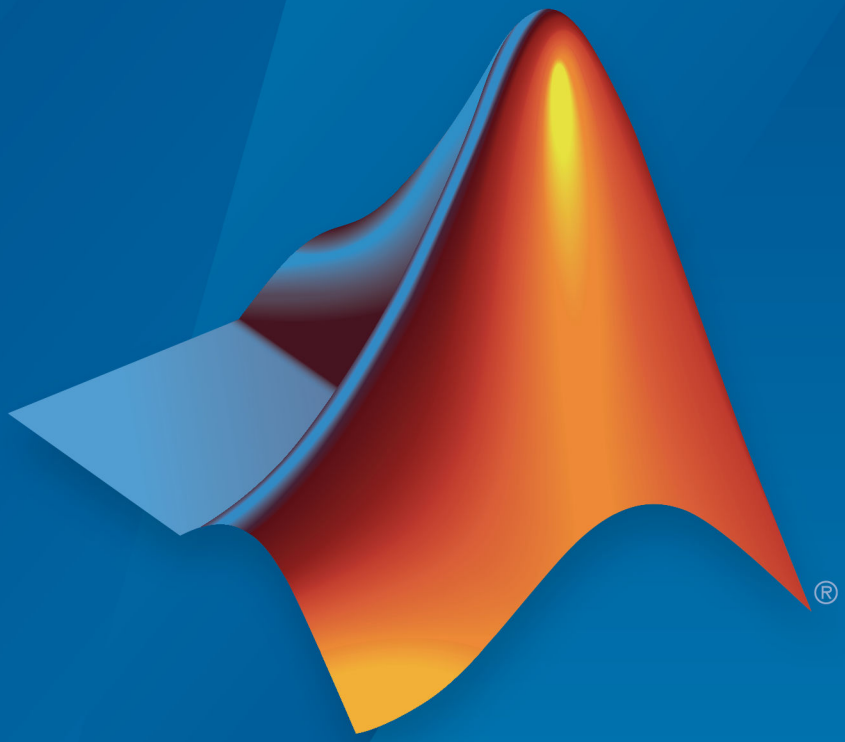
MATLAB® Compiler SDK™

C/C++ User's Guide



MATLAB®

R2017b

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

MATLAB® Compiler SDK™ C/C++ User's Guide

Revision History

March 2015	Online only	New for Version 6.0 (Release R2015a)
September 2015	Online only	Revised for Version 6.1 (Release 2015b)
October 2015	Online only	Rereleased for Version 6.0.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 6.2 (Release 2016a)
September 2016	Online only	Revised for Version 6.3 (Release R2016b)
March 2017	Online only	Revised for Version 6.3.1 (Release R2017a)
September 2017	Online only	Revised for Version 6.4 (Release R2017b)

Contents

1 Installation and Configuration

2 Libraries

3 Deployment Process

4 Distributing Code to an End User

5 Compiler Commands

6 Troubleshooting

7 Reference Information

8 Functions

A C++ Utility Library Reference

Installation and Configuration

Configure the mbuild Options File

The `mbuild` utility compiles and links applications that integrate MATLAB generated shared libraries. Its options file specifies the compiler and linker settings used to build the application.

By default, the `mbuild` utility selects the appropriate compiler using preset default configuration.

To change the options used by the `mbuild` utility:

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

   You can determine the path to the user preference folder using the MATLAB `prefdir` function.
2. Edit your copy of the options file to correspond to your specific needs, and save the modified file.

Solve Installation Problems

You can contact MathWorks:

- Via the website at `www.mathworks.com`. On the MathWorks home page, click **My Account** to access your MathWorks Account, and follow the instructions.
- Via email at `service@mathworks.com`.

Libraries

Integrate a C/C++ Shared Library into an Application

This example shows how to call a C++ shared library built with MATLAB Compiler SDK from a C++ application.

1. Create a C/C++ shared library using a MATLAB function. For more information, see "Create a C/C++ Shared Library with MATLAB Code".
2. Navigate to the `for_testing` folder created when you generated the C/C++ shared library.
3. Create a new file called `addmatrix_app.cpp`.
4. Copy the following C++ code into the file and save it. This is the application that calls the C/C++ shared library.

```
// Include the C++ shared library header
#include "addmatrix.h"

int run_main(int argc, char **argv)
{
  // Set up the application state for the MATLAB Runtime instance created in the ap
  if (!mclInitializeApplication(NULL,0))
    {
      std::cerr << "could not initialize the application properly"
                << std::endl;
          return -1;
    }

  // Load the required MATLAB code into the MATLAB Runtime.
  if( !addmatrixInitialize() )
    {
      std::cerr << "could not initialize the library properly"
                << std::endl;
         return -1;
    }

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

```
        // Create output array
        mwArray out;

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

        std::cout << "The value of added matrix is:" << std::endl;
        std::cout << out << std::endl;
      }

    // Catch the MATLAB generated mwException
    catch (const mwException& e)
      {
        std::cerr << e.what() << std::endl;
        return -2;
      }

    // Catch any other exceptions that may be thrown
    catch (...)
      {
        std::cerr << "Unexpected error thrown" << std::endl;
        return -3;
      }

    // Release the resources used by the generated MATLAB code
    addmatrixTerminate();

    // Release all state and resources used by the MATLAB Runtime for the application
    mclTerminateApplication();
    return 0;
}

int main()
{
    // Initialize the MATLAB Runtime
    mclmcrInitialize();

    // Create a new thread and run the MATLAB generated code in it.
    return mclRunMain((mclMainFcnType)run_main,0,NULL);
}
```

5 Use the system's command line to navigate to the `for_testing` folder where you created `addmatrix_app.cpp`.

6 Use `mbuild` at the system's command line to compile and link the application.

```
mbuild addmatrix_app.cpp addmatrix.lib
```

The `.lib` extension is for Windows®. On Mac the file extension will be `.dylib`, and on Linux® it will be `.so`.

7 From the system's command prompt, run the application.

```
addmatrix_app
The value of added matrix is:
     2     8    14
     4    10    16
     6    12    18
```

To follow up on this example:

- Try installing the new application on a different computer.
- Try building an installer for the application.
- Try integrating a shared library that consists of more than one function.

See Also

Call a Shared Library

To use a MATLAB Compiler SDK generated shared library in your application:

1. Include the generated header file for each library in your application.

   Each generated shared library has an associated header file named *libname*.h.
2. Initialize the MATLAB Runtime proxy layer by calling `mclmcrInitialize()`.
3. Use `mclRunMain()` to call the C function where your MATLAB functions are used.

   `mclRunMain()` provides a convenient cross platform mechanism for wrapping the execution of MATLAB code.

   Caution Do not use `mclRunMain()` if your application brings up its own full graphical environment.
4. Initialize the MATLAB Runtime and set the global settings by calling `mclInitializeApplication()` API function.

   Call the `mclInitializeApplication()` function once per application, and it must be called before calling any other MATLAB API functions. You may pass in application-level options to this function. `mclInitializeApplication()` returns a Boolean status code.
5. For each MATLAB Compiler SDK generated shared library that you include in your application, call the initialization function for the library.

   The initialization function performs library-local initialization. It unpacks the deployable archive and starts a MATLAB Runtime instance with the necessary information to execute the code in that archive. The library initialization function is named *libname*`Initialize()`. This function returns a Boolean status code.

   Note On Windows, if you want to have your shared library call a MATLAB shared library, the MATLAB library initialization function (e.g., `<libname>Initialize, <libname>Terminate, mclInitialize, mclTerminate`) cannot be called from your shared library during the `DllMain(DLL_ATTACH_PROCESS)` call. This applies whether the intermediate shared library is implicitly or explicitly loaded. Place the call somewhere after `DllMain()`.
6. Call the exported functions of each library as needed.
7. When your application no longer needs a given library, call the termination function for the library.

The terminate function frees the resources associated with the libraries MATLAB Runtime instance. The library termination function is named *libname*`Terminate()`. Once a library has been terminated, the functions exported by the library cannot be called again in the application.

8 When your application no longer needs to call any MATLAB Compiler SDK generated libraries, call the `mclTerminateApplication` API function.

This function frees application-level resources used by the MATLAB Runtime. Once you call this function, no further calls can be made to MATLAB Compiler SDK generated libraries in the application.

The following code example is from `matrixdriver.c`:

```
#include stdio.h

/* Include the MATLAB Runtime header file and the library specific header file
 * as generated by MATLAB Compiler SDK */

#include "libmatrix.h"

/* This function is used to display a double matrix stored in an mxArray */

void display(const mxArray* in);

int run_main(int argc, char **argv)
{
    mxArray *in1, *in2; /* Define input parameters */
    mxArray *out = NULL;/* and output parameters to be passed to the library functions

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

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

    /* Call the library initialization routine and make sure that the
     * library was initialized properly. */

    if (!libmatrixInitialize()){
        fprintf(stderr,"Could not initialize the library.\n");
        return -2;
    }
```

```
    else
    {
        /* Call the library function */
        mlfAddmatrix(1, &out, in1, in2);

    /* Display the return value of the library function */

        printf("The value of added matrix is:\n");
        display(out);

    /* Destroy the return value since this variable will be reused in
     * the next function call. Since we are going to reuse the variable,
     * we have to set it to NULL. Refer to MATLAB Compiler SDK documentation
     * for more information on this. */

        mxDestroyArray(out); out=0;
        mlfMultiplymatrix(1, &out, in1, in2);
        printf("The value of the multiplied matrix is:\n");
        display(out);
        mxDestroyArray(out); out=0;
        mlfEigmatrix(1, &out, in1);
        printf("The eigenvalues of the first matrix are:\n");
        display(out);
        mxDestroyArray(out); out=0;

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

    /* Free the memory created */
        mxDestroyArray(in1); in1=0;
        mxDestroyArray(in2); in2 = 0;
    }

/* Note that you should call mclTerminate application at the end of
 * your application. */

    mclTerminateApplication();
    return 0;
}

/*DISPLAY This function will display the double matrix stored in an mxArray.
 * This function assumes that the mxArray passed as input contains double
 * array. */
```

```
void display(const mxArray* in)
{
    int i=0, j=0; /* loop index variables */
    int r=0, c=0; /* variables to store the row and column length of the matrix */
    double *data; /* variable to point to the double data stored within the mxArray */

    /* Get the size of the matrix */
    r = mxGetM(in);
    c = mxGetN(in);

    /* Get a pointer to the double data in mxArray */
    data = mxGetPr(in);

    /* Loop through the data and display the same in matrix format */
    for( i = 0; i < c; i++ ){
        for( j = 0; j < r; j++){
            printf("%4.2f\t",data[j*c+i]);
        }
        printf("\n");
    }
    printf("\n");
}

int main()
{
    /* Call the mclInitializeApplication routine. Make sure that the application
     * was initialized properly by checking the return status. This initialization
     * has to be done before calling any MATLAB API's or MATLAB Compiler SDK generated
     * shared library functions. */

    if( !mclInitializeApplication(NULL,0) )
    {
        fprintf(stderr, "Could not initialize the application.\n");
        return -1;
    }
    return mclRunMain((mclMainFcnType)run_main,0,NULL);
}
```

Restrictions When Using MATLAB Function loadlibrary

You cannot use the MATLAB function `loadlibrary` inside of MATLAB to load a C shared library built with MATLAB Compiler SDK.

For more information about using `loadlibrary`, see “Calling Shared Libraries in Deployed Applications” (MATLAB Compiler).

Integrate C Shared Libraries

In this section...

C Shared Library Wrapper

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

C Shared Library Example

This example takes several MATLAB files and creates a C shared library. It also includes a standalone driver application to call the shared library.

Building the Shared Library

1 Copy the following files from *matlabroot*`\extern\examples\compilersdk` to your work directory:

