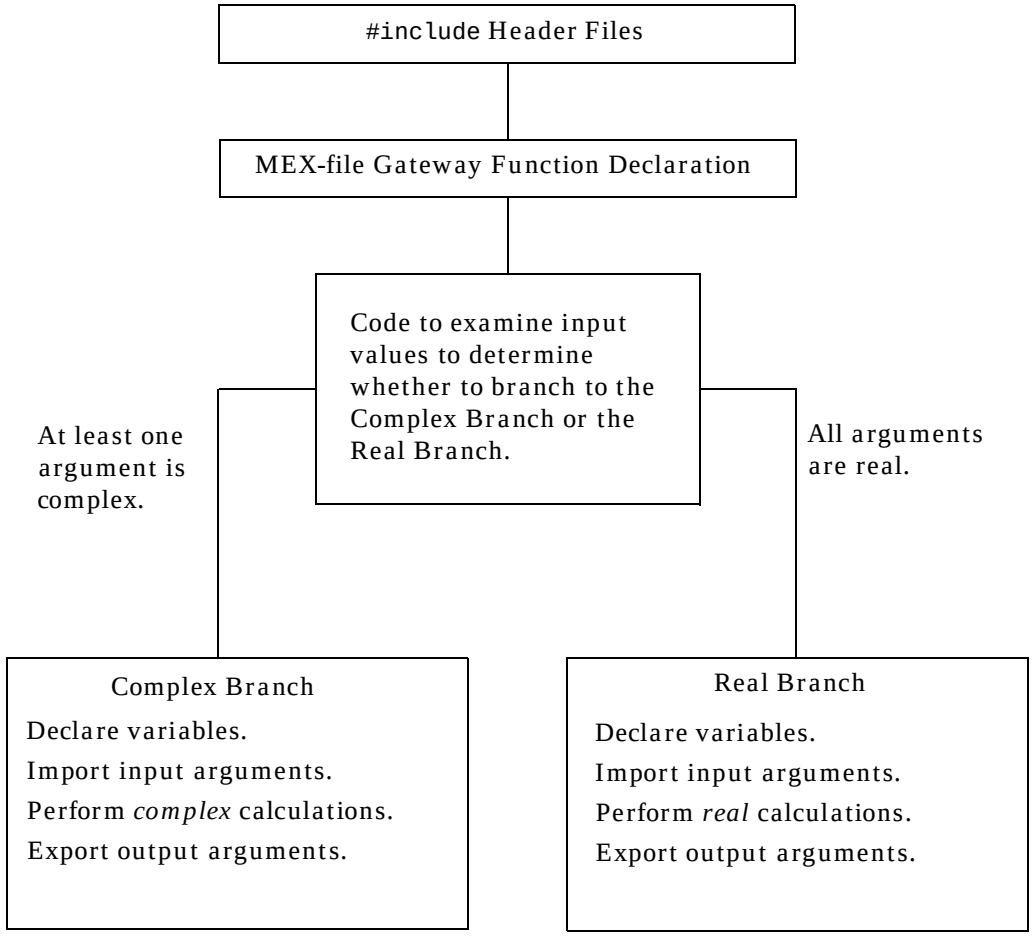


Figure 6-1: Structure of MEX-File Source Code Generated by Compiler

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 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"
```

The included header files are:

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 ****************
 *
 *       I0_             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 I0_ = 0
```

The mapping between commented MATLAB Compiler assumptions and the actual C variable declarations is:

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);
```

See Chapter 4 for a detailed examination of how option flags, assertions, and pragmas influence the calculations section.

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 external applications. This section explains the code. Most C source code generated by the `mcc –e` command shares this common format:

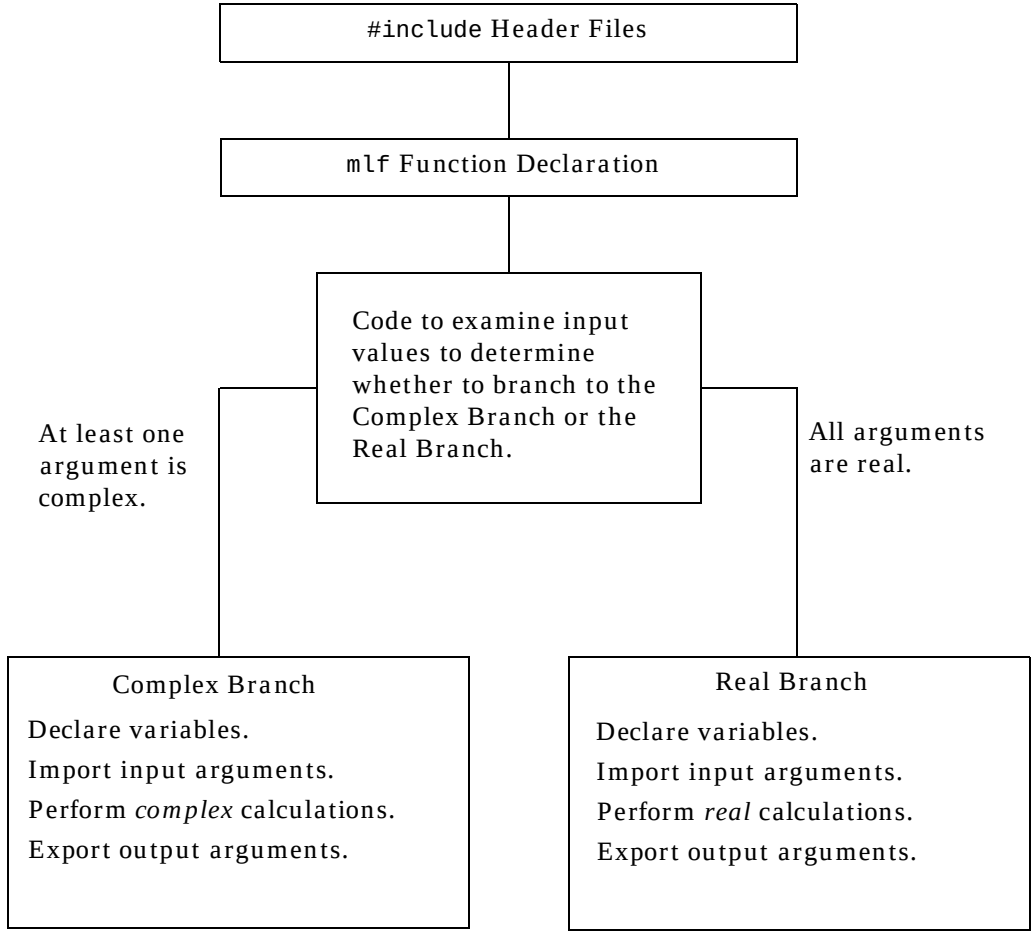


Figure 6-2: Structure of Stand-Alone C Source Code Generated by Compiler

Header Files

The MATLAB Compiler generates these `#include` preprocessor statements:

```
#include <math.h>
#include "matrix.h"
#include "mcc.h"
#include "matlab.h"
```

The included header files are:

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` reference page in Chapter 8 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.

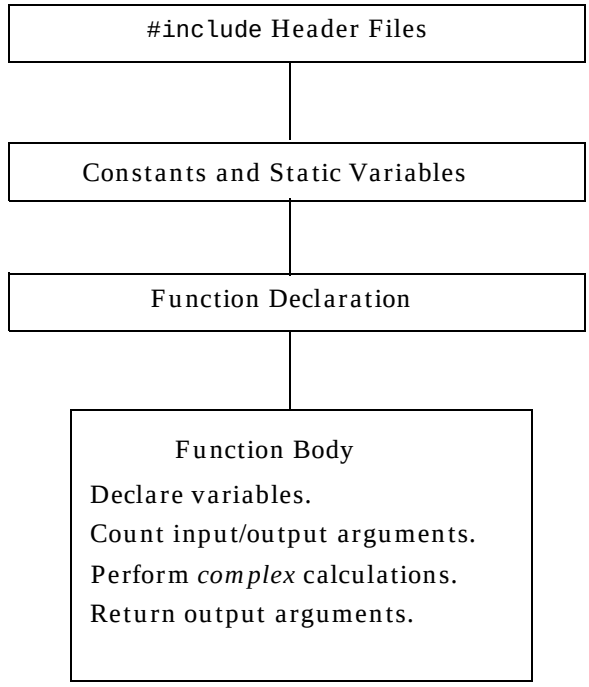


Figure 6-3: Structure of Stand-Alone C++ Source Code Generated by Compiler

Header Files

All generated C++ functions include two header files:

```
#include <iostream.h>
#include "matlab.hpp"
```

The included header files are:

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` reference page in Chapter 8 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);
```

7

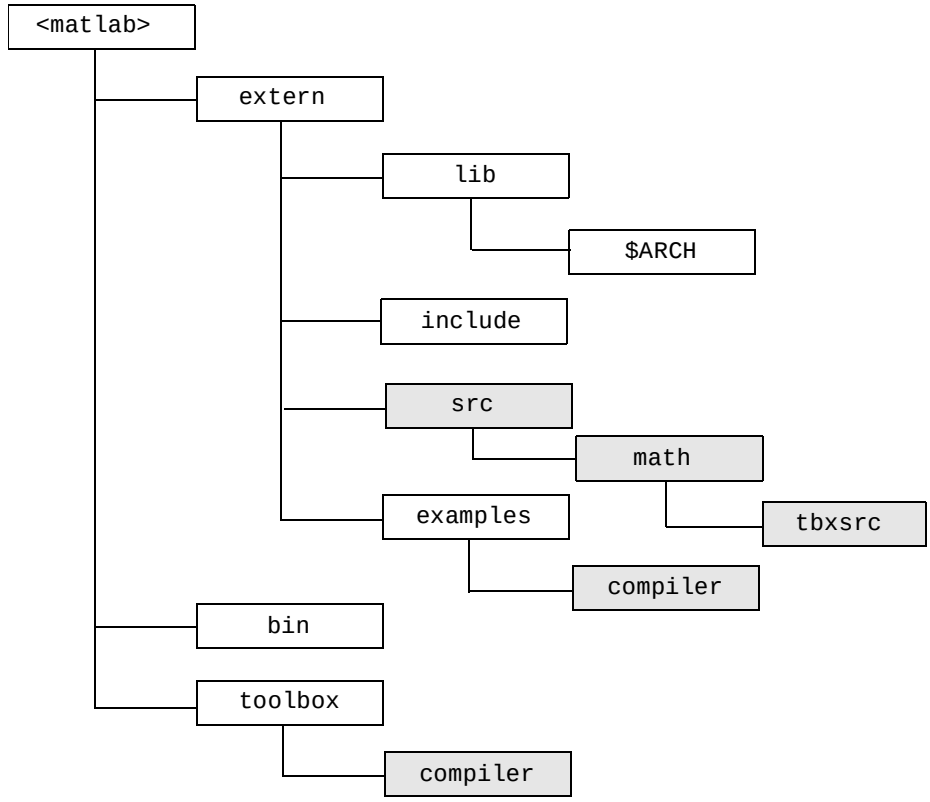
Directory Organization

This chapter describes the directory organization of the MATLAB Compiler on UNIX, Microsoft Windows, and Macintosh systems.

Note that if you also install the C or 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, which is:

Compiler_Readme	Optional document that describes configuration details, known problems, workarounds, and other useful information.

<matlab>/extern/lib/$ARCH

The <matlab>/extern/lib/$ARCH directory contains libraries, where $ARCH specifies a particular UNIX platform. For example, on a Sun SPARCstation running SunOs 4, the $ARCH directory is named sun4.

The library for the MATLAB Compiler is:

libmccmx.a	MATLAB Compiler Library for MEX-files. Contains the mcc and mcm routines required for building MEX-files.

On some UNIX platforms, this directory contains two versions of this library. Library filenames ending with .a are static libraries and filenames ending with .so or .sl are shared libraries.

The libraries for the MATLAB C Math Library, a separate product, are:

libmmfile.a	MATLAB M-File Math Library. Contains callable versions of the M-files in the MATLAB Toolbox. Needed for building stand-alone external applications that require MATLAB toolbox functions.
libmcc.a	MATLAB Compiler Library for stand-alone external applications. Contains the mcc and mcm routines required for building stand-alone external applications.

`libmatlb.a`	MATLAB Math Built-In Library. Contains callable versions of MATLAB built-in math functions and operators. Required for building stand-alone external applications.

The library for the MATLAB C++ Math Library, a separate product, is:

`libmatpp.a`	MATLAB C++ Math Library. Contains callable versions of MATLAB built-in math functions and operators. Required for building stand-alone C++ external applications.

<matlab>/extern/include

The `<matlab>/extern/include` directory contains the header files for developing C or C++ applications that interface with MATLAB.

The header file for the MATLAB Compiler is:

`mcc.h`	Header file for MATLAB Compiler Library.

The header file for the MATLAB C Math Library, a separate product, is:

`matlab.h`	Header file for MATLAB Math Built-In Library and MATLAB M-File Math Library.

The relevant header files from MATLAB are:

`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.

<matlab>/extern/include/cpp

The header file for the MATLAB C++ Math Library, a separate product, is:

`matlab.hpp`	Header file for MATLAB C++ Math Library.

<matlab>/extern/src/math/tbxsrc

The `<matlab>/extern/src/math/tbxsrc` directory contains the MATLAB Compiler-compatible M-files. These files are:

`automesh.m`	`base2dec.m`	`bin2dec.m`
`corrcoef.m`	`cov.m`	`datestr.m`
`datevec.m`	`fmin.m`	`fmins.m`
`fzero.m`	`gradient.m`	`griddata.m`
`hex2dec.m`	`hex2num.m`	`interp1q.m`
`interp2.m`	`interp4.m`	`interp5.m`
`interp6.m`	`ismember.m`	`ntrp23.m`
`ntrp23s.m`	`ntrp45.m`	`numjac.m`

ode113.m	ode15s.m	ode23.m
odeset.m	odezero.m	polyeig.m
polyfit.m	polyval.m	quad.m
quad8.m	str2num.m	strcat.c
strvcat.c		

<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 8-35).
fibocert.m	M-file that explores assertions (page 4-15).
fibocon.m	M-file used to show MEX-file source code (page 6-4).
fibomult.m	M-file that explores helper functions (page 4-21).
hello.m	M-file that displays Hello, World.
houdini.m	Script M-file that cubes each element of a 2-by-2 matrix (page 3-12).
initprnt.m	Dummy M-file (page 5-41).
lu2.m	M-file example that benefits from %#ivdep (page 8-9).
DOT.mexrc.sh	Example .mexrc.sh file that supports ANSI-C compilers.

`Makefile`	Example `gmake`-compatible makefile for building stand-alone external 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 5-31).
`mrank.m`	M-file to calculate the rank of magic squares (page 5-31).
`mrankmac.c`	Macintosh version of `mrankp.c`.
`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 5-48).
`mrankwin.c`	Windows version of `mrankp.c`.
`multarg.m`	M-file that contains two input and two output arguments (page 5-49).
`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 5-49).
`mycb.m`	M-file example used to study callbacks (page 4-21).
`mydep.m`	M-file example used to show a case when `%#ivdep` generates incorrect results (page 8-9).
`myfunc.m`	M-file that explores helper functions (page 4-14).

`myph.c`	C function print handler initialization (page 5-40).
`mypoly.m`	M-file example used to study callbacks (page 4-23).
`novector.m`	M-file example that demonstrates the influence of vectorization (page 4-29).
`plot1.m`	M-file example that calls `feval` (page 4-25).
`plotf.m`	M-file example that calls `feval` multiple times (page 4-25).
`squares1.m`	M-file example that does not preallocate a matrix (page 4-27).
`squares2.m`	M-file example that preallocates a matrix (page 4-27).
`squibo.m`	M-file to calculate “squibonacci” numbers. Compile `squibo.m` into a MEX-file (page 3-3).
`squibo2.m`	M-file example that contains a `%#realonly` pragma (page 8-11).
`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 4-29).

<matlab>/bin

Files in the <matlab>/bin directory that are relevant to compiling include:

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.
cxxopts.sh	Options file for building C++ MEX-files.
gccopts.sh	Options file for building gcc MEX-files.
mbuildopts.sh	Shell script used by mbuild and mbuild.m.
mexopts.sh	Options file for building MEX-files using the native compiler.

<matlab>/toolbox/compiler

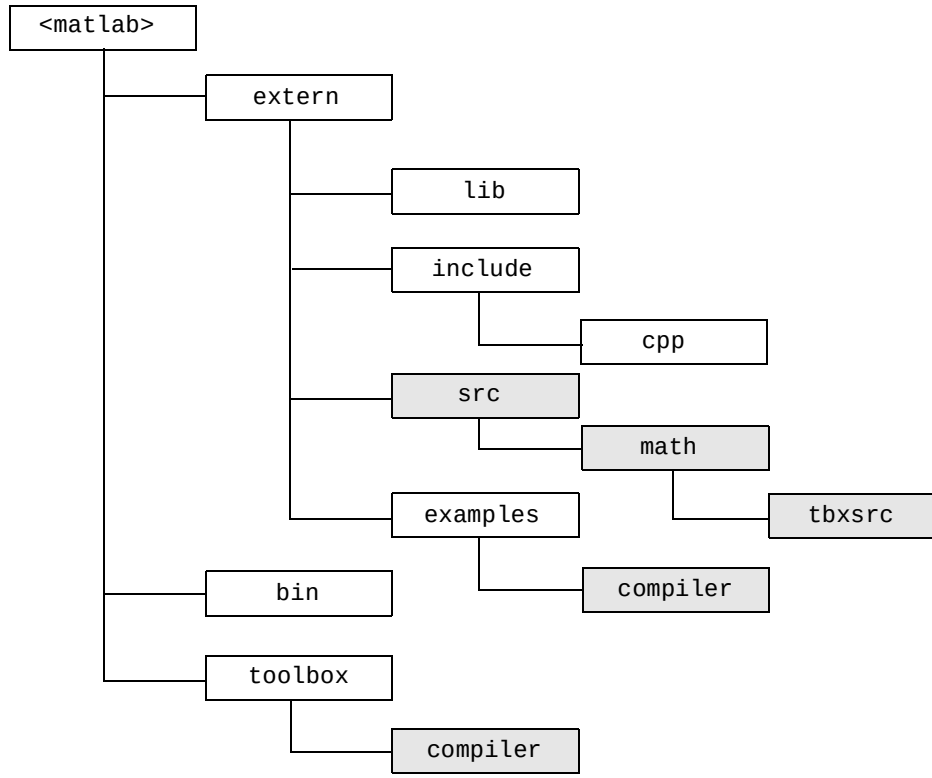
The <matlab>/toolbox/compiler directory contains the MATLAB Compiler's additions to the MATLAB command set:

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	Builds stand-alone applications from MATLAB command prompt.
mbvector.m	Assert variable is vector.
mcc.m	Invoke the MATLAB Compiler.
mccexec.mex	MATLAB Compiler internal routine.
mccload.mex	MATLAB Compiler internal routine.
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, which is:

Compiler_Readme	Optional document that describes configuration details, known problems, workarounds, and other useful information.

<matlab>\bin

The <matlab>\bin directory contains the libraries required to build external applications and MEX-files.

The libraries for the MATLAB Compiler are:

libmccmx.dll	MATLAB Compiler Library for MEX-files. Contains the mcc and mcm routines required for building MEX-files.

The libraries for the MATLAB C Math Library, a separate product, are:

libmmfile.dll	MATLAB M-File Math Library. Contains callable versions of the M-files in the MATLAB Toolbox. Needed for building stand-alone external applications that require MATLAB toolbox functions.
libmcc.dll	MATLAB Compiler Library for stand-alone external applications. Contains the mcc and mcm routines required for building stand-alone external applications.
libmatlb.dll	MATLAB Math Built-In Library. Contains callable versions of MATLAB built-in math functions and operators. Required for building stand-alone external 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`	Switches 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`	Switches and settings for C and C++ compilers to create stand-alone applications, e.g., `msvccomp.bat` for use with Microsoft Visual C and `msvccompp.bat` for use with Microsoft Visual 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++ external applications.

`libmatpb.lib`	MATLAB C++ Math Library for Borland compiler.

`libmatpm.lib`	MATLAB C++ Math Library for Microsoft Visual C++ (MSVC) compiler.
`libmatpw.lib`	MATLAB C++ Math Library for Watcom compiler.

<matlab>\extern\include

The `<matlab>\extern\include` directory contains the header files that come with MATLAB-based, software development products.

The header file for the MATLAB Compiler is:

`mcc.h`	Header file for MATLAB Compiler Library.

In addition, the header file for the MATLAB C Math Library, a separate product is:

`matlab.h`	Header file for MATLAB Math Library.

The relevant header files from MATLAB are:

`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 external applications.
`libmatlb.def` `_libmatlb.def`	Contains names of functions exported from the MATLAB Math Built-In Library.
`libmccmx.def` `_libmccmx.def`	Contains names of functions exported from `libmccmx`.
`libmx.def` `_libmx.def`	Contains names of functions exported from `libmx.dll`.
`libut.def` `_libut.def`	Contains names of functions exported from `libut.dll`.

<matlab>\extern\include\cpp

The header file for the MATLAB C++ Math Library, a separate product, is:

`matlab.hpp`	Header file for MATLAB C++ Math Library.

<matlab>\extern\src\math\tbxsrc

The <matlab>\extern\src\math\tbxsrc directory contains the MATLAB Compiler-compatible M-files. These files are:

automesh.m	base2dec.m	bin2dec.m
corrcoef.m	cov.m	datestr.m
datevec.m	fmin.m	fmins.m
fzero.m	gradient.m	griddata.m
hex2dec.m	hex2num.m	interp1q.m
interp2.m	interp4.m	interp5.m
interp6.m	ismember.m	ntrp23.m
ntrp23s.m	ntrp45.m	numjac.m
ode113.m	ode15s.m	ode23.m
odeset.m	odezero.m	polyeig.m
polyfit.m	polyval.m	quad.m
quad8.m	str2num.m	strcat.c
strvcat.c		

<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 8-35).
fibocert.m	M-file that explores assertions (page 4-15).

`fibocon.m`	M-file used to show MEX-file source code (page 6-4).
`fibomult.m`	M-file that explores helper functions (page 4-21).
`hello.m`	M-file that displays `Hello, World`.
`houdini.m`	Script M-file that cubes each element of a 2-by-2 matrix (page 3-12).
`initprnt.m`	Dummy M-file (page 5-41).
`lu2.m`	M-file example that benefits from `%#ivdep` (page 8-9).
`README`	Short description of each file in this directory.
`main.m`	M-file “main program” that calls `mrank` (page 5-31).
`mrank.m`	M-file to calculate the rank of magic squares (page 5-31).
`mrankmac.c`	Macintosh version of `mrankp.c`.
`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 5-48).
`mrankwin.c`	Windows version of `mrankp.c`.
`multarg.m`	M-file that contains two input and two output arguments (page 5-49).
`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 5-49).

`mycb.m`	M-file example used to study callbacks (page 4-21).
`mydep.m`	M-file example used to show a case when `%#ivdep` generates incorrect results (page 8-9).
`myfunc.m`	M-file that explores helper functions (page 4-14).
`myph.c`	C function print handler initialization (page 5-40).
`mypoly.m`	M-file example used to study callbacks (page 4-23).
`novector.m`	M-file example that demonstrates the influence of vectorization (page 4-29).
`plot1.m`	M-file example that calls `feval` (page 4-25).
`plotf.m`	M-file example that calls `feval` multiple times (page 4-25).
`squares1.m`	M-file example that does not preallocate a matrix (page 4-27).
`squares2.m`	M-file example that preallocates a matrix (page 4-27).
`squibo.m`	M-file to calculate “squibonacci” numbers. Compile `squibo.m` into a MEX-file (page 3-3).
`squibo2.m`	M-file example that contains a `%#realonly` pragma (page 8-11).
`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 4-29).