```
matlabroot\extern\examples\compilersdk\addmatrix.m
matlabroot\extern\examples\compilersdk\multiplymatrix.m
matlabroot\extern\examples\compilersdk\eigmatrix.m
```

2 To create the shared library, enter the following command on a single line:

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

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

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

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

Tip You can also build the shared library using the **Library Compiler** app.

Writing a Driver Application for a Shared Library

Copy *matlabroot*\extern\examples\compilersdk\matrixdriver.c to your working directory. This file contains the driver code for the application.

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

1. Initialize the MATLAB Runtime using `mclmcrInitialize()`.
2. Use `mclRunMain()` to call the code that uses the MATLAB generated shared library.
3. Declare variables and process/validate input arguments.
4. Call `mclInitializeApplication`, and test for success. This function sets up the global MATLAB Runtime state and enables the construction of MATLAB Runtime instances.

Caution Avoid issuing `cd` commands from the driver application prior to calling `mclInitializeApplication`. Failure to do so can cause a failure in MATLAB Runtime initialization.

5. Call, once for each library, `<libraryname>Initialize`, to create the MATLAB Runtime instance required by the library.
6. Invoke functions in the library, and process the results. (This is the main body of the program.)

Note If your driver application displays MATLAB figure windows, you should include a call to `mclWaitForFiguresToDie(NULL)` before calling the `Terminate` functions and `mclTerminateApplication` in the following two steps.

7. Call, once for each library, `<lib>Terminate`, to destroy the associated MATLAB Runtime.

Caution `<lib>Terminate` will bring down enough of the MATLAB Runtime address space that the same library (or any other library) cannot be initialized. Issuing a `<lib>Initialize` call after a `<lib>Terminate` call causes unpredictable results. Instead, use the following structure:

```
...code...
mclInitializeApplication();
```

```
lib1Initialize();
lib2Initialize();

lib1Terminate();
lib2Terminate();
mclTerminateApplication();
...code...
```

8 Call `mclTerminateApplication` to free resources associated with the global MATLAB Runtime state.

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

Compiling the Driver Application

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

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

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

This generates a standalone application, `matrixdriver.exe`, on Windows, and `matrixdriver`, on UNIX®.

Testing the Driver Application

These steps test your standalone driver application and shared library on your development machine.

1 To run the application, add the directory containing the shared library that was created in “Building the Shared Library” on page 2-10 to your dynamic library path.

2 Update the path for your platform by following the instructions in “MATLAB Runtime Path Settings for Development and Testing” on page 7-2.

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

```
matrixdriver.exe                                (On Windows)
matrixdriver                                     (On UNIX)
matrixdriver.app/Contents/MacOS/matrixdriver (On Mac)
```

The results are displayed as

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

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

The eigenvalues of the first matrix are:
16.12        -1.12        -0.00
```

Integrate C++ Shared Libraries

In this section...

C++ Shared Library Wrapper

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

C++ Shared Library Example

This example rewrites the C shared library example using C++. The procedure for creating a C++ shared library from MATLAB files is identical to the procedure for creating a C shared library, except you use the `cpplib` wrapper. Enter the following command on a single line:

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

The `-W cpplib:<libname>` option tells MATLAB Compiler SDK to generate a function wrapper for a shared library and call it `<libname>`. The `-T link:lib` option specifies the target output as a shared library. Note the directory where the product puts the shared library because you will need it later.

Writing the Driver Application

Note Due to name mangling in C++, you must compile your driver application with the same version of your third-party compiler that you use to compile your C++ shared library.

In the C++ version of the `matrixdriver` application `matrixdriver.cpp`, arrays are represented by objects of the class `mwArray`. Every `mwArray` class object contains a pointer to a MATLAB array structure. For this reason, the attributes of an `mwArray` object are a superset of the attributes of a MATLAB array. Every MATLAB array contains information about the size and shape of the array (i.e., the number of rows,

columns, and pages) and either one or two arrays of data. The first array stores the real part of the array data and the second array stores the imaginary part. For arrays with no imaginary part, the second array is not present. The data in the array is arranged in column-major, rather than row-major, order.

Caution Avoid issuing `cd` commands from the driver application prior to calling `mclInitializeApplication`. Failure to do so can cause a failure in MATLAB Runtime initialization.

For information about how MATLAB Compiler SDK uses a proxy layer for the libraries that an application must link, see "Understand the mclmcrrt Proxy Layer" on page 2-26.

Compiling the Driver Application

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

```
mbuild matrixdriver.cpp libmatrixp.lib            (Windows)
mbuild matrixdriver.cpp -L. -lmatrixp -I.         (UNIX)
```

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

On Windows, if this is not the case, specify the full path to `libmatrixp.lib`, and use a `-I` option to specify the directory containing the header file.

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

Incorporating a C++ Shared Library into an Application

There are two main differences to note when using a C++ shared library:

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

Exported Function Signature

The C++ shared library target generates two sets of interfaces for each MATLAB function. For more information, see “Functions Generated from MATLAB Files” on page 2-33. The generic signature of the exported C++ functions is as follows:

MATLAB Functions with No Return Values

```
bool MW_CALL_CONV <function-name>(<const_mwArray_references>);
```

MATLAB Functions with at Least One Return Value

```
bool MW_CALL_CONV <function-name>(int <number_of_return_values>,
     <mwArray_references>, <const_mwArray_references>);
```

In this case, *`const_mwArray_references`* represents a comma-separated list of references of type `const mwArray&` and *`mwArray_references`* represents a comma-separated list of references of type `mwArray&`. For example, in the `libmatrix` library, the C++ interface to the `addmatrix` MATLAB function is generated as:

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

where `a` is an output parameter and `a1` and `a2` are input parameters.

Input arguments passed to the MATLAB function via `varargin` must be passed via a single `mwArray` that is a cell array. Each element in the cell array must constitute an input argument. Output arguments retrieved from the MATLAB function via `varargout` must be retrieved via a single `mwArray` that is a cell array. Each element in the cell array will constitute an output argument. The number of elements in the cell array will be equal to `number_of_return_values` - the number of named output parameters. Also note that,

- If the MATLAB function takes a `varargin` argument, the C++ function must be passed an `mwArray` corresponding to that `varargin`, even if the `mwArray` is empty.
- If the MATLAB function takes a `varargout` argument, the C++ function must be passed an `mwArray` corresponding to that `varargin`, even if `number_of_return_values` is set to the number of named output arguments, which means meaning that `varargout` will be empty.
- The `varargout` argument needs to follow any named output arguments and precede any input arguments.

- The `varargin` argument needs to be the last argument.

Error Handling

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

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

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

Working with C++ Shared Libraries and Sparse Arrays

The MATLAB Compiler SDK C/C++ API includes static factory methods for working with sparse arrays.

For a complete list of the methods, see “C++ Utility Classes” on page A-4.

Use Multiple Shared Libraries in Single Application

In this section...

When developing applications that use multiple MATLAB shared libraries, consider the following:

- Each MATLAB shared library must be initialized separately.
- Each MATLAB shared library must be terminated separately.
- MATLAB function handles cannot be shared between shared libraries.
- MATLAB figure handles cannot be shared between shared libraries.
- MATLAB objects cannot be shared between shared libraries.
- C, Java®, and .NET objects cannot be shared between shared libraries.
- Executable data stored in cell arrays and structures cannot be shared between shared libraries

Initialize and Terminate Multiple Shared Libraries

To initialize or terminate multiple shared libraries:

1. Initialize the MATLAB Runtime using `mclmcrIntialize()`.
2. Call the portion of the application that executes the MATLAB code using `mclRunMain()`.
3. Before initializing the shared libraries, initialize the MATLAB application state using `mclInitializeApplication()`.
4. For each MATLAB shared library, call the generated initialization function, *`library`*`Initialize()`.
5. Add the code for working with the MATLAB code.
6. For each MATLAB shared library, release the resources used by the library using the generated termination function, *`library`*`Terminate()`.
7. Release the resources used by the MATLAB Runtime by calling `mclTerminateApplication()`.

This example shows the use of two shared libraries.

```
#include <stdio.h>
#include "libAddMatrix.h"
#include "libSubMatrix.h"

int run_main(int argc, const char *argv[])
{

   if( !mclInitializeApplication(NULL,0) )
   {
       fprintf(stderr, "Could not initialize the application.\n");
       return -1;
   }

   if (!libAddMatrixInitialize())
   {
       fprintf(stderr,"Could not initialize the AddMatrix library.\n");
       return -2;
   }

   if (!libSubMatrixInitialize())
   {
       fprintf(stderr,"Could not initialize the SubMatrix library.\n");
       return -2;
   }

   try
   {
   ...
   }
   catch (const mwException& e)
   {
       std::cerr << e.what() << std::endl;
       return -2;
   }
   catch (...)
   {
       std::cerr << "Unexpected error thrown" << std::endl;
       return -3;
   }

   libAddMatrixTerminate();

   libSubMatrixTerminate();
```

```
   mclTerminateApplication();
   return 0;
}

int main(int ac, const char *av[])
{
   int err = 0;
   mclmcrInitialize();
   err = mclRunMain((mclMainFcnType) run_main, ac, av);
   return err;
}
```

Work with MATLAB Function Handles

MATLAB function handles can be passed between an application and the MATLAB Runtime instance from which it originated. However, a MATLAB function handle cannot be passed into a MATLAB Runtime instance other than the one in which it originated. For example, suppose you had two MATLAB functions, `get_plot_handle` and `plot_xy`, and `plot_xy` used the function handle created by `get_plot_handle`.

```
% Saved as get_plot_handle.m
function h = get_plot_handle(lnSpec, lnWidth, mkEdge, mkFace, mkSize)
h = @draw_plot;
    function draw_plot(x, y)
        plot(x, y, lnSpec, ...
            'LineWidth', lnWidth, ...
            'MarkerEdgeColor', mkEdge, ...
            'MarkerFaceColor', mkFace, ...
            'MarkerSize', mkSize)
    end
end

% Saved as plot_xy.m
function plot_xy(x, y, h)
h(x, y);
end
```

If you compiled them into two shared libraries, the call to `plot_xy` would throw an exception.

```
#include <stdio.h>
#include "get_plot_handle.h"
#include "plot_xy.h"
```

```
int run_main(int argc, const char *argv[])
{

   if( !mclInitializeApplication(NULL,0) )
   {
       fprintf(stderr, "Could not initialize the application.\n");
       return -1;
   }

   if (!get_plot_handleInitialize())
   {
       fprintf(stderr,
         "Could not initialize the get_plot_handle library.\n");
       return -2;
   }

   if (!plot_xyInitialize())
   {
       fprintf(stderr,"Could not initialize the plot_xy library.\n");
       return -2;
   }

   try
   {
     mwArray lnSpec('--rs');
     mwArray lnWidth;
     lnWidth = 2.0;
     mwArray mkEdge('k');
     mwArray mkFace('g');
     mwArray mkSize;
     mkSize = 10.0;
     mwArray plot;
     get_plot_handle(1, plot, lnSpec, lnWidth, mkEdge, mkFace, mkSize);

     double x_data[] = {1,2,3,4,5,6,7,8,9};
     double y_data[] = {2,6,12,20,30,42,56,72,90};
     mwArray x(9, 1, mxDOUBLE_CLASS, mxREAL);
     mwArray y(9, 1, mxDOUBLE_CLASS, mxREAL);
     x.SetData(x_data, 9);
     y.SetData(y_data, 9);
     ploy_xy(x, y, plot);
   }
   catch (const mwException& e)
```

```
    {
        std::cerr << e.what() << std::endl;
        return -2;
    }
    catch (...)
    {
        std::cerr << "Unexpected error thrown" << std::endl;
        return -3;
    }

    get_plot_handleTerminate();

    plot_xyTerminate();

    mclTerminateApplication();
    return 0;
}

int main(int ac, const char *av[])
{
    int err = 0;
    mclmcrInitialize();
    err = mclRunMain((mclMainFcnType) run_main, ac, av);
    return err;
}
```