<matlab>\toolbox\compiler

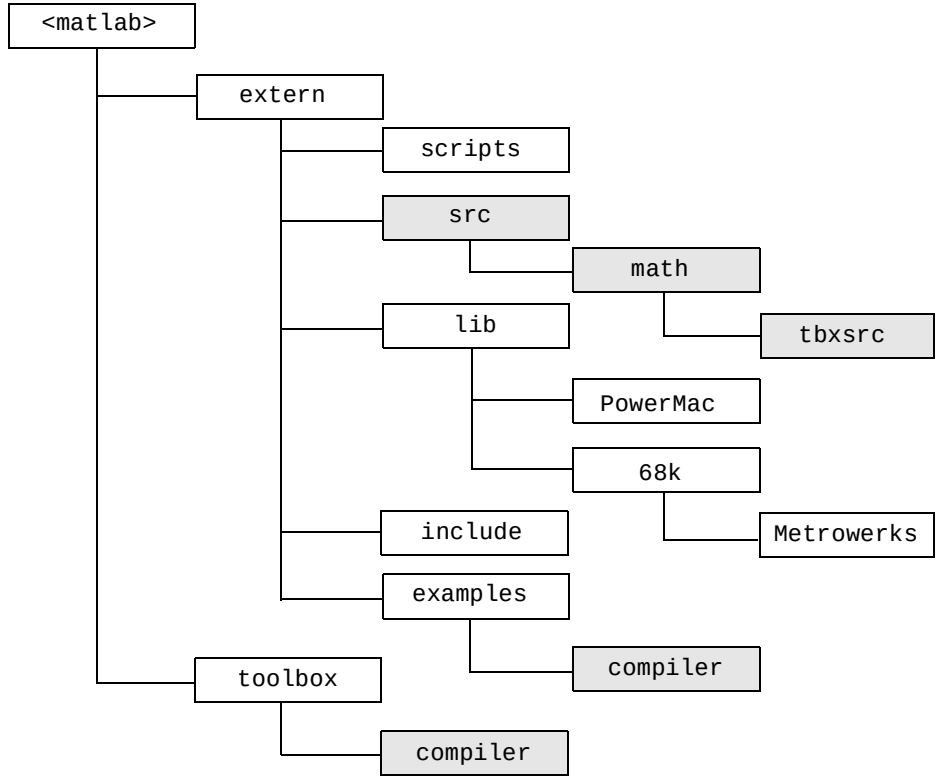
The <matlab>\toolbox\compiler directory contains the MATLAB Compiler's additions to the MATLAB command set:

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	Builds stand-alone applications from MATLAB command prompt.
mbvector.m	Assert variable is vector.
mcc.m	Invoke the MATLAB Compiler.
mccexec.mex	MATLAB Compiler internal routine.
mccload.mex	MATLAB Compiler internal routine.
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 Macintosh

Installation of the MATLAB Compiler places many new files into folders already used by MATLAB. In addition, installing the MATLAB Compiler creates several new folders. This figure illustrates the folders in which the MATLAB Compiler files are located.



MATLAB installation creates unshaded folders.

MATLAB Compiler installation creates shaded folders.

In the illustration, `<matlab>` symbolizes the top-level folder where MATLAB is installed on your system.

<matlab>

The <matlab> folder, in addition to containing many other folders, can contain one MATLAB Compiler document, which is:

Compiler_Readme	Optional document that describes configuration details, known problems, workarounds, and other useful information.

<matlab>:extern:scripts:

The <matlab>:extern:scripts: folder contains:

mbuild	Script that controls the building and linking of your code.
mex	MPW script that generates MEX-files from input C source code.
Various mbuildopts.* files	Auxiliary scripts for mbuild and mbuild.m.
Various mexopts.* files	Auxiliary scripts for mex and mex.m.

<matlab>:extern:src:math:tbxsrc:

The <matlab>:extern:src:math:tbxsrc: folder contains the MATLAB Compiler-compatible M-files. These files are:

automesh.m	base2dec.m	bin2dec.m
corrcoef.m	cov.m	datestr.m
datevec.m	fmin.m	fmins.m
fzero.m	gradient.m	griddata.m
hex2dec.m	hex2num.m	interp1q.m
interp2.m	interp4.m	interp5.m

interp6.m	ismember.m	ntrp23.m
ntrp23s.m	ntrp45.m	numjac.m
ode113.m	ode15s.m	ode23.m
odeset.m	odezero.m	polyeig.m
polyfit.m	polyval.m	quad.m
quad8.m	str2num.m	strcat.c
strvcat.c		

<matlab>:extern:lib:PowerMac:

The <matlab>:extern:lib:PowerMac: folder contains the required libraries for MPW and Metrowerks programmers.

The files for the MATLAB Compiler are:

libmccmx	MATLAB Compiler Library for MEX-files. Contains the mcc and mcm routines required for building MEX-files. This is a shared library.
libmat	MATLAB MAT-file access library. Shared library, installed with MATLAB.
libmi	Used by libmx. Shared library, installed with MATLAB.
libmx	MATLAB Application Program Interface Library. Contains the array access routines. Shared library, installed with MATLAB.
libut	MATLAB Utilities Library. Contains the utility routines used by various components in the background. Shared library, installed with MATLAB.

The libraries for the MATLAB C Math Library, a separate product, are:

`libmmfile`	MATLAB M-File Math Library. Contains callable versions of the M-files in the MATLAB Toolbox. Needed for building stand-alone external applications that require MATLAB toolbox functions. This is a shared library.
`libmcc`	MATLAB Compiler Library for stand-alone external applications. Contains the `mcc` and `mcm` routines required for building stand-alone external applications. This is a shared library.
`libmatlb`	MATLAB Math Built-In Library. Contains callable versions of MATLAB built-in math functions and operators. Required for building stand-alone external applications. This is a shared library.

<matlab>:extern:lib:68k:Metrowerks:

The `<matlab>:extern:lib:68k:Metrowerks:` folder contains the required libraries for Metrowerks programmers working on Motorola 680x0 platforms. The libraries are:

`libmccmx.lib`	MATLAB Compiler Library for MEX-files. Contains the `mcc` and `mcm` routines required for building MEX-files.
`libmat.lib`	MATLAB MAT-file access library, installed with MATLAB.
`libmi.lib`	Used by `libmx`, installed with MATLAB.

libmx.lib	MATLAB Application Program Interface Library, which contains the array access routines.
libut.lib	MATLAB Utilities Library, which contains the utility routines used by various components in the background.

The libraries for the MATLAB C Math Library, a separate product, are:

libmmfile.lib	MATLAB M-File Math Library. Contains callable versions of the M-files in the MATLAB Toolbox. Needed for building stand-alone external applications that require MATLAB toolbox functions.
libmcc.lib	MATLAB Compiler Library for stand-alone external applications. Contains the mcc and mcm routines required for building stand-alone external applications.
libmatlb.lib	MATLAB Math Built-In Library. Contains callable versions of MATLAB built-in math functions and operators. Required for building stand-alone external applications.

<matlab>:extern:include:

The <matlab>:extern:include folder contains the header files for developing C applications that interface to MATLAB.

The header file for the MATLAB Compiler is:

mcc.h	Header file for MATLAB Compiler Library.

The header file for the MATLAB C Math Library, a separate product, is:

`matlab.h`	Header file for MATLAB Math Built-In Library and MATLAB M-File Math Library.

The relevant header files for MATLAB are:

`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.

<matlab>:extern:examples:compiler:

The `<matlab>:extern:examples:compiler:` folder holds the sample M-files and C functions described in this book. For some examples, the online version may differ from the version printed in this book. In these cases, assume that the online versions are correct.

`earth.m`	M-file that accesses a global variable (page 8-35).
`fibocert.m`	M-file that explores assertions (page 4-15).
`fibocon.m`	M-file used to show MEX-file source code (page 6-4).
`fibomult.m`	M-file that explores helper functions (page 4-21).
`hello.m`	M-file that displays `Hello, World`.

houdini.m	Script M-file that cubes each element of a 2-by-2 matrix (page 3-12).
initprnt.m	Dummy M-file (page 5-41).
lu2.m	M-file example that benefits from %#ivdep (page 8-9).
main.m	M-file "main program" that calls mrank (page 5-31).
mrank.m	M-file to calculate the rank of magic squares (page 5-31).
mrankmac.c	Macintosh version of mrankp.c.
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 5-48).
mrankwin.c	Windows version of mrankp.c.
multarg.m	M-file that contains two input and two output arguments (page 5-49).
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 5-49).
mycb.m	M-file example used to study callbacks (page 4-21).
mydep.m	M-file example used to show a case when %#ivdep generates incorrect results (page 8-9).
myfunc.m	M-file that explores helper functions (page 4-14).

myph.c	C function print handler initialization (page 5-40).
mypoly.m	M-file example used to study callbacks (page 4-23).
novector.m	M-file example that demonstrates the influence of vectorization (page 4-29).
plot1.m	M-file example that calls feval (page 4-25).
plotf.m	M-file example that calls feval multiple times (page 4-25).
squares1.m	M-file example that does not preallocate a matrix (page 4-27).
squares2.m	M-file example that preallocates a matrix (page 4-27).
squibo.m	M-file to calculate “squibonacci” numbers. Compile squibo.m into a MEX-file (page 3-3).
squibo2.m	M-file example that contains a %#realonly pragma (page 8-11).
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 4-29).

<matlab>:toolbox:compiler:

The <matlab>:toolbox:compiler: folder 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	Builds stand-alone applications from MATLAB command prompt.
mbvector.m	Assert variable is vector.
mcc.m	Invoke the MATLAB Compiler.
mccexec.mex	MATLAB Compiler internal routine.
mccload.mex	MATLAB Compiler internal routine.
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.

8

Reference

Introduction

This chapter contains reference pages for all the pragmas, assertions, and real-only functions that come with the MATLAB Compiler. This chapter also contains reference pages for the MATLAB Compiler command (`mcc`) and `mbuild`.

Pragmas	
`%#inbounds`	Inbounds pragma.
`%#ivdep`	Ignore vector dependencies pragma.
`%#realonly`	Real-only pragma.

Assertions	
`mbchar`	Assert variable is a character string.
`mbcharscalar`	Assert variable is a character scalar.
`mbcharvector`	Assert variable is a character vector.
`mbint`	Assert variable is an integer.
`mbintscalar`	Assert variable is an integer scalar.
`mbintvector`	Assert variable is an integer vector.
`mbreal`	Assert variable is real.
`mbrealscalar`	Assert variable is a real scalar.
`mbrealvector`	Assert variable is a real vector.
`mbscalar`	Assert variable is a scalar.
`mbvector`	Assert variable is a vector.

Note: The MATLAB Compiler treats assertion functions (i.e., the functions starting with "`mb`") as type declarations; they are not used for type verification.

Real-Only Functions	
`reallog`	Natural logarithm for nonnegative real inputs.
`realpow`	Array power function for real-only output.
`realsqrt`	Square root for nonnegative real inputs.

MATLAB Compiler Options for C++

If you use the MATLAB Compiler `mcc` option flag `-p` to generate C++ code, then only the following other option flags apply:

- `-l`
- `-m`
- `-v`
- `-w`

Other option flags, such as `-i`, do not apply to C++ generated code. The following pragmas are also ignored:

- `%#inbounds` (corresponds to MATLAB Compiler `-i` option flag)
- `%#ivdep`
- `%#realonly` (corresponds to MATLAB Compiler `-r` option flag)

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. If you do not tell the Compiler which functions will be called through `feval` in stand-alone mode (`-e/m/p`) and the functions are not contained in the C/C++ Math Library, the Compiler will produce a runtime error.

If you are using the `%#function` pragma to define functions that are not available in M-code, you must write a dummy M-file 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-file that contains a function line, such as:

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

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-file.

In stand-alone C mode, any function that will be called with `feval` must be a separately compiled function with the standard `mlfxxx()` interface. The function can be written by hand in C code or generated using the MATLAB Compiler.

In stand-alone C++ mode, any function can be called through `feval`, but the function must be mentioned with the pragma.

%#inbounds

Purpose

Inbounds pragma.

Syntax

```
%#inbounds
```

Description

This pragma has no effect on C++ generated code (i.e., if the `-p` MATLAB Compiler option flag is used).

`%#inbounds` is the pragma version of the MATLAB Compiler option flag `-i`. Placing the pragma

```
%#inbounds
```

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 alpha beta
```

the `%#inbounds` pragma in `alpha.m` has no influence on the way the MATLAB Compiler compiles `beta.m`.

See Also

`mcc` (the `-i` option), `%#realonly`

%#ivdep

Purpose

Ignore-vector-dependencies (ivdep) pragma.

Syntax

```
%#ivdep
```

Description

This pragma has no effect on C++ generated code (i.e., if the `-p` MATLAB Compiler option flag is used).

The `%#ivdep` pragma tells the MATLAB Compiler to ignore vector dependencies in the assignment statement that immediately follows it. 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 8-1 shows that the `%#ivdep` pragma had a significant impact on the performance of `lu2`.

Table 8-1: Performance for lu2(magic(100)), run 20 times

MEX-File	Elapsed Time (in sec.)
`lu2` containing `%#ivdep`	1.8610
`lu2` omitting `%#ivdep`	2.6102

Purpose

Real-only pragma.

Syntax

```
%#realonly
```

Description

This pragma has no effect on C++ generated code (i.e., if the `-p` MATLAB Compiler option flag is used).

`%#realonly` is the pragma version of the MATLAB Compiler option flag `-r`. Placing this pragma

```
%#realonly
```

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.

Example

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 squibo2
```

An alternative way of ending up with the same code is to omit the `%#realonly` pragma and to compile with

```
mcc -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 alpha beta
```

the `%#realonly` pragma in `alpha.m` has no influence on the way the MATLAB Compiler compiles `beta.m`.

See Also

`mcc` (the `–r` option flag), `%#inbounds`

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 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

mbcharscalar

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

```
n = ['hello' 'world'];
mbcharscalar(n)
??? Error using ==> mbcharscalar
argument of mbcharscalar must be scalar
```

See Also

`mbchar`, `mbcharvector`, `mbreal`, `mbscalar`, `mbvector`, `mbintscalar`, `mbintvector`

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 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

mbint

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 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`

mbintscalar

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 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`

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 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`

mbreal

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 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`

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 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`

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 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`

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 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

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 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`

Purpose

Create an application using the MATLAB Math Library.

Syntax

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

Description

`mbuild` is a script that supports various switches that allow you to customize the building and linking of your code. The only required option that all users must execute is `setup`; the other options are provided for users who want to customize the process. The `mbuild` syntax and options are:

Option	Description
`-c`	Compile only; do not link.
`-D<name>[=<def>]`	(UNIX and Macintosh) 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 path name if it is not in current directory. (Not necessary if you use the `-setup` option.)
`-f <file>`	(Macintosh) Use `<file>` as the options file. (Not necessary if you use the `-setup` option.) If `<file>` is specified, it is used as the options file. If `<file>` is not specified and there is a file called `mbuildopts` in the current directory, it is used as the options file. If `<file>` is not specified and `mbuildopts` is not in the current directory and the file `mbuildopts` is in the directory `<matlab>:extern:scripts:`, it is used as the options file. Otherwise, an error occurs.

Option	Description
-F <file>	(UNIX) Use <file> as the options file. (Not necessary if you use the -setup option.) <file> is searched for in the following manner: The file that occurs first in this list is used: • ./<filename> • $HOME/matlab/<filename> • $TMW_ROOT/bin/<filename>
-F <file>	(Windows) Use <file> as the options file. (Not necessary if you use the -setup option.) <file> is searched for in the current directory first and then in the same directory as mbuild.bat.
-g	Build an executable with debugging symbols included.
-h[elp]	Help; prints a description of mbuild and the list of options.
-I<pathname>	Include <pathname> in 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.
<name>=<def>	(UNIX and Macintosh) Override options file setting for variable <name>.
-n	No execute flag. Using this option causes the commands used to compile and link the target to be displayed without executing them.
-output <name>	Create an executable named <name>. (An appropriate executable extension is automatically appended.)
-O	Build an optimized executable.

Option	Description
-setup	Set up default options file. This switch should be the only argument passed.
-U<name>	(UNIX and Windows) Undefine C preprocessor macro <name>.
-v	Verbose; print all compiler and linker settings.

mcc

Purpose

Invoke MATLAB Compiler.

Syntax

```
mcc [-options] mfile1      [mfile2 ... mfileN]
                           [fun1=feval_arg1 ... funN=feval_argN]
```

Description

`mcc` is the MATLAB command that invokes the MATLAB Compiler. You can issue the `mcc` command only from the MATLAB interpreter prompt. The only 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 myfun
```

If you specify a relative pathname for an M-file, the M-file must be in a directory or folder on your MATLAB search path, or in your current directory or folder.

You may specify one or more MATLAB Compiler option flags to `mcc`. (The list of option flags appears later in this reference page.) Each option flag has 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 -ir myfun
mcc -i -r myfun
```

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 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 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`.

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 citrus fun1=lemon    % correct
mcc citrus fun1 = lemon  % incorrect
```

MATLAB Compiler 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 only the following other option flags apply:

- `-l`
- `-m`
- `-v`
- `-w`

Other option flags, such as `-i`, do not apply to C++ generated code.

-c (C Code Only)

Generate C code but do not invoke `mex` or `mbuild`, i.e., do not produce a MEX-file.

Note: `-c` and `-e` do *not* do the same thing. The C code generated by `-e` can be linked into a stand-alone external application; the C code generated by `-c` cannot.

-e (Stand-Alone External C Code)

Generate C code for stand-alone external applications. The resulting C code cannot be used to create MEX-files. Stand-alone external applications do not require MATLAB at runtime. Stand-alone external applications can run even if MATLAB is not installed on the system. See Chapter 5 for complete details on stand-alone external applications.

If you specify `-e`, the MATLAB Compiler does not invoke `mex`, consequently, it does not produce a MEX-file.

-f <filename> (Specifying Options File)

Use the specified options file when calling `mex`. 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` script. See the *Application Program Interface Guide* for more information about using this option with the `mex` script.

Notes: This option is only available in MEX-mode.

Although this option works as documented, it is suggested that you use `mex -setup` to switch compilers.

-g (Debugging Information)

Cause `mex` to invoke the C compiler with the appropriate C compiler options for debugging. You should specify `-g` if you want to debug the MEX-file 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 compiler generates.

If you specify –g on the same command line as –e, the MATLAB Compiler ignores –g.

Note: This option flag has no effect on C++ generated code, i.e., if the –p MATLAB Compiler option flag is used.

-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 minimize_it fmins
```

instead of

```
mcc –h minimize_it
```

Note: Due to MATLAB Compiler restrictions, some of the V5 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/math/tbxsrc, where <matlab> represents the top-level directory where MATLAB is installed on your system.