One way to handle the situation is to compile both functions into a single shared library. For example, if you called the shared library `plot_functions`, your application would only need one call to initialize the function and you could pass the function handle for `plot_xy` without error.

```
#include <stdio.h>
#include "get_plot_handle.h"
#include "plot_xy.h"

int run_main(int argc, const char *argv[])
{

    if( !mclInitializeApplication(NULL,0) )
    {
        fprintf(stderr, "Could not initialize the application.\n");
        return -1;
    }
```

```
    if (plot_functionsInitialize())
    {
        fprintf(stderr,
          "Could not initialize the plot_functions library.\n");
        return -2;
    }

    try
    {
      mwArray lnSpec('--rs');
      mwArray lnWidth;
      lnWidth = 2.0;
      mwArray mkEdge('k');
      mwArray mkFace('g');
      mwArray mkSize;
      mkSize = 10.0;
      mwArray plot;
      get_plot_handle(1, plot, lnSpec, lnWidth, mkEdge, mkFace, mkSize);

      double x_data[] = {1,2,3,4,5,6,7,8,9};
      double y_data[] = {2,6,12,20,30,42,56,72,90};
      mwArray x(9, 1, mxDOUBLE_CLASS, mxREAL);
      mwArray y(9, 1, mxDOUBLE_CLASS, mxREAL);
      x.SetData(x_data, 9);
      y.SetData(y_data, 9);
      ploy_xy(x, y, plot);
    }
    catch (const mwException& e)
    {
        std::cerr << e.what() << std::endl;
        return -2;
    }
    catch (...)
    {
        std::cerr << "Unexpected error thrown" << std::endl;
        return -3;
    }

    plot_functionsTerminate();

    mclTerminateApplication();
    return 0;
}
```

```
int main(int ac, const char *av[])
{
   int err = 0;
   mclmcrInitialize();
   err = mclRunMain((mclMainFcnType) run_main, ac, av);
   return err;
}
```

Work with Objects

MATLAB Compiler SDK enables you to return the following types of objects from the MATLAB Runtime to your application code:

- MATLAB
- C++
- .NET
- Java

However, you cannot pass an object created in one MATLAB Runtime instance into a different MATLAB Runtime instance. This conflict can happen when a function that returns an object and a function that manipulates that object are compiled into different shared libraries.

For example, you develop two functions. The first creates a bank account for a customer based on some set of conditions. The second transfers funds between two accounts.

```
% Saved as account.m
classdef account < handle

    properties
        name
    end

    properties (SetAccess = protected)
        balance = 0
        number
    end

    methods
        function obj = account(name)
            obj.name = name;
            obj.number = round(rand * 1000);
```

```
        end

        function deposit(obj, deposit)
            new_bal = obj.balance + deposit;
            obj.balance = new_bal;
        end

        function withdraw(obj, withdrawl)
            new_bal = obj.balance - withdrawl;
            obj.balance = new_bal;
        end

    end
end

% Saved as open_acct .m
function acct = open_acct(name, open_bal )

    acct = account(name);

    if open_bal > 0
        acct.deposit(open_bal);
    end

end

% Saved as transfer.m
function transfer(source, dest, amount)

    if (source.balance > amount)
        dest.deposit(amount);
        source.withdraw(amount);
    end

end
```

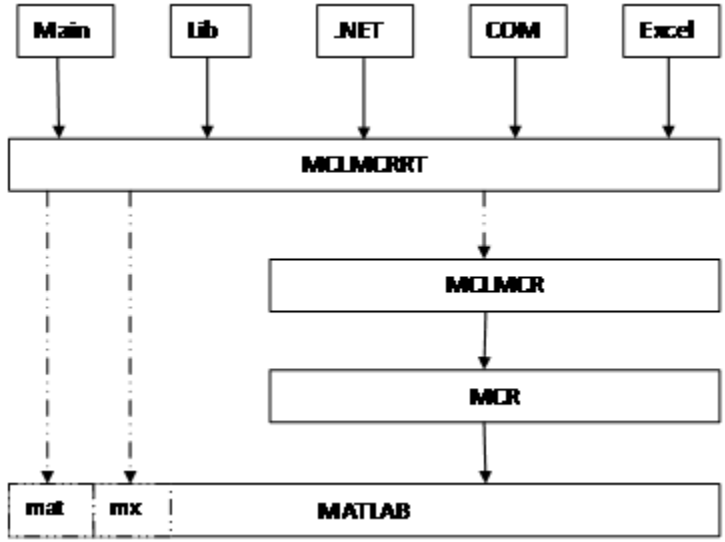
If you compiled `open_acct.m` and `transfer.m` into separate shared libraries, you could not transfer funds using accounts created with `open_acct`. The call to `transfer` throws an exception. One way of resolving this is to compile both functions into a single shared library. You could also refactor the application such that you are not passing MATLAB objects to the functions.

Understand the mclmcrrt Proxy Layer

All application and software components generated by MATLAB Compiler™ and MATLAB Compiler SDK need to link against only one MATLAB library, `mclmcrrt`. This library provides a proxy API for all the public functions in MATLAB libraries used for matrix operations, MAT-file access, utility and memory management, and application MATLAB Runtime. The `mclmcrrt` library lies between deployed MATLAB code and these other version-dependent libraries, providing the following functionality:

- Ensures that multiple versions of the MATLAB Runtime can coexist
- Provides a layer of indirection
- Ensures applications are thread-safe
- Loads the dependent (re-exported) libraries dynamically

The relationship between `mclmcrrt` and other MATLAB libraries is shown in the following figure.



The MCLMCRRT Proxy Layer

In the figure, solid arrows designate static linking and dotted arrows designate dynamic linking. The figure illustrates how the `mclmcrrt` library layer sits above the `mclmcr` and `mcr` libraries. The `mclmcr` library contains the run-time functionality of the deployed

MATLAB code. The `mcr` module ensures each bundle of deployed MATLAB code runs in its own context at run time. The `mclmcrrt` proxy layer, in addition to loading the `mclmcr`, also dynamically loads the `MX` and `MAT` modules, primarily for `mxArray` manipulation. For more information, see the MathWorks® Support database and search for information on the `MSVC` shared library.

Caution Deployed applications must only link to the `mclmcrrt` proxy layer library (`mclmcrrt.lib` on Windows, `mclmcrrt.so` on Linux, and `mclmcrrt.dylib` on Macintosh). Do not link to the other libraries shown in the figure, such as `mclmcr`, `libmx`, and so on.

Call MATLAB Compiler SDK API Functions from C/C++

In this section...

Functions in the Shared Library

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

To generate the functions described in this section, first copy `sierpinski.m`, `main_for_lib.c`, `main_for_lib.h`, and `triangle.c` from *`matlabroot`*`\extern\examples\compilersdk` into your directory, and then execute the appropriate MATLAB Compiler SDK command.

Type of Application

For a C Application on Windows

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

For a C Application on UNIX

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

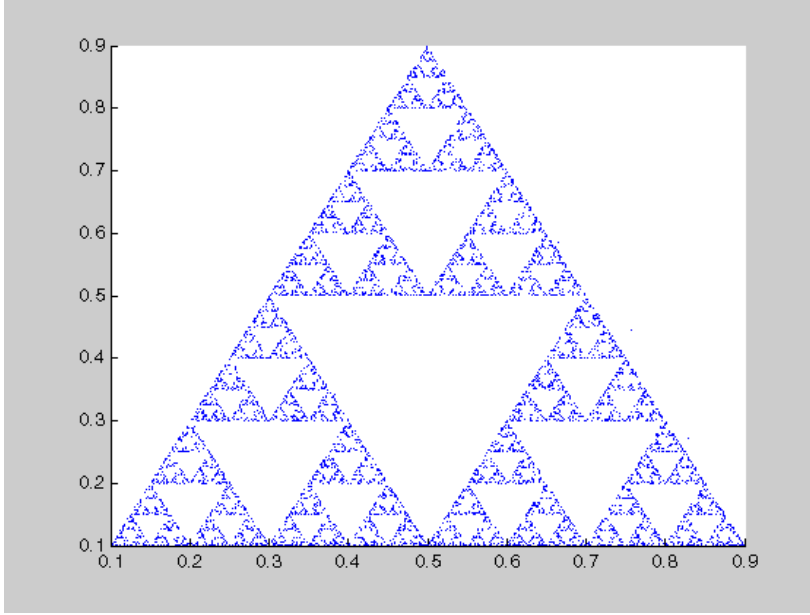
For a C++ Application on Windows

```
mcc -W cpplib:libtrianglep -T link:lib sierpinski.m
mbuild triangle.cpp main_for_lib.c libtrianglep.lib
```

For a C++ Application on UNIX

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

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



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

Structure of Programs That Call Shared Libraries

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

1. Declare variables and process/validate input arguments.
2. Call `mclInitializeApplication`, and test for success. This function sets up the global MATLAB Runtime state and enables the construction of MATLAB Runtime instances.
3. Call, once for each library, `<libraryname>Initialize`, to create the MATLAB Runtime instance required by the library.
4. Invoke functions in the library, and process the results. (This is the main body of the program.)
5. Call, once for each library, `<libraryname>Terminate`, to destroy the associated MATLAB Runtime.
6. Call `mclTerminateApplication` to free resources associated with the global MATLAB Runtime state.
7. Clean up variables, close files, etc., and exit.

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

Library Initialization and Termination Functions

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

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

```
bool libtriangleInitialize(void)
```

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

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

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

By calling this function, you can provide your own versions of the print and error handling routines called by the MATLAB Runtime. Each of these routines has the same signature (for complete details, see “Print and Error Handling Functions” on page 2-31). By overriding the defaults, you can control how output is displayed and, for example, whether or not it goes into a log file.

Note Before calling either form of the library initialization routine, you must first call `mclInitializeApplication` to set up the global MATLAB Runtime state. See “Call a Shared Library” on page 2-5 for more information.

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

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

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

Library termination is simple.

```
void libtriangleTerminate(void)
```

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

Print and Error Handling Functions

By default, MATLAB Compiler SDK generated applications and shared libraries send printed output to standard output and error messages to standard error. MATLAB Compiler SDK generates a default print handler and a default error handler that

implement this policy. If you'd like to change this behavior, you must write your own error and print handlers and pass them in to the appropriate generated initialization function.

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

The default print handler takes the following form.

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

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

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

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

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

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

Note If you provide alternate C++ implementations of either `mclDefaultPrintHandler` or `mclDefaultErrorHandler`, then functions must be declared `extern "C"`. For example:

```
extern "C" int myPrintHandler(const char *s);
```

Functions Generated from MATLAB Files

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

Note For C shared libraries, MATLAB Compiler SDK generates the `mlx` and `mlf` functions as described in this section. For C++ shared libraries, the product generates the `mlx` function the same way it does for the C shared library. However, the product generates a modified `mlf` function with these differences:

- The `mlf` before the function name is dropped to keep compatibility with R13.
- The arguments to the function are `mwArray` instead of `mxArray`.

mlx Interface Function

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

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

mlf Interface Function

The second of the generated functions begins with the prefix `mlf`. This function expects its input and output arguments to be passed in as individual variables rather than

packed into arrays. If the function is capable of producing one or more outputs, the first argument is the number of outputs requested by the caller.

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

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

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

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

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

If the output variables you pass in to an `mlf` function are not NULL, the `mlf` function will attempt to free them using `mxDestroyArray`. This means that you can reuse output variables in consecutive calls to `mlf` functions without worrying about memory leaks. It also implies that you must pass either `NULL` or a valid MATLAB array for all output variables or your program will fail because the memory manager cannot distinguish between a non-initialized (invalid) array pointer and a valid array. It will try to free a pointer that is not NULL -- freeing an invalid pointer usually causes a segmentation fault or similar fatal error.