-i (Inbounds Code)

Generate 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 only specify `-i` 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 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 Windows/NT. 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 switch does not work well for mathematical M-files. It works best for command line M-files such as dir and type.

-p (Stand-Alone External C++ Code)

Generate C++ code for stand-alone external applications. The resulting C code cannot be used to create MEX-files. Stand-alone external applications do not require MATLAB at runtime. Stand-alone external applications can run even if MATLAB is not installed on the system. See Chapter 5 for complete details on stand-alone external 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 probably crash.

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.

The `-s` option flag does not influence stand-alone external applications.

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.

-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.

-v (Verbose)

Display the steps in compilation, including:

- The command that is invoked
- The compiler version number
- The invocation of mex

The –v flag passes the –v flag to mex and displays information about mex.

-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.

On UNIX systems, the MATLAB Compiler passes –w to mex, causing UNIX C compilers to generate warning messages where applicable.

-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 gazelle
```

Optimize the performance of gazelle.mex by compiling with two optimization option flags:

```
mcc –ri gazelle
```

Compile two M-files (gazelle.m and cheetah.m) to create one MEX-file (gazelle.mex):

```
mcc gazelle cheetah
```

Compile gazelle.m to create gazelle.c (C source code for a stand-alone external application):

```
mcc –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 myfun corresponds to fun1:

```
mcc 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 -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 -q -h myfun
```

See Also `%#inbounds`, `%#ivdep`, `mbint`, `mbreal`, `mbscalar`, `mbvector`, `%#realonly`, `reallog`, `realpow`, `realsqrt`

Simulink-Specific Options

These options let you generate complete S-functions that are compatible with the Simulink S-function block.

-S (Simulink S-Function)

Output an 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.

-u (Number of Inputs) and -y (Number of Outputs)

Allow you to exercise more control over the number of valid inputs or outputs for your function. 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.

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`

realpow

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`

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`

9

MATLAB Compiler Library

Introduction

This chapter lists the functions in the MATLAB Compiler Library and provides a brief description of what they do. The functions fall into these categories:

- Functions that implement MATLAB built-in functions.
- Functions that implement MATLAB operators.
- Low-level math functions.
- Output functions.
- Functions that manipulate the `mxArray` type.
- Miscellaneous functions.

There are actually two different MATLAB Compiler Libraries:

- The MATLAB Compiler Library for MEX-files (`libmccmx`).
- The MATLAB Compiler Library for stand-alone external applications (`libmcc`).

Both versions of the library contain the same list of routines. The header file for either library is `mcc.h`. However, despite the similarities, the routines in `libmccmx` do not always produce the same results as their counterparts in `libmcc`. This is because the routines in each library are implemented somewhat differently.

The function prototypes for these routines will change at the next release of the MATLAB Compiler. In addition, some of these routines will become obsolete at the next release.

Note: This chapter is for informational purposes only, to give you a clearer understanding of the code that the MATLAB Compiler generates. You should *not* write C or C++ code that calls these functions; just let the MATLAB Compiler call these functions.

Functions That Implement MATLAB Built-In Functions	
`void mccAbs( mxArray *ppp, mxArray *q )`	Implements `abs` function.
`void mccAcos( mxArray *p, mxArray *q )`	Implements `acos` function (currently real only).
`void mccAll( mxArray *p, mxArray *q )`	Implements `all` function.
`void mccAny( mxArray *p, mxArray *q )`	Implements `any` function.
`void mccAsin( mxArray *p, mxArray *q )`	Implements `asin` function (currently real only).
`void mccAtan( mxArray *p, mxArray *q )`	Implements `atan` function (currently real only).
`void mccAtan2( mxArray *p, mxArray *q, mxArray *r )`	Implements `atan2` function (currently real only).
`int mccBoolAll( mxArray *q )`	Implements `all` for vectors.
`int mccBoolAny( mxArray *q )`	Implements `any` for vectors.
`void mccCeil( mxArray *p, mxArray *q )`	Implements `ceil` function.
`void mccCos( mxArray *p, mxArray *q )`	Implements `cos` function (currently real only).
`void mccError( mxArray *p )`	Implements `error` function.
`void mccEval( mxArray *p )`	Issues an error message (via stub `eval` function) and then exits.

Functions That Implement MATLAB Built-In Functions (Continued)	
`void mccFindstr( mxArray *p, mxArray *q, mxArray *r)`	Implements `findstr` function. This function works with complex arguments as well as usual strings. If the inputs are not vectors, the function sometimes returns `[]` where the usual M-file returns an error message.
`void mccFix( mxArray *p, mxArray *q )`	Implements `fix` function.
`void mccFloor( mxArray *p, mxArray *q )`	Implements `floor` function.
`int mccGetDimensionSize( mxArray *p, int nn )`	Internal version of `a = size(b,n)`.
`int mccGetLength( mxArray *p )`	Internal version of `a = length(b)`.
`void mccGetMatrixSize( int *pm, int *pn, mxArray *p )`	Internal version of `[m,n] = size(a)`.
`void mccImag( mxArray *p, mxArray *q )`	Implements `imag` of a matrix.
`int mccIsEmpty( mxArray *p )`	Internal version of `a = isempty(b)`.
`void mccLog( mxArray *p, mxArray *q )`	Implements `log` function (currently real only).
`void mccLog10( mxArray *p, mxArray *q )`	Implements `log10` function (currently real only).
`void mccLower( mxArray *p, mxArray *q )`	Implements `lower`.
`void mccMax( mxArray *p, mxArray *q )`	Implements `max` of a matrix (currently real only).

Functions That Implement MATLAB Built-In Functions (Continued)	
`void mccMin( mxArray *p, mxArray *q )`	Implements `min` of a matrix (currently real only).
`void mccOnes( mxArray *p, mxArray *q )`	Implements `ones(a)`.
`void mccOnesMN( mxArray *p, int m, int n )`	Implements `ones(m,n)`.
`double mccRealVectorMax( mxArray *p )`	Implements `max` of a real vector.
`double mccRealVectorMin( mxArray *p )`	Implements `min` of a vector of reals.
`double mccRealVectorProduct( mxArray *p )`	Implements scalar product of a real vector.
`double mccRealVectorSum( mxArray *p )`	Implements `sum` on real vector, returning scalar result.
`void mccReshape( mxArray *p, mxArray *q, int m, int n )`	Implements `reshape` function.
`void mccReshape2( mxArray *p, mxArray *q, mxArray *r )`	Implements two-argument `reshape` function.
`double mccRint( double x )`	Rounds array index expression to integer.
`void mccRound( mxArray *p, mxArray *q )`	Implements `round` function.
`void mccSign( mxArray *p, mxArray *q )`	Implements `sign` function (currently real only).
`void mccSin( mxArray *p, mxArray *q )`	Implements `sin` function (currently real only).
`void mccSize( mxArray *p, mxArray *q )`	Internal version of `a = size(b)`.

Functions That Implement MATLAB Built-In Functions (Continued)	
`void mccSqrt( mxArray *p, mxArray *q )`	Implements `sqrt` function (currently real only).
`int mccStrcmp( mxArray *p, mxArray *q )`	Implements `strcmp` function. This compares complex parts as well, like the interpreter version.
`void mccSum( mxArray *p, mxArray *q )`	Implements `sum` of a matrix (currently real only).
`void mccTan( mxArray *p, mxArray *q )`	Implements `tan` function (currently real only).
`void mccUpper( mxArray *p, mxArray *q )`	Implements `upper`.
`void mccVectorProduct( double *pr, double *pi, mxArray *p )`	Implements scalar product of a complex vector.
`void mccVectorSum( double *pr, double *pi, mxArray *p )`	Implements `sum` on complex vector, returning scalar result.
`void mccZerosMN( mxArray *p, int m, int n )`	Implements `zeros(m,n)`.
`double mcmPi( )`	Function returning value of `pi`.