Using varargin and varargout in a MATLAB Function Interface

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

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

For example, consider this MATLAB file interface:

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

The corresponding C interface for this is

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

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

Caution The C++ shared library interface does not support `varargin` with zero (0) input arguments. Calling your program using an empty `mwArray` results in the compiled library receiving an empty array with `nargin = 1`. The C shared library interface allows you to call `mlfFOO(NULL)` (the compiled MATLAB code interprets this as `nargin=0`). However, calling `FOO((mwArray)NULL)` with the C++ shared library interface causes the compiled MATLAB code to see an empty array as the first input and interprets `nargin=1`.

For example, compile some MATLAB code as a C++ shared library using `varargin` as the MATLAB function's list of input arguments. Have the MATLAB code display the variable `nargin`. Call the library with function *`FOO`*`()` and it won't compile, producing this error message:

```
... 'FOO' : function does not take 0 arguments
```

Call the library as:

```
mwArray junk;
FOO(junk);
```

or

```
FOO((mwArray)NULL);
```

At runtime, `nargin=1`. In MATLAB, *`FOO`*`()` is `nargin=0` and `FOO([])` is `nargin=1`.

C++ Interfaces for MATLAB Functions Using varargin and varargout

The C++ `mlx` interface for MATLAB functions does not change even if the functions use `varargin` or `varargout`. However, the C++ function interface (the second set of functions) changes if the MATLAB function is using `varargin` or `varargout`.

For examples, view the generated code for various MATLAB function signatures that use `varargin` or `varargout`.

Note For simplicity, only the relevant part of the generated C++ function signature is shown in the following examples.

function varargout = foo(varargin)

For this MATLAB function, the following C++ overloaded functions are generated:

```
No input no output:
void foo()

Only inputs:
void foo(const mwArray& varargin)

Only outputs:
void foo(int nargout, mwArray& varargout)

Most generic form that has both inputs and outputs:
void foo(int nargout, mwArray& varargout,
                      const mwArray& varargin)
```

function varargout = foo(i1, i2, varargin)

For this MATLAB function, the following C++ overloaded functions are generated:

```
Most generic form that has outputs and all the inputs
void foo(int nargout, mwArray& varargout, const
                            mwArray& i1, const
                            mwArray& i2, const
                             mwArray& varargin)

Only inputs:
```

```
void foo(const mwArray& i1,
      const mwArray& i2, const mwArray& varargin)
```

function [o1, o2, varargout] = foo(varargin)

For this MATLAB function, the following C++ overloaded functions are generated:

```
Most generic form that has all the outputs and inputs
void foo(int nargout, mwArray& o1, mwArray& o2,
                            mwArray& varargout,
                         const mwArray& varargin)


Only outputs:
void foo(int nargout, mwArray& o1, mwArray& o2,
                                   mwArray& varargout)
```

function [o1, o2, varargout] = foo(i1, i2, varargin)

For this MATLAB function, the following C++ overloaded function is generated:

```
Most generic form that has all the outputs and
                                      all the inputs
void foo(int nargout, mwArray& o1, mwArray& o2,

mwArray& varargout,
         const mwArray& i1, const mwArray& i2,
                        const mwArray& varargin)
```

Retrieving MATLAB Runtime State Information While Using Shared Libraries

When using shared libraries, you may call functions to retrieve specific information from the MATLAB Runtime state. For details, see “Set and Retrieve MATLAB Runtime Data for Shared Libraries”.

Memory Management and Cleanup

In this section...

Overview

Generated C++ code provides consistent garbage collection via the object destructors and the MATLAB Runtime's internal memory manager optimizes to avoid heap fragmentation.

If memory constraints are still present on your system, try preallocating arrays in MATLAB. This will reduce the number of calls to the memory manager, and the degree to which the heap fragments.

Passing mxArrays to Shared Libraries

When an `mxArray` is created in an application which uses the MATLAB Runtime, it is created in the managed memory space of the MATLAB Runtime.

Therefore, it is very important that you never create `mxArray`s (or call any other MATLAB function) before calling `mclInitializeApplication`.

It is safe to call `mxDestroyArray` when you no longer need a particular `mxArray` in your code, even when the input has been assigned to a persistent or global variable in MATLAB. MATLAB uses reference counting to ensure that when `mxDestroyArray` is called, if another reference to the underlying data still exists, the memory will not be freed. Even if the underlying memory is not freed, the `mxArray` passed to `mxDestroyArray` will no longer be valid.

For more information about `mclInitializeApplication` and `mclTerminateApplication`, see “Call a Shared Library” on page 2-5.

For more information about `mxArray`, see “C Matrix Library API” (MATLAB).

3

Deployment Process

This chapter tells you how to deploy compiled MATLAB code to end users.

Package C/C++ Applications

1. Gather and package the following files for installation on end user computers:
   - MATLAB Runtime installer

     See “Download the MATLAB Runtime Installer from the Web” on page 3-5.
   - MATLAB generated shared library
   - Executable for the application
2. Include directions for installing the MATLAB Runtime.

   See “Install and Configure the MATLAB Runtime” on page 3-5.

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

About the MATLAB Runtime

In this section...

The MATLAB Runtime is a standalone set of shared libraries, MATLAB code, and other files that enables the execution of MATLAB files on computers without an installed version of MATLAB. Applications that use artifacts built with MATLAB Compiler SDK require access to an appropriate version of the MATLAB Runtime to run.

End-users of compiled artifacts without access to MATLAB must install the MATLAB Runtime on their computers or know the location of a network-installed MATLAB Runtime. The installers generated by the compiler apps may include the MATLAB Runtime installer. If you compiled your artifact using `mcc`, you should direct your end-users to download the MATLAB Runtime installer from the website `http://www.mathworks.com/products/compiler/mcr`.

See “Install and Configure the MATLAB Runtime” on page 3-5 for more information.

How is the MATLAB Runtime Different from MATLAB?

The MATLAB Runtime differs from MATLAB in several important ways:

- In the MATLAB Runtime, MATLAB files are encrypted and immutable.
- MATLAB has a desktop graphical interface. The MATLAB Runtime has all the MATLAB functionality without the graphical interface.
- The MATLAB Runtime is version-specific. You must run your applications with the version of the MATLAB Runtime associated with the version of MATLAB Compiler SDK with which it was created. For example, if you compiled an application using version 6.3 (R2016b) of MATLAB Compiler, users who do not have MATLAB installed must have version 9.1 of the MATLAB Runtime installed. Use `mcrversion` to return the version number of the MATLAB Runtime.
- The MATLAB paths in a MATLAB Runtime instance are fixed and cannot be changed. To change them, you must first customize them within MATLAB.

Performance Considerations and the MATLAB Runtime

MATLAB Compiler SDK was designed to work with a large range of applications that use the MATLAB programming language. Because of this, run-time libraries are large.

Since the MATLAB Runtime technology provides full support for the MATLAB language, including the Java programming language, starting a compiled application takes approximately the same amount of time as starting MATLAB. The amount of resources consumed by the MATLAB Runtime is necessary in order to retain the power and functionality of a full version of MATLAB.

Calls into the MATLAB Runtime are serialized so calls into the MATLAB Runtime are threadsafe. This can impact performance.

Install and Configure the MATLAB Runtime

In this section...

Download the MATLAB Runtime Installer from the Web

Download the MATLAB® Runtime from the website at http://www.mathworks.com/products/compiler/mcr.

Install the MATLAB Runtime Interactively

To install the MATLAB Runtime:

1 Unzip/Extract the archive containing the MATLAB Runtime installer.

Platform	Steps
Windows	Double-click the self-extracting MATLAB Runtime installer that you downloaded from the web. For example, an R2017b runtime will have the name `MCR_R2017b_win64_installer.exe`. Double clicking the installer extracts the necessary files and automatically starts the installer.

Platform	Steps
Linux	Unzip the MATLAB Runtime installer at the terminal using the `unzip` command. For example, if you are unzipping the R2017b MATLAB Runtime installer, at the Terminal, type: `unzip MCR_R2017b_glnxa64_installer.zip`
macOS	Unzip the MATLAB Runtime installer at the terminal using the `unzip` command. For example, if you are unzipping the R2017b MATLAB Runtime installer, at the Terminal, type: `unzip MCR_R2017b_maci64_installer.dmg.zip`

Note The release part of the installer filename (*_R2017b_*) will change from one release to the next.

2 Start the MATLAB Runtime installer.

Platform	Steps
Windows	Installer automatically starts after completing the previous step.

Linux	At the Terminal, type:
	
	`sudo ./install`
	
	Note On Debian® based Linux distributions, you will need to type:
	
	`gksudo ./install`

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

3 When the MATLAB Runtime installer starts, it displays a dialog box. Read the information and then click **Next** to proceed with the installation.
4 Specify the folder in which you want to install the MATLAB Runtime in the **Folder Selection** dialog box.

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

---

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

The MATLAB Runtime installer starts copying files into the installation folder.
6 On Linux and macOS platforms, after copying files to your disk, the MATLAB Runtime installer displays the **Product Configuration Notes** dialog box. This dialog box contains information necessary for setting your path environment variables. Copy the path information from this dialog box and then click **Next**.
7 Click **Finish** to exit the installer.

Install the MATLAB Runtime Non-Interactively

To install the MATLAB Runtime without having to interact with the installer dialog boxes, use one of the MATLAB Runtime installer's non-interactive modes:

- silent—the installer runs as a background task and does not display any dialog boxes
- automated—the installer displays the dialog boxes but does not wait for user interaction

When run in silent or automated mode, the MATLAB Runtime installer uses default values for installation options. You can override these defaults by using MATLAB Runtime installer command-line options or an installer control file.

Note When running in silent or automated mode, the installer overwrites the default installation location.

Running the Installer in Silent Mode

To install the MATLAB Runtime in silent mode:

1 Extract the contents of the MATLAB Runtime installer file to a temporary folder, called `$temp` in this documentation.

Note On Windows systems, **manually** extract the contents of the installer file.

2 Run the MATLAB Runtime installer, specifying the `-mode` option and `-agreeToLicense yes` on the command line.

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

Platform	Command
Windows	`setup -mode silent -agreeToLicense yes`
Linux	`./install -mode silent -agreeToLicense yes`
macOS	`./install -mode silent -agreeToLicense yes`

Note If you do not include the `-agreeToLicense yes` the installer will not install the MATLAB Runtime.

3 View a log of the installation.

On Windows systems, the MATLAB Runtime installer creates a log file, named `mathworks_`*`username`*`.log`, where *`username`* is your Windows log-in name, in the location defined by your `TEMP` environment variable.

On Linux and macOS systems, the MATLAB Runtime installer displays the log information at the command prompt, unless you redirect it to a file.

Customizing a Non-Interactive Installation

When run in one of the non-interactive modes, the installer will use the default values unless told to do otherwise. Like the MATLAB installer, the MATLAB Runtime installer accepts a number of command line options that modify the default installation properties.

Option	Description
`-destinationFolder`	Specifies where the MATLAB Runtime will be installed.
`-outputFile`	Specifies where the installation log file is written.
`-automatedModeTimeout`	Specifies how long, in milliseconds, that the dialog boxes are displayed when run in automatic mode.
`-inputFile`	Specifies an installer control file with the values for all of the above options.

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

Install the MATLAB Runtime without Administrator Rights

To install the MATLAB Runtime as a user without administrator rights on Windows:

1. Use the MATLAB Runtime installer to install it on a Windows machine where you have administrator rights.
2. Copy the folder where the MATLAB Runtime was installed to the machine without administrator rights. You can compress the folder into zip file and distribute to multiple users.
3. On the machine without administrator rights, add the *`mcr_root`*`\runtime\`*`arch`* directory onto the user's path environment variable.

   Note You don't need administrator rights for adding directories to a user's path environment variable.

Multiple MATLAB Runtime Versions on Single Machine

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

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

MATLAB and MATLAB Runtime on Same Machine

You do not need to install MATLAB Runtime on your machine if your machine has MATLAB installed. The version of MATLAB should be the same as the version of MATLAB that was used to create the compiled MATLAB code. Also, to act as the MATLAB Runtime replacement, the MATLAB installation must include MATLAB Compiler.

You can, however, install the MATLAB Runtime for debugging purposes.

Modifying the Path

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

- **Windows**

  To run deployed MATLAB code against MATLAB Runtime install, *mcr_root**ver*\runtime\win64 must appear on your system path before *matlabroot*\runtime\win64.

  If *mcr_root**ver*\runtime*arch* appears first on the compiled application path, the application uses the files in the MATLAB Runtime install area.

  If *matlabroot*\runtime*arch* appears first on the compiled application path, the application uses the files in the MATLAB installation area.

- **UNIX**

  To run deployed MATLAB code against MATLAB Runtime on Linux, Linux x86-64, or the <mcr_root>/runtime/<arch> folder must appear on your LD_LIBRARY_PATH before *matlabroot*/runtime/<arch>.

- **macOS**

To run deployed MATLAB code on macOS, the `<mcr_root>/runtime` folder must appear on your `DYLD_LIBRARY_PATH` before *`matlabroot`*`/runtime/<arch>`.

To run MATLAB on macOS or Intel® Mac, *`matlabroot`*`/runtime/<arch>` must appear on your `DYLD_LIBRARY_PATH` before the `<mcr_root>/bin` folder.

Uninstall MATLAB Runtime

The method you use to uninstall MATLAB Runtime from your computer varies depending on the type of computer.

Windows

1. Start the uninstaller.

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

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

Linux

1. Exit the application.
2. Enter this command at the Linux prompt:

   ```
   rm -rf mcr_root
   ```

   where *`mcr_root`* represents the name of your top-level MATLAB installation folder.

macOS

1. Exit the application.
2. Navigate to your MATLAB Runtime installation folder. For example, the installation folder might be named `MATLAB_Compiler_Runtime.app` in your Applications folder.

3 Drag your MATLAB Runtime installation folder to the trash, and then select **Empty Trash** from the Finder menu.

Use Parallel Computing Toolbox in Deployed Applications

There are two ways to pass a cluster profile to a standalone application that uses the Parallel Computing Toolbox:

1. Save the cluster profile to your MATLAB preferences.

   The cluster profile will be automatically bundled with the generated application and available to the Parallel Computing Toolbox code.
2. Embed the cluster profile in the application.

Embed Parallel Computing Toolbox Profile in the Application

To embed a Parallel Computing Toolbox profile in an application, you must ensure that the application loads a Parallel Computing Toolbox profile. You have two options for loading a profile:

- load the cluster profile in the compiled MATLAB function

```
function run_parallel_funct
setmcruserdata('ParallelProfile', 'profile');
a = parallel_funct
end
```

- load the cluster profile in the application calling the MATLAB function

```
mxArray *key = mxCreateString("ParallelProfile");
mxArray *value = mxCreateString("\usr\userdir\config.settings");
if (!setmcruserdata(key, value))
{
    fprintf(stderr,
            "Could not set MCR user data: \n %s ",
            mclGetLastErrorMessage());
    return -1;
}
```

When you package and deploy an application that uses Parallel Computing Toolbox you must ensure that the Parallel Computing Toolbox profile is included along with the application. The profile must also be placed in the location expected by the application.

Deploy Applications on Network Drives

You can deploy a compiled application to a network drive so that it can be accessed by all network users without having them install the MATLAB Runtime on their individual machines.

Note There is no need to perform these steps on a Linux system.

The component registration is in support of Excel® add-ins and COM components, which both run on Windows only.

Distributing to a Linux network file system is exactly the same as distributing to a local file system. You only need to set up the `LD_LIBRARY_PATH` or use scripts which points to the MATLAB Runtime installation.

1. On any Windows machine, run `mcrinstaller` function to obtain name of the MATLAB Runtime Installer executable.
2. Copy the entire MATLAB Runtime installation folder onto a network drive.
3. Copy the compiled application into a separate folder in the network drive and add the path `<mcr_root>\<ver>\runtime\<arch>` to all client machines. All network users can then execute the application.
4. Run `vcredist_x86.exe` on for 32-bit clients; run `vcredist_x64.exe` for 64-bit clients.
5. If you are using MATLAB Compiler SDK to create COM objects, register `mwcomutil.dll` on every client machine.

   To register the DLLs, at the DOS prompt enter

   `mwregsvr <fully_qualified_pathname\dllname.dll>`

   These DLLs are located in `<mcr_root>\<ver>\runtime\<arch>`.

   Note These libraries are automatically registered on the machine on which the installer was run.

MATLAB Compiler SDK Deployment Messages

To enable display of MATLAB Compiler SDK deployment messages, see the *MATLAB Desktop Tools and Environment* documentation.

4

Distributing Code to an End User

MATLAB Runtime Component Cache and Deployable Archive Embedding

Deployable archive data is automatically embedded directly in shared libraries by default and extracted to a temporary folder.

Automatic embedding enables usage of the MATLAB Runtime component cache features through environment variables.

These variables allow you to specify the following:

- Define the default location where you want the deployable archive to be automatically extracted
- Add diagnostic error printing options that can be used when automatically extracting the deployable archive, for troubleshooting purposes
- Tuning the MATLAB Runtime component cache size for performance reasons.

Use the following environment variables to change these settings.

Environment Variable	Purpose	Notes
MCR_CACHE_ROOT	When set to the location of where you want the deployable archive to be extracted, this variable overrides the default per-user component cache location.	Does not apply
MCR_CACHE_VERBOSE	When set to any value, this variable prints logging details about the component cache for diagnostic reasons. This can be very helpful if problems are encountered during deployable archive extraction.	Logging details are turned off by default (for example, when this variable has no value).

Environment Variable	Purpose	Notes
MCR_CACHE_SIZE	When set, this variable overrides the default component cache size.	The initial limit for this variable is 32M (megabytes). This may, however, be changed after you have set the variable the first time. Edit the file `.max_size`, which resides in the file designated by running the `mcrcachedir` command, with the desired cache size limit.

Note If you run `mcc` specifying conflicting wrapper and target types, the archive will not be embedded into the generated component. For example, if you run:

```
mcc -W lib:myLib -T link:exe test.m test.c
```

the generated `test.exe` will not have the archive embedded in it, as if you had specified a `-C` option to the command line.

Caution Do not extract the files within the `.ctf` file and place them individually under version control. Since the `.ctf` file contains interdependent MATLAB functions and data, the files within it must be accessed only by accessing the `.ctf` file. For best results, place the entire `.ctf` file under version control.

Compiler Commands

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

Command Overview

In this section...

Compiler Options

`mcc` is the MATLAB command that invokes the compiler. You can issue the `mcc` command either from the MATLAB command prompt or the system prompt.

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

```
mcc -m -v myfun
```

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

Combining Options

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

```
mcc -mv myfun
```

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

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

This format is *not* valid:

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

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

If you include any C or C++ file names on the `mcc` command line, the files are passed directly to `mbuild`, along with any MATLAB Compiler SDK generated C or C++ files.

Conflicting Options on the Command Line

If you use conflicting options, the compiler resolves them from left to right, with the rightmost option taking precedence. For example, using the equivalencies in “Macros”,

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

is equivalent to:

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

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

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

Using File Extensions

The valid, recommended file extension for a file submitted to the compiler is `.m`. Always include the `.m` extension, when compiling with `mcc` .

Note `.p` files have precedence over `.m` files, therefore if both `.p` files and `.m` files reside in a folder, and a file name is specified without an extension, the `.p` file will be selected.

Interfacing MATLAB Code to C/C++ Code

To designate code to be compiled with C or C++, rewrite the C or C++ function as a MEX-file and call it from your application.

You can control whether the MEX-file or a MATLAB stub gets called by using the `isdeployed` function.

Code Proper Return Types From C and C++ Methods

To avoid potential problems, ensure all C methods you write (and reference from within MATLAB code) return a `bool` return type indicating the status. C++ methods should return nothing (`void`).

Include Files for Compilation Using %#function

In this section...

Using feval

In standalone mode, the pragma `%#function <function_name-list>` informs the compiler that the specified function(s) should be included in the compilation, whether or not the dependency analysis detects it. Without this pragma, the dependency analysis will not be able to locate and compile all MATLAB files used in your application. This pragma adds the top-level function as well as all the local functions in the file to the compilation.

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

Using %#function

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

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

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

% List all the possible windows.
% Note the list of functions in the following function pragma is
% on a single line of code.
%#function bartlett, barthannwin, blackman, blackmanharris,
bohmanwin, chebwin, flattopwin, gausswin, hamming, hann, kaiser,
nuttallwin, parzenwin, rectwin, tukeywin, triang
```

```
window = feval(filterName,N);
% Apply the window to the data.
ret = data.*window;
```

Compiler Tips

In this section...

Calling a Function from the Command Line

You can make a MATLAB function into a standalone that is directly callable from the system command line. All the arguments passed to the MATLAB function from the system command line are strings. Two techniques to work with these functions are:

- Modify the original MATLAB function to test each argument and convert the strings to numbers.
- Write a wrapper MATLAB function that does this test and then calls the original MATLAB function.

  For example:

  ```
  function x=foo(a, b)
      if (ischar(a)), a = str2num(a), end;
      if (ischar(b)), b = str2num(b), end;

  % The rest of your MATLAB code here...
  ```

  You only do this if your function expects numeric input. If your function expects strings, there is nothing to do because that's the default from the command line.

Using MAT-Files in Deployed Applications

To use a MAT-file in a deployed application, use the `-a` option to include the file in the deployable archive.

Compiling a GUI That Contains an ActiveX Control

When you save a GUI that contains ActiveX® components, GUIDE creates a file in the current folder for each such component. The file name consists of the name of the GUI followed by an underscore (_) and `activex`*n*, where *n* is a sequence number. For example, if the GUI is named `ActiveXcontrol` then the file name would be `ActiveXcontrol_activex1`. The file name does not have an extension.

If you use the `mcc` command to compile a GUIDE-created GUI that contains an ActiveX component, you must use the `-a` option to add the ActiveX control files that GUIDE saved in the current folder to the deployable archive. Your command should be similar to

```
mcc -m mygui -a mygui_activex1
```

where `mygui_activex1` is the name of the file. If you have more than one such file, use a separate `-a` option for each file.

Deploying Applications That Call the Java Native Libraries

If your application interacts with Java, you need to specify the search path for native method libraries by editing `librarypath.txt` and deploying it.

1. Copy `librarypath.txt` from *`matlabroot`*`/toolbox/local/librarypath.txt`.
2. Place `librarypath.txt` in `<mcr_root>/<ver>/toolbox/local`.

   `<mcr_root>` refers to the complete path where the MATLAB Runtime library archive files are installed on your machine.
3. Edit `librarypath.txt` by adding the folder that contains the native library that your application's Java code needs to load.

Locating .fig Files in Deployed Applications

MATLAB Compiler and MATLAB Compiler SDK locate `.fig` files automatically when there is a MATLAB file with the same name as the `.fig` file in the same folder. If the `.fig` file does not follow this rule, it must be added with the `-a` option.

Terminating Figures by Force In an Application

The purpose of `mclWaitForFiguresToDie` is to block execution of a calling program as long as figures created in the deployed application are displayed. `mclWaitForFiguresToDie` takes no arguments. Your application can call `mclWaitForFiguresToDie` any time during execution. Typically you use `mclWaitForFiguresToDie` when:

- There are one or more figures you want to remain open.
- The function that displays the graphics requires user input before continuing.

When `mclWaitForFiguresToDie` is called, execution of the calling program is blocked if any figures created by the calling object remain open.

Both .NET assemblies and Java packages use `mclWaitForFiguresToDie` through the use of wrapper methods. See "Block Console Display When Creating Figures" and "Execution of Applications that Create Figures" for more details and code fragment examples.

Caution Use caution when calling the `mclWaitForFiguresToDie` function. Calling this function from an interactive program like Excel can hang the application. This function should be called *only* from console-based programs.

Passing Arguments to and from a Standalone Application

To pass input arguments to a MATLAB Compiler generated standalone application, you pass them just as you would to any console-based application. For example, to pass a file called `helpfile` to the compiled function called `filename`, use