Functions That Implement MATLAB Operators	
`void mccArrayLeftDivide( mxArray *p, mxArray *q, mxArray *r )`	Implements complex array left division.
`void mccArrayPower( mxArray *p, mxArray *q, mxArray *r )`	Implements complex array power.

Functions That Implement MATLAB Operators (Continued)	
`void mccArrayRightDivide( mxArray *p, mxArray *q, mxArray *r )`	Implements complex array right division.
`void mccColon( mxArray *p, double lo, double step, double hi )`	Creates `p` equal to `lo:step:hi`.
`void mccColon2( mxArray *p, double lo, double hi )`	Creates `p` equal to `lo:hi`.
`void mccColonOnLhs( mxArray *p, int mn )`	Checks the sizes of an assignment to `a()`.
`void mccConjTrans( mxArray *ppp, mxArray *q )`	Implements conjugate transpose operator.
`void mccCopy( mxArray *p, mxArray *q )`	Copies matrix `q` into matrix `p`.
`void mccInnerProduct( double *re, double *im, mxArray *p, mxArray *q )`	Complex inner product of two complex vectors.
`void mccIntColon( mxArray *p, int ilo, int step, double hi )`	Implements `:` (colon) operator for integer scalars.
`int mccIntVectorMax( mxArray *p )`	Implements `max` of vector of integers.
`int mccIntVectorMin( mxArray *p )`	Implements `min` of vector of integers.
`void mccLeftDivide( mxArray *p, mxArray *q, mxArray *r )`	Implements complex matrix left division.
`void mccMatrixColon( mxArray *p, mxArray *q, mxArray *r )`	Version of `colon` that accepts complex matrices.

Functions That Implement MATLAB Operators (Continued)	
`void mccMatrixColon2( mxArray *p, mxArray *q, mxArray *r )`	Version of `colon2` that accepts complex matrices.
`void mccMatrixExpand( mxArray *matrix, double re, double im )`	Implements replacing an entire matrix with a scalar value; e.g., `x(:) = N`.
`void mccMultiply( mxArray *ppp, mxArray *q, mxArray *r )`	Implements complex matrix multiply.
`void mccPower( mxArray *p, mxArray *q, mxArray *r )`	Implements complex matrix power.
`void mccReal( mxArray *p, mxArray *q )`	Returns real part of a matrix.
`void mccRealArrayLeftDivide( mxArray *p, mxArray *q, mxArray *r )`	Implements real array left division.
`void mccRealArrayRightDivide( mxArray *p, mxArray *q, mxArray *r )`	Implements real array right division.
`double mccRealInnerProduct( mxArray *p, mxArray *q )`	Inner product of two real vectors.
`void mccRealLeftDivide( mxArray *p, mxArray *q, mxArray *r )`	Implements real matrix left division.
`void mccRealMatrixMultiply( mxArray *ppp, mxArray *q, mxArray *r )`	Implements real matrix multiply.
`void mccRealPower( mxArray *p, mxArray *q, mxArray *r )`	Implements array power (`.^`) (currently real only).
`void mccRealRightDivide( mxArray *p, mxArray *q, mxArray *r )`	Implements real matrix right division.

Functions That Implement MATLAB Operators (Continued)	
`void mccRightDivide( mxArray *p, mxArray *q, mxArray *r )`	Implements complex matrix right division.
`void mccTrans( mxArray *ppp, mxArray *q )`	Implements transpose operator.

Low-Level Math Functions	
`int mcmColonCount( double lo, double step, double hi )`	Number of elements in 3-argument colon expression.
`int mcmColonMax( double lo, double step, double hi )`	Largest element in 3-argument colon expression.
`void mcmComplexRound( double *pr, double *pi, double qr, double qi )`	Complex round function.
`void mcmDivide( double *cr, double *ci, double ar, double ai, double br, double bi )`	Complex divide routine.
`double mcmDivideImagPart( double ar, double ai, double br, double bi )`	Imaginary part of a complex division.
`double mcmDivideRealpart( double ar, double ai, double br, double bi )`	Real part of a complex division.
`double mcmEps( )`	Function returning value of eps.
`int mcmFix( double d )`	fix function.

Low-Level Math Functions (Continued)	
`double mcmHypot( double re, double im )`	`hypot` function.
`int mcmIntMax( int m, int n )`	`max` of two integers.
`int mcmIntMin( int m, int n )`	`min` of two integers.
`int mcmIntSign( int n )`	`sign` function (integer argument).
`void mcmLog( double *par, double *pai, double br, double bi )`	Complex logarithm.
`double mcmLog10( double d )`	`log10` function (real argument only).
`double mcmMax( double m, double n )`	`max` of two reals.
`double mcmMin( double m, double n )`	`min` of two reals.
`void mcmPower( double *par, double *pai, double br, double bi, double cr, double ci)`	Compiler complex scalar power (from `mathlib/mlCpow.cpp`).
`double mccRealmax( )`	Gets `realmax` from a local copy after the first time.
`double mccRealmin( )`	Gets `realmin` from a local copy after the first time.
`double mcmRealPowerInt( double d, int n )`	Real raised to an integer power.

Low-Level Math Functions (Continued)	
`double mcmRealSign( double d )`	`sign` function (real argument).
`double mcmRound( double d )`	`round` function.

Output Functions	
`void mccPrint( mxArray *p, char *s )`	Implements printing when ; (semicolon) is left off in M-code.
`void mccPrintf( const char *format, … )`	Local version of `mexPrintf`.
`void mccPuts( char *s )`	Outputs a string.
`void mccUndefVariable( mxArray *p, mxArray *s )`	Issues error message about a use of a function argument not present in the call. The output variable is a dummy.
`void mccUndefVariable1( mxArray *s )`	Error routine called at runtime when an M-code variable is not defined.
`void mcmError( char *ss )`	Routine to print ss and stop, with an M-file line number if you compiled with the `-l` option.
`void mcmErrorWithLine( char *s, int line )`	Routine to print s and stop, supplying an M-file line number.
`void mcmFatal( char *ss )`	Routine to print an internal compiler runtime error.
`void mcmInternal( char *ss, int line )`	Routine to print ss, with an internal (C++ source file) line number as well as a user (M-file) line number. Used for debugging.

Functions That Manipulate the mxArray Type	
`void mccCreateConstant2DMatrix( mxArray *p, double re, double im, mxArray *q, mxArray *r)`	Expands the scalar `re+i*im` into matrix `p` whose dimensions are given by the matrices (of ones) `q` and `r`.
`void mccCreateConstantMatrix( mxArray *p, double re, double im, mxArray *q )`	Expands the scalar `re+i*im` into matrix `p` whose dimension is given by `q`.
`void mccCreateRealConstant2DMatrix( mxArray *p, double re, mxArray *q, mxArray *r )`	Expands the scalar `re` into a matrix `p` whose dimensions are given by the matrices (of ones) `q` and `r`.
`void mccCreateRealConstantMatrix( mxArray *p, double re, mxArray *q )`	Expands the scalar `re` into a matrix `p` whose dimension is given by `q`.
`void mccCreateRealScalar( mxArray *p, double re )`	Makes `p` a 1-by-1 matrix and store the value (`re`) in the (1,1) element.
`void mccCreateScalar( mxArray *p, double re, double im )`	Replaces `p` by a 1-by-1 matrix, and store (`re+i*im`) in the (1,1) element.
`void mccFixInternalMatrix( mxArray *matlab5_matrix, mxArray *compiler_matrix, int return_value )`	Translates a compiler matrix into a MATLAB matrix.
`void mccSetMatrixElement( mxArray *p, int i, int j, double re, double im )`	Stores the value (`re+i*im`) in the (`i`,`j`) element of `p`. Check the current size, and grow if necessary.
`void mccSetRealMatrixElement( mxArray *p, int i, int j, double re )`	Stores the real value (`re`) in the (`i`,`j`) element of `p`. Check the current size, and grow if necessary.

Functions That Manipulate the mxArray Type (Continued)	
`void mccSetRealVectorElement( mxArray *p, int i, double re )`	Stores the real value (`re`) in the `i` element of `p`. Check the current size, and grow if necessary.
`void mccSetVectorElement( mxArray *p, int i, double re, double im )`	Stores the value (`re+i*im`) in the `i` element of `p`. Check the current size, and grow if necessary.

Miscellaneous Functions	
`int mccAllocateMatrix( mxArray *p, int m, int n )`	Creates an m-by-n array, `p`. If the complex flag of `p` is on, the created matrix is complex. If the returned space is not zeroed (e.g., reuse of older space), the function returns 1; otherwise, the function returns 0.
`int mccArgc( )`	Gets the argument count for a stand-alone program.
`void mccArgv( mxArray *p, int n )`	Gets an argument for a stand-alone program.
`int mccCalcSubscriptDimensions( int mm, int *pn, int bm, int bn, mxArray *p )`	Calculates dimensions of `a(b)`.
`void mccCallMATLAB( int nlhs, mxArray **plhs, int nrhs, mxArray **prhs, char *s, int line )`	General callback into MATLAB to run an M-file or MEX-file.
`void mccCatenateColumns( mxArray *ppp, mxArray *a, mxArray *b )`	Implements `ppp = [a,b]`.

Miscellaneous Functions (Continued)	
`void mccCatenateRows( mxArray *ppp, mxArray *a, mxArray *b )`	Implements `pp = [a;b]`.
`void mccCheckMatrixSize( mxArray *p, int m, int n )`	Checks size of two-dimensional array assignment.
`void mccCheckVectorSize( mxArray *p, int mn )`	Checks size of one-dimensional array assignment.
`void mccCloseMATFile( )`	Closes and writes the MAT-file for save.
`void mccColExpand( mxArray *matrix, mxArray *cols, double re, double im )`	Replaces entire columns of an input matrix; e.g., `x(i,:) = 7`.
`void mccCreateEmpty( mxArray *p )`	Makes `p` into an empty matrix.
`void mccCreateString( mxArray *p, char *s )`	Copies the string `s` into the matrix `p`.
`void mccDEBUG( mxArray *p, char *ss )`	Prints the matrix passed as the first argument with a label given by the second. Intended for debugging (not all matrix elements printed).
`void mccFind( mxArray *p, mxArray *q )`	Implements `find` function.
`void mccFindIndex( mxArray *q, mxArray *p )`	`q` is set to `p` unless `p` is all 0's and 1's and its size matches `r`. In this case, `q` is set to `find(p)`.
`void mccFindScalar( mxArray *p, int bool )`	Used in indexing when performing logical scalar indexing.
`void mccForCol( mxArray *p, mxArray *q, int n )`	Generates column of data for a `for` statement.

Miscellaneous Functions (Continued)	
`void mccFreeMatrix( mxArray *p )`	Routine to free a matrix.
`void mccGetGlobal( mxArray *p, const char *name )`	Gets the value of global `name` and puts it in `p`.
`double mccGetImagMatrixElement( mxArray *p, int i, int j )`	Returns the imaginary part of element `(i,j)` of matrix `p`.
`double mccGetImagVectorElement( mxArray *p, int i )`	Returns the imaginary part of element `i` of matrix `p`.
`int mccGetMaxIndex( mxArray *p, int asz )`	Returns the maximum element in an index array `p`. Accounts for the possibility that the array is a 0-1 array.
`double mccGetRealMatrixElement( mxArray *p, int i, int j )`	Returns element `(i,j)` of the real part of matrix `p`.
`double mccGetRealVectorElement( mxArray *p, int i )`	Returns element `i` of the real part of matrix `p`.
`char * mccGetString( mxArray *p )`	Converts a compiler matrix into a C string (for use with `feval` only).
`void mccGrowMatrix( mxArray *p, int m, int n )`	Grows a two-dimensional array `p` to size m-by-n. Doesn't grow in place for arrays. However, if it looks like a one-dimensional array would work, call `mccGrowVector`.
`void mccGrowVector( mxArray *p, int mn )`	Grows a one-dimensional array `p` to size `mn`.
`int mccIfCondition( mxArray *p )`	Emulates MATLAB interpreter's behavior for `if`'s of arrays. Note this is subtly different from `all(all(p))`.

Miscellaneous Functions (Continued)	
`void mccImport( mxArray *p, const mxArray *q, int flg, int line )`	Imports a MATLAB matrix. Optionally free the MATLAB matrix.
`void mccImportCopy( mxArray *p, const mxArray *q, int flg, int line )`	Imports a MATLAB matrix, making a copy so MATLAB Compiler-generated code can change it. Optionally frees the MATLAB data structure.
`double mccImportReal( int *pflag, int *flags, const mxArray *q, char *ss )`	Similar to `mxGetScalar`, but checks that the value really is a real scalar. If the dimension is not 1-by-1 or there is a complex part, a fatal error is produced.
`double mccImportScalar( double *p, int *pflag, int *flags, const mxArray *q, char *ss )`	Similar to `mxGetScalar`, but checks that the value is a complex scalar. If the dimension is not 1-by-1 or there is a complex part, a fatal error occurs.
`int mccIsImag( mxArray *p )`	Returns 1 if matrix `p` has nontrivial imaginary part.
`void mccIsLetter( mxArray *p, mxArray *q )`	Implements `isletter` function.
`void mccLoad( mxArray *loaded_matrix, mxArray *name )`	`load` function (modified for the Compiler).
`void mccOpenMATFile( mxArray *name )`	Opens the MAT-file for save.
`void mccReturnFirstValue( mxArray **qin, mxArray *p )`	Returns the first argument of a function. Note that the semantics are slightly different for the first argument if the output argument has never been set.

Miscellaneous Functions (Continued)	
`void mccReturnScalar( mxArray **qin, double re, double im, mxArrayType kind, int flags )`	Creates and returns a scalar quantity. The flag describes various situations. If `mxSTRING` or `mccSET` is on, the variable has been set. The `mccCOMPLEX` bit is true if the result is to be complex (otherwise, `im` must be 0). The `mccNOTFIRST` bit is true if the result is not the first output argument. (This affects whether a null or an empty matrix is returned if the value was never set.) The `mccString` bit is true if the result is a string.
`void mccReturnValue( mxArray **qin, mxArray *p )`	Returns the second and later return values from a compiled function.
`void mccRowExpand( mxArray *matrix, mxArray *rows, double re, double im )`	Replaces entire rows of an input matrix; e.g., `x(:,i) = 7`.
`void mccSave( mxArray *name, mxArray *value )`	save function (modified for the compiler).
`void mccSetArgs( int argc, char **argv )`	Sets the `argc` and `argv` arguments.
`void mccSetGlobal( const char *name, mxArray *p )`	Sets the global `name` to the value in `p`.
`void mccSetIntMatrixElement( mxArray *p, int i, int j, int v )`	Assigns `int` to a matrix element.
`void mccSetIntVectorElement( mxArray *p, int i, int v )`	Assigns `int` to a vector element.
`mxArray * mccTempMATStr( char *s )`	Returns a temporary MATLAB matrix containing string `s`. This is good only for callbacks.

Miscellaneous Functions (Continued)	
`mxArray * mccTempMatrix( double re, double im, int cx, int flags )`	Returns a *very* temporary MATLAB matrix, good only for a callback.
`mxArray * mccTempMatrixElement( mxArray *p, double di, double dj )`	Converts one element of a matrix into a scalar matrix; preserve string flag.
`mxArray * mccTempVectorElement( mxArray *p, double di )`	Converts one element of a vector into a scalar matrix; preserve string flag.
`void mccZapColumns( mxArray *q, mxArray *p, mxArray *r )`	Removes the columns of `p` indicated by `r`. Put the result in `q`.
`void mccZapElements( mxArray *q, mxArray *p, mxArray *r )`	Removes the elements of `p` indicated by `r`. Put the result in `q`.
`void mccZapRows( mxArray *q, mxArray *p, mxArray *r )`	Removes the rows of `p` indicated by `r`. Put the result in `q`.
`void mccZeros( mxArray *p, mxArray *q )`	Implements `zeros(x)`, for a matrix `x`.
`void mccZerosCopyShape( mxArray *p, mxArray *q )`	Makes `p` a matrix of zeros of the same shape as `q`.
`int mcmCalcResultSize( int mm, int *pn, int m, int n )`	`mm` and `*pn` are the current size of a matrix result. A new matrix whose size is given by `m` and `n` is combined with the current size. `mcmCalcResultSize` returns the resulting value of `m`, and updates the value of `n` through `*pn`.
`void mcmCheck( int sz1, int sz2 )`	Checks the total size of an assignment.
`int mcmSetLineNumber(void)`	Sets the current line number (used for the `-l` switch).

Index

Symbols

A

B

C

D

E

L

M

O

P

Q

R

S

T

U

V

W

Y

Z