```
filename helpfile
```

To pass numbers or letters (e.g., 1, 2, and 3), use

```
filename 1 2 3
```

Do not separate the arguments with commas.

To pass matrices as input, use

```
filename "[1 2 3]" "[4 5 6]"
```

You have to use the double quotes around the input arguments if there is a space in it. The calling syntax is similar to the `dos` command. For more information, see the MATLAB `dos` command.

The things you should keep in mind for your MATLAB file before you compile are:

- The input arguments you pass to your application from a system prompt are considered as string input. If, in your MATLAB code before compilation, you are expecting the data in different format, say double, you will need to convert the string input to the required format. For example, you can use `str2num` to convert the string input to numerical data. You can determine at run time whether or not to do this by using the `isdeployed` function. If your MATLAB file expects numeric inputs in MATLAB, the code can check whether it is being run as a standalone application. For example:

  ```
  function myfun (n1, n2)
  if (isdeployed)
      n1 = str2num(n1);
      n2 = str2num(n2);
  end
  ```

- You cannot return back values from your standalone application to the user. The only way to return values from compiled code is to either display it on the screen or store it in a file. To display your data on the screen, you either need to unsuppress (do not use semicolons) the commands whose results yield data you want to return to the screen or, use the `disp` command to display the value. You can then redirect these outputs to other applications using output redirection (> operator) or pipes (only on UNIX systems).

Passing Arguments to a Double-Clickable Application

On Windows, if you want to run the standalone application by double-clicking it, you can create a batch file that calls this standalone application with the specified input arguments. Here is an example of the batch file:

```
rem main.bat file that calls sub.exe with input parameters
sub "[1 2 3]" "[4 5 6]"
@echo off
pause
```

The last two lines of code keep your output on the screen until you press a key. If you save this file as `main.bat`, you can run your code with the specified arguments by double-clicking the `main.bat` icon.

Using Graphical Applications in Shared Library Targets

When deploying a GUI as a shared library to a C/C++ application, use `mclWaitForFiguresToDie` to display the GUI until it is explicitly terminated.

Using the VER Function in a Compiled MATLAB Application

When you use the `VER` function in a compiled MATLAB application, it will perform with the same functionality as if you had called it from MATLAB. However, be aware that when using `VER` in a compiled MATLAB application, only version information for toolboxes which the compiled application uses will be displayed.

Troubleshooting

Common Issues

Some of the most common issues encountered when using MATLAB Compiler SDK generated shared libraries are:

- **Compilation fails with an error message.** This can indicate a failure during any one of the internal steps involved in producing the final output.
- **Compilation succeeds but the application does not execute because required DLLs are not found.** All shared libraries required for your standalone executable or shared library are contained in the MATLAB Runtime. Installing the MATLAB Runtime is required for any of the deployment targets.
- **Compilation succeeds, and the resultant file starts to execute but then produces errors and/or generates a crash dump.**
- **The compiled program executes on the machine where it was compiled but not on other machines.**
- **The compiled program executes on some machines and not others.**

Compilation Failures

You typically compile your MATLAB code on a development machine, test the resulting executable on that machine, and deploy the executable and MATLAB Runtime to a test or customer machine without MATLAB. The compilation process performs dependency analysis on your MATLAB code, creates an encrypted archive of your code and required toolbox code, generates wrapper code, and compiles the wrapper code into an executable. If your application fails to build an executable, the following questions may help you isolate the problem.

Is your installed compiler supported by MATLAB Compiler SDK?

See the current list of supported compilers at `http://www.mathworks.com/support/compilers/current_release/`.

Are error messages produced at compile time?

See error messages in "Error and Warning Messages" on page 6-15.

Are you compiling within or outside of MATLAB?

`mcc` can be invoked from the operating system command line or from the MATLAB prompt. When you run `mcc` inside the MATLAB environment, MATLAB will modify environment variables in its environment as necessary so `mcc` will run. Issues with `PATH`, `LD_LIBRARY_PATH`, or other environment variables seen at the operating system command line are often not seen at the MATLAB prompt. The environment that MATLAB uses for `mcc` can be listed at the MATLAB prompt. For example:

```
>>!set
```

lists the environment on Windows platforms.

```
>>!printenv
```

lists the environment on UNIX platforms. Using this path allows you to use `mcc` from the operating system command line.

Have you tried to compile any of the C/C++ examples in MATLAB Compiler SDK help?

The source code for all C/C++ examples is provided with MATLAB Compiler SDK and is located in *`matlabroot`*`\extern\examples\compilersdk`, where *`matlabroot`* is the root folder of your MATLAB installation.

Is your MATLAB object failing to load?

If your MATLAB object fails to load, it is typically a result of the MATLAB Runtime not finding required class definitions.

When working with MATLAB objects that are loaded from a MAT file, remember to include the following statement in your MATLAB function:

```
%#function class_constructor
```

Using the `%#function` pragma forces dependency analyzer to load needed class definitions, enabling the MATLAB Runtime to successfully load the object.

If you are compiling a driver application, are you using mbuild?

MathWorks recommends and supports using `mbuild` to compile your driver application. `mbuild` is designed and tested to correctly build driver applications. It will ensure that all MATLAB header files are found by the C/C++ compiler, and that all necessary libraries are specified and found by the linker.

Are you trying to compile your driver application using Microsoft Visual Studio or another IDE?

If you are using an IDE, in addition to linking to the generated export library, you need to include an additional dependency to `mclmcrrt.lib`. This library is provided for all supported Microsoft compilers in *`matlabroot`*`\extern\lib\`*`arch`*`\microsoft`.

Are you importing the correct versions of import libraries?

If you have multiple versions of MATLAB installed on your machine, it is possible that an older or incompatible version of the library is referenced. Ensure that the only MATLAB library that you are linking to is `mclmcrrt.lib` and that it is referenced from the appropriate folder.

Are you able to compile the matrixdriver example?

Typically, if you cannot compile the examples in the documentation, it indicates an issue with the installation of MATLAB or your system compiler. See “Integrate C Shared Libraries” on page 2-10 and “Integrate C++ Shared Libraries” on page 2-14 for these examples.

Do you get the `MATLAB:I18n:InconsistentLocale` Warning?

The warning message

```
MATLAB:I18n:InconsistentLocale - The system locale setting,
system_locale_name, is different from the user locale
setting, user_locale_name
```

indicates a mismatch between locale setting on Microsoft Windows systems. This may affect your ability to display certain characters. For information about changing the locale settings, see your operating system Help.

Testing Failures

After you have successfully compiled your application, the next step is to test it on a development machine and deploy it on a target machine. Typically the target machine does not have a MATLAB installation and requires that the MATLAB Runtime be installed. A distribution includes all of the files that are required by your application to run, which include the executable, deployable archive and the MATLAB Runtime.

See “Package C/C++ Applications” on page 3-2 for information on distribution contents for specific application types and platforms.

Test the application on the development machine by running the application against the MATLAB Runtime shipped with MATLAB Compiler SDK. This will verify that library dependencies are correct, that the deployable archive can be extracted and that all MATLAB code, MEX—files and support files required by the application have been included in the archive. If you encounter errors testing your application, the questions in the column to the right may help you isolate the problem.

Are you able to execute the application from MATLAB?

On the development machine, you can test your application's execution by issuing `!` *`application-name`* at the MATLAB prompt. If your application executes within MATLAB but not from outside, this can indicate an issue with the one of the system variables:

- `PATH`
- `LD_LIBRARY_PATH`
- `DYLD_LIBRARY_PATH`

Does the application begin execution and result in MATLAB or other errors?

Ensure that you included all necessary files when compiling your application (see the `readme.txt` file generated with your compilation for more details).

Functions that are called from your main MATLAB file are automatically included by MATLAB Compiler SDK as are functions included using the `%#function` pragma. However, functions that are not explicitly called, for example through EVAL, need to be included at compilation using the `-a` switch of the `mcc` command. Also, any support files like `.mat`, `.txt`, or `.html` files need to be added to the archive with the `-a` switch. There is a limitation on the functionality of MATLAB and associated toolboxes that can be

compiled. Check the documentation to see that the functions used in your application's MATLAB files are valid. Check the file `mccExcludedFiles.log` on the development machine. This file lists all functions called from your application that cannot be compiled.

Do you have multiple MATLAB versions installed?

Executables generated using MATLAB Compiler SDK components are designed to run in an environment where multiple versions of MATLAB are installed. Some older versions of MATLAB may not be fully compatible with this architecture.

On Windows, ensure that the *`matlabroot`*`\runtime\win64` of the version of MATLAB in which you are compiling appears ahead of *`matlabroot`*`\runtime\win64` of other versions of MATLAB installed on the `PATH` environment variable on your machine.

Similarly, on UNIX, ensure that the dynamic library paths (`LD_LIBRARY_PATH` on Linux) match. Do this by comparing the outputs of `!printenv` at the MATLAB prompt and `printenv` at the shell prompt. Using this path allows you to use `mcc` from the operating system command line.

If you are testing a shared library and driver application, did you install the MATLAB Runtime?

All shared libraries required for a shared library are contained in the MATLAB Runtime. Installing the MATLAB Runtime is required for any of the deployment targets.

Do you receive an error message about a missing DLL?

Error messages indicating missing DLLs such as `mclmcrrt7x.dll` or `mclmcrrt7x.so` are generally caused by incorrect installation of the MATLAB Runtime. It is also possible that the MATLAB Runtime is installed correctly, but that the `PATH`,`LD_LIBRARY_PATH`, or `DYLD_LIBRARY_PATH` variables are set incorrectly. For information on installing the MATLAB Runtime on a deployment machine, see “Install and Configure the MATLAB Runtime” on page 3-5.

Caution Do not solve these problems by moving libraries or other files within the MATLAB Runtime folder structure. The MATLAB Runtime system is designed to accommodate different MATLAB Runtime versions operating on the same machine. The folder structure is an important part of this feature.

Are you receiving errors when trying to run the shared library application?

Calling MATLAB Compiler SDK generated shared libraries requires correct initialization and termination in addition to library calls themselves. For information on calling shared libraries, see “Call MATLAB Compiler SDK API Functions from C/C++” on page 2-28.

Some key points to consider to avoid errors at run time:

- Ensure that the calls to `mclinitializeApplication` and *`libname`*`Initialize` are successful. The first function enables construction of MATLAB Runtime instances. The second creates the MATLAB Runtime instance required by the library named *`libname`*. If these calls are not successful, your application will not execute.
- Do not use any `mw-` or `mx-`functions before calling `mclinitializeApplication`. This includes static and global variables that are initialized at program start. Referencing `mw-` or `mx-`functions before initialization results in undefined behavior.
- Do not re-initialize (call `mclinitializeApplication`) after terminating it with `mclTerminateApplication`. The `mclinitializeApplication` and *`libname`*`Initialize` functions should be called only once.
- Ensure that you do not have any library calls after `mclTerminateApplication`.
- Ensure that you are using the correct syntax to call the library and its functions.

Does your system’s graphics card support the graphics application?

In situations where the existing hardware graphics card does not support the graphics application, you should use software OpenGL®. OpenGL libraries are visible for an application by appending `matlab/sys/opengl/lib/arch` to the `LD_LIBRARY_PATH`. For example:

```
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:matlab/sys/opengl/lib/arch
```

Is OpenGL properly installed on your system?

When searching for OpenGL libraries, the MATLAB Runtime first looks on the system library path. If OpenGL is not found there, it will use the `LD_LIBRARY_PATH` environment variable to locate the libraries. If you are getting failures due to the OpenGL libraries not being found, you can append the location of the OpenGL libraries to the `LD_LIBRARY_PATH` environment variable. For example:

```
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:matlab/sys/opengl/lib/glnxa64
```

Application Deployment Failures

After the application is working on the test machine, failures can be isolated in end-user deployment. The end users of your application need to install the MATLAB Runtime on their machines. The MATLAB Runtime includes a set of shared libraries that provides support for all features of MATLAB. If your application fails during end-user deployment, the following questions in the column to the right may help you isolate the problem.

Note There are a number of reasons why your application might not deploy to end users, after running successfully in a test environment. For a detailed list of guidelines for writing MATLAB code that can be consumed by end users, see "Write Deployable MATLAB Code"

Is the MATLAB Runtime installed?

Installing the MATLAB Runtime is required for any of the deployment targets. See "Install and Configure the MATLAB Runtime" on page 3-5 for complete information.

If running on UNIX or Mac, did you update the dynamic library path after installing the MATLAB Runtime?

For information on installing the MATLAB Runtime on a deployment machine, see "Install and Configure the MATLAB Runtime" on page 3-5.

Do you receive an error message about a missing DLL?

Error messages indicating missing DLLs such as `mclmcrrt7x.dll` or `mclmcrrt7x.so` are generally caused by incorrect installation of the MATLAB Runtime. It is also possible that the MATLAB Runtime is installed correctly, but that the `PATH`, `LD_LIBRARY_PATH`, or `DYLD_LIBRARY_PATH` variables are set incorrectly. For information on installing the MATLAB Runtime on a deployment machine, see "Install and Configure the MATLAB Runtime" on page 3-5.

Caution Do not solve these problems by moving libraries or other files within the MATLAB Runtime folder structure. The MATLAB Runtime system is designed to accommodate different MATLAB Runtime versions operating on the same machine. The folder structure is an important part of this feature.

Do you have write access to the necessary folders?

The first operation attempted by an application with compiled MATLAB code is extraction of the deployable archive. If the archive is not extracted, the application cannot access the compiled MATLAB code and the application fails.

There are three possible folders where the deployable archive is extracted:

- If the deployable archive is embedded and you are using the default environment settings, the archive extracts into the current user's temp folder.
- If the deployable archive is embedded and you set the environment variable `MCR_CACHE_ROOT`, the archive extracts into the folder specified by `MCR_CACHE_ROOT`.
- If the deployable archive is not embedded, the archive extracts into the current folder of the component.

Troubleshoot mbuild

This section identifies some of the more common problems that might occur when configuring `mbuild` to create standalone 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, you are asked if you want to overwrite the existing options file. If you choose to do so, the existing options file is copied to a new location and a new options file is created.

Directory or File Not Writeable. If a destination folder 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. If you run `mbuild filename` and get errors, it may be because you are not using the proper options file. Run `mbuild -setup` to ensure proper compiler and linker settings.

Compiler and/or Linker Not Found. On Windows, if you get errors such as `unrecognized command` or `file not found`, make sure the command-line tools are installed and the path and other environment variables are set correctly in the options file. For Microsoft Visual Studio®, for example, make sure to run `vcvars32.bat` (MSVC 6.x and earlier) or `vsvars32.bat` (MSVC 8.x and later).

mbuild Not a Recognized Command. If `mbuild` is not recognized, verify that *`matlabroot`*`\bin` is in your path. On UNIX, it may be necessary to rehash.

mbuild Works from the Shell But Not from MATLAB (UNIX). If the command

```
mcc -m hello
```

works from the UNIX command prompt but not from the MATLAB prompt, you may have a problem with your `.cshrc` file. When MATLAB launches a new C shell to perform compilations, it executes the `.cshrc` script. If this script causes unexpected changes to the `PATH` environment variable, an error may occur. You can test this before starting MATLAB by performing the following:

```
setenv SHELL /bin/sh
```

If this works correctly, then you should check your `.cshrc` file for problems setting the `PATH` environment variable.

Internal Error when Using mbuild -setup (Windows). Some antivirus software packages may conflict with the `mbuild-setup` process. If you get an error message during `mbuild -setup` of the following form

```
mex.bat: internal error in sub get_compiler_info(): don't
recognize <string>
```

then you need to disable your antivirus software temporarily and rerun `mbuild-setup`. After you have successfully run the `setup` option, you can re-enable your antivirus software.

Verification of mbuild Fails. If none of the previous solutions addresses your difficulty with `mbuild`, contact Technical Support at MathWorks at `http://www.mathworks.com/contact_TS.html`.

Deployed Applications

Checking access to X display <IP-address>:0.0 . . . If no response hit ^C and fix host or access control to host. Otherwise, checkout any error messages that follow and fix . . . Successful. . . . This message can be ignored.

??? Error: File: /home/username/<MATLAB file_name>Line: 1651 Column: 8 Arguments to IMPORT must either end with ".*" or else specify a fully qualified class name: "<class_name>" fails this test. The `import` statement is referencing a Java class (`<class_name>`) that MATLAB Compiler SDK (if the error occurs at compile time) or the MATLAB Runtime (if the error occurs at run time) cannot find. To work around this, ensure that the JAR file that contains the Java class is stored in a folder that is on the Java class path. (See *`matlabroot`*`/toolbox/local/classpath.txt` for the class path.) If the error occurs at run time, the classpath is stored in *`matlabroot`*`/toolbox/local/classpath.txt` when running on the development machine. It is stored in `<mcr_root>/toolbox/local/classpath.txt` when running on a target machine.

Undefined function or variable 'matlabrc'. When MATLAB or the MATLAB Runtime starts, they attempt to execute the MATLAB file `matlabrc.m`. This message means that this file cannot be found. To work around this, try each of these suggestions in this order:

- **Undefined function or variable 'matlabrc'.** Ensure that your application runs in MATLAB (uncompiled) without this error.
- **Undefined function or variable 'matlabrc'.** Ensure that MATLAB starts up without this error.
- **Undefined function or variable 'matlabrc'.** Verify that the generated deployable archive contains a file called `matlabrc.m`.
- **Undefined function or variable 'matlabrc'.** Verify that the generated code (in the `*_mcc_component_data.c*` file) adds the deployable archive folder containing `matlabrc.m` to the MATLAB Runtime path.
- **Undefined function or variable 'matlabrc'.** Delete the `*_mcr` folder and rerun the application.
- **Undefined function or variable 'matlabrc'.** Recompile the application.

Error: library mclmcrrt76.dll not found. This error can occur for the following reasons:

- **Error: library mclmcrrt76.dll not found.** The machine on which you are trying to run the application an different, incompatible version of the MATLAB Runtime installed on it than the one the application was originally built with.
- **Error: library mclmcrrt76.dll not found.** You are not running a version of MATLAB Compiler SDK compatible with the MATLAB Runtime version the application was built with.

To solve this problem, on the deployment machine, install the version of MATLAB you used to build the application.

Invalid .NET Framework.\n Either the specified framework was not found or is not currently supported. This error occurs when the .NET Framework version your application is specifying (represented by *n*) is not supported by the current version of MATLAB Compiler SDK.

System.AccessViolationException: Attempted to read or write protected memory. The message:

```
System.ArgumentException: Generate Queries
                                threw General Exception:
System.AccessViolationException: Attempted to
                        read or write protected memory.
 This is often an indication that other memory is corrupt.
```

indicates a library initialization error caused by a Microsoft Visual Studio project linked against a `MCLMCRRT7`*`XX`*`.DLL` placed outside *`matlabroot`*.

Error and Warning Messages

In this section...

About Error and Warning Messages

This appendix lists and describes error messages and warnings generated by the compiler. Compile-time messages are generated during the compile or link phase. It is useful to note that most of these compile-time error messages should not occur if the MATLAB software can successfully execute the corresponding MATLAB file.

Use this reference to:

- Confirm that an error has been reported
- Determine possible causes for an error
- Determine possible ways to correct an error

When using MATLAB Compiler SDK, if you receive an internal error message, record the specific message and report it to Technical Support at `http://www.mathworks.com/contact_TS.html`.

Compile-Time Errors

Error: An error occurred while shelling out to mex/mbuild (error code = errorno). Unable to build (specify the -v option for more information)

The compiler reports this error if `mbuild` or `mex` generates an error.

Error: An error occurred writing to file "filename": reason

The file can not be written. The reason is provided by the operating system. For example, you may not have sufficient disk space available to write the file.

Error: Cannot write file "filename" because MCC has already created a file with that name, or a file with that name was specified as a command line argument

The compiler has been instructed to generate two files with the same name. For example:

```
mcc -W lib:liba liba -t     % Incorrect
```

Error: Could not check out a Compiler license

No additional MATLAB Compiler SDK licenses are available for your workgroup.

Error: File: "filename" not found

A specified file can not be found on the path. Verify that the file exists and that the path includes the file's location. You can use the `-I` option to add a folder to the search path.

Error: File: "filename" is a script MATLAB file and cannot be compiled with the current Compiler

The compiler cannot compile script MATLAB files.

Error: File: filename Line: # Column: # A variable cannot be made storageclass1 after being used as a storageclass2

You cannot change a variable's storage class (global/local/persistent). Even though MATLAB allows this type of change in scope, the compiler does not.

Error: Found illegal whitespace character in command line option: "string". The strings on the left and right side of the space should be separate arguments to MCC

For example:

```
mcc('-m', '-v', 'hello')% Correct
mcc('-m -v', 'hello')      % Incorrect
```

Error: Improper usage of option -optionname. Type "mcc -?" for usage information

You have incorrectly used a MATLAB Compiler SDK option. For more information about MATLAB Compiler SDK options, see “mcc Command Arguments Listed Alphabetically” (MATLAB Compiler), or type `mcc -?` at the command prompt.

Error: libraryname library not found

MATLAB has been installed incorrectly.

Error: mclFreeStackTrace

The compiler reports this error when `startup.m` file executes functions that alter MATLAB search path, for example, `cd` or `addpath`. Hence, `startup.m` file cannot execute appropriately in deployed environment.

Insert the `isdeployed` function before the path altering functions in the `startup.m` file and recompile the application after modifying `startup.m`.

Error: No source files were specified (-? for help)

You must provide the compiler with the name of the source file(s) to compile.

Error: "optionname" is not a valid -option argument

You must use an argument that corresponds to the option. For example:

```
mcc -W main ...    % Correct
mcc -W mex  ...    % Incorrect
```

Error: Out of memory

Typically, this message occurs because the compiler requests a larger segment of memory from the operating system than is currently available. Adding additional memory to your system can alleviate this problem.

Error: Previous warning treated as error

When you use the `-w error` option, this error appears immediately after a warning message.

Error: The argument after the -option option must contain a colon

The format for this argument requires a colon. For more information, see `mcc`, or type `mcc -?` at the command prompt.

Error: The environment variable MATLAB must be set to the MATLAB root directory

On UNIX, the `MATLAB` and `LM_LICENSE_FILE` variables must be set. The `mcc` shell script does this automatically when it is called the first time.

Error: The license manager failed to initialize (error code is errornumber)

You do not have a valid license or no additional licenses are available.

Error: The option -option is invalid in modename mode (specify -? for help)

The specified option is not available.

Error: The specified file "filename" cannot be read

There is a problem with your specified file. For example, the file is not readable because there is no read permission.

Error: The -optionname option requires an argument (e.g. "proper_example_usage")

You have incorrectly used a compiler option. For more information about compiler options, see `mcc`, or type `mcc -?` at the command prompt.

Error: -x is no longer supported

The compiler no longer generates MEX-files because there is no longer any performance advantage to doing so. The MATLAB JIT Accelerator makes compilation for speed obsolete.

Error: Unable to open file "filename":<string>

There is a problem with your specified file. For example, there is no write permission to the output folder, or the disk is full.

Error: Unable to set license linger interval (error code is errornumber)

A license manager failure has occurred. Contact Technical Support with the full text of the error message.

Error: Unknown warning enable/disable string: warningstring

`-w enable:`, `-w disable:`, and `-w error:` require you to use one of the warning string identifiers listed in “Warning Messages” on page 6-19.

Error: Unrecognized option: -option

The option is not a valid option. See `mcc`, for a complete list of valid options for MATLAB Compiler SDK, or type `mcc -?` at the command prompt.

Warning Messages

This section lists the warning messages that MATLAB Compiler SDK can generate. Using the `-w` option for `mcc`, you can control which messages are displayed. Each warning message contains a description and the warning message identifier string (in parentheses) that you can enable or disable with the `-w` option. For example, to produce an error message if you are using a trial MATLAB Compiler SDK license to create your standalone application, you can use:

```
mcc -w error:trial_license -mvg hello
```

To enable all warnings except those generated by the `save` command, use:

```
mcc -w enable -w disable:trial_license ...
```

To display a list of all the warning message identifier strings, use:

```
mcc -w list
```

For additional information about the `-w` option, see `mcc`.

Warning: File: filename Line: # Column: # The #function pragma expects a list of function names

Identifier: `pragma_function_missing_names`

This pragma informs the compiler that the specified function(s) provided in the list of function names will be called through an `feval` call. This will automatically compile the selected functions.

Warning: MATLAB file "filename" was specified on the command line with full path of "pathname", but was found on the search path in directory "directoryname" first

Identifier: `specified_file_mismatch`

The compiler detected an inconsistency between the location of the MATLAB file as given on the command line and in the search path. The compiler uses the location in the search path. This warning occurs when you specify a full path name on the `mcc` command line and a file with the same base name (*`filename`*) is found earlier on the search path. This warning is issued in the following example if the file `afile.m` exists in both `dir1` and `dir2`,

```
mcc -m -I /dir1 /dir2/afile.m
```

Warning: The file filename was repeated on MATLAB Compiler SDK command line

Identifier: `repeated_file`

This warning occurs when the same file name appears more than once on the compiler command line. For example,

```
mcc -m sample.m sample.m
```

Warning: The name of a shared library should begin with the letters "lib". "libraryname" doesn't

Identifier: `missing_lib_sentinel`

This warning is generated if the name of the specified library does not begin with the letters “`lib`”. For example,

```
mcc -t -W lib:liba -T link:lib a0 a1
```

will not generate a warning while

```
mcc -t -W lib:a -T link:lib a0 a1
```

will generate a warning.

This warning is specific to UNIX and does not occur on the Windows operating system.

Warning: All warnings are disabled

Identifier: `all_warnings`

This warning displays all warnings generated by the compiler. This warning is disabled.

Warning: A line has num1 characters, violating the maximum page width (num2)

Identifier: `max_page_width_violation`

This warning is generated if there are lines that exceed the maximum page width, `num2`. This warning is disabled.

Warning: The option -optionname is ignored in modename mode (specify -? for help)

Identifier: `switch_ignored`

This warning is generated if an option is specified on the `mcc` command line that is not meaningful in the specified mode. This warning is enabled.

Warning: Unrecognized Compiler pragma “pragmaname”

Identifier: `unrecognized_pragma`

This warning is generated if you use an unrecognized pragma. This warning is enabled.

Warning: "functionname1" is a MEX- or P-file being referenced from "functionname2"

Identifier: `mex_or_p_file`

This warning is generated if `functionname2` calls `functionname1`, which is a MEX- or P-file. This warning is enabled.

Trial Compiler license. The generated application will expire 30 days from today, on date

Identifier: `trial_license`

This warning displays the date that the deployed application will expire. This warning is enabled.

Dependency Analysis Errors

- “MATLAB Runtime/Dispatcher Errors” on page 6-21
- “XML Parser Errors” on page 6-21

MATLAB Runtime/Dispatcher Errors

These errors originate directly from the MATLAB Runtime/Dispatcher. If one of these error occurs, report it to Technical Support at MathWorks at `http://www.mathworks.com/contact_TS.html`.

XML Parser Errors

These errors appear as

```
depfun Error: XML error: <message>
```

Where <message> is a message returned by the XML parser. If this error occurs, report it to Technical Support at MathWorks at http://www.mathworks.com/contact_TS.html.

Reference Information

MATLAB Runtime Path Settings for Development and Testing

In this section...

Path for Java Development on All Platforms

There are additional requirements when programming in the Java programming language. For more information see “Configure Your Java Environment”.

Path Modifications Required for Accessibility

In order to use some screen-readers or assistive technologies, such as JAWS®, you must add the following DLLs to your Windows path:

```
matlabroot\sys\java\jre\arch\jre\bin\JavaAccessBridge.dll
matlabroot\sys\java\jre\arch\jre\bin\WindowsAccessBridge.dll
```

You may not be able to use such technologies without doing so.

Windows Settings for Development and Testing

When programming with compiled MATLAB code, add the following folder to your system `PATH` environment variable:

```
matlabroot\runtime\win32|win64
```

Linux Settings for Development and Testing

Add the following platform-specific folders to your dynamic library path.

Note For readability, the following commands appear on separate lines, but you must enter each `setenv` command on one line.

```
setenv LD_LIBRARY_PATH
   matlabroot/runtime/glnxa64:
   matlabroot/bin/glnxa64:
   matlabroot/sys/os/glnxa64:
   matlabroot/sys/opengl/lib/glnxa64
```

OS X Settings for Development and Testing

Add the following platform-specific folders to your dynamic library path.

Note For readability, the following commands appear on separate lines, but you must enter each `setenv` command on one line.

```
setenv DYLD_LIBRARY_PATH
   matlabroot/runtime/maci64:
   matlabroot/bin/maci64:
   matlabroot/sys/os/maci64:
```

MATLAB Runtime Path Settings for Run-Time Deployment

In this section...

General Path Guidelines

Regardless of platform, be aware of the following guidelines with regards to placing specific folders on the path:

- Always avoid including `arch` on the path. Failure to do so may inhibit ability to run multiple MATLAB Runtime instances.
- Ideally, set the environment in a separate shell script to avoid run-time errors caused by path-related issues.

Path for Java Applications on All Platforms

When your users run applications that contain compiled MATLAB code, you must instruct them to set the path so that the system can find the MATLAB Runtime.

Note When you deploy a Java application to end users, they must set the class path on the target machine.

The system needs to find `.jar` files containing the MATLAB libraries. To tell the system how to locate the `.jar` files it needs, specify a `classpath` either in the `javac` command or in your system environment variables.

Windows Path for Run-Time Deployment

The following folder should be added to the system path:

*mcr_root**version*\runtime\win64

mcr_root refers to the complete path where the MATLAB Runtime library archive files are installed on the machine where the application is to be run.

mcr_root is version specific; you must determine the path after you install the MATLAB Runtime.

Note If you are running the MATLAB Runtime installer on a shared folder, be aware that other users of the share may need to alter their system configuration.

Linux Paths for Run-Time Deployment

Use these `setenv` commands to set your MATLAB Runtime paths.

```
setenv LD_LIBRARY_PATH
   mcr_root/version/runtime/glnxa64:
   mcr_root/version/bin/glnxa64:
   mcr_root/version/sys/os/glnxa64:
   mcr_root/version/sys/opengl/lib/glnxa64
```

OS X Paths for Run-Time Deployment

Use these `setenv` commands to set your MATLAB Runtime paths.

```
setenv DYLD_LIBRARY_PATH
   mcr_root/version/runtime/maci64:
   mcr_root/version/bin/maci64:
   mcr_root/version/sys/os/maci64
```

MATLAB Compiler SDK Licensing

Use MATLAB Compiler SDK Licenses for Development

You can run the MATLAB Compiler SDK compiler from the MATLAB command prompt or the system prompt.

MATLAB Compiler SDK uses a lingering license. This means that when the MATLAB Compiler SDK license is checked out, a timer is started. When that timer reaches 30 minutes, the license key is returned to the license pool. The license key will not be returned until that 30 minutes is up, regardless of whether `mcc` has exited or not.

Each time a compiler command is issued, the timer is reset.

Running MATLAB Compiler SDK in MATLAB Mode

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

Running MATLAB Compiler SDK in Standalone Mode

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

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

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

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

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

This message is given when no licenses are available. As long as licenses are available, the user gets the license and no message is displayed. The best way to guarantee that all MATLAB Compiler SDK users have constant access to MATLAB Compiler SDK is to have an adequate supply of licenses for your users.

Deployment Product Terms

A

Add-in — A Microsoft Excel add-in is an executable piece of code that can be actively integrated into a Microsoft Excel application. Add-ins are front-ends for COM components, usually written in some form of Microsoft Visual Basic®.

Application program interface (API) — A set of classes, methods, and interfaces that is used to develop software applications. Typically an API is used to provide access to specific functionality. See `MWArray`.

Application — An end user-system into which a deployed functions or solution is ultimately integrated. Typically, the end goal for the deployment customer is integration of a deployed MATLAB function into a larger enterprise environment application. The deployment products prepare the MATLAB function for integration by wrapping MATLAB code with enterprise-compatible source code, such as C, C++, C# (.NET), F#, and Java code.

Assembly — An executable bundle of code, especially in .NET.

B

Binary — See *Executable*.

Boxed Types — Data types used to wrap opaque C structures.

Build — See *Compile*.

C

Class — A user-defined type used in C++, C#, and Java, among other object-oriented languages, that is a prototype for an object in an object-oriented language. It is analogous to a derived type in a procedural language. A class is a set of objects which share a common structure and behavior. Classes relate in a class hierarchy. One class is a specialization (a subclass) of another (one of its *superclasses*) or comprises other classes. Some classes use other classes in a client-server relationship. Abstract classes have no members, and concrete classes have one or more members. Differs from a MATLAB class

Compile — In MATLAB Compiler and MATLAB Compiler SDK, to compile MATLAB code involves generating a binary that wraps around MATLAB code, enabling it to execute in various computing environments. For example, when MATLAB code is

compiled into a Java package, a Java wrapper provides Java code that enables the MATLAB code to execute in a Java environment.

COM component — In MATLAB Compiler, the executable back-end code behind a Microsoft Excel add-in. In MATLAB Compiler SDK, an executable component, to be integrated with Microsoft COM applications.

Console application — Any application that is executed from a system command prompt window.

D

Data Marshaling — Data conversion, usually from one type to another. Unless a MATLAB deployment customer is using type-safe interfaces, data marshaling—as from mathematical data types to MathWorks data types such as represented by the `MWArray` API—must be performed manually, often at great cost.

Deploy — The act of integrating MATLAB code into a larger-scale computing environment, usually to an enterprise application, and often to end users.

Deployable archive — The deployable archive is embedded by default in each binary generated by MATLAB Compiler or MATLAB Compiler SDK. It houses the deployable package. All MATLAB-based content in the deployable archive uses the Advanced Encryption Standard (AES) cryptosystem. See “Additional Details” (MATLAB Compiler).

DLL — Dynamic link library. Microsoft's implementation of the shared library concept for Windows. Using DLLs is much preferred over the previous technology of static (or non-dynamic) libraries, which had to be manually linked and updated.

E

Empties — Arrays of zero (`0`) dimensions.

Executable — An executable bundle of code, made up of binary bits (zeros and ones) and sometimes called a *binary*.

F

Fields — For this definition in the context of MATLAB Data Structures, see *Structs*.

Fields and Properties — In the context of .NET, *Fields* are specialized classes used to hold data. *Properties* allow users to access class variables as if they were accessing member fields directly, while actually implementing that access through a class method.

I

Integration — Combining deployed MATLAB code's functionality with functionality that currently exists in an enterprise application. For example, a customer creates a mathematical model to forecast trends in certain commodities markets. In order to use this model in a larger-scale financial application (one written with the Microsoft .NET Framework, for instance) the deployed financial model must be integrated with existing C# applications, run in the .NET enterprise environment.

Instance — For the definition of this term in context of MATLAB Production Server™ software, see *MATLAB Production Server Server Instance*.

J

JAR — Java archive. In computing software, a JAR file (or Java ARchive) aggregates many files into one. Software developers use JARs to distribute Java applications or libraries, in the form of classes and associated metadata and resources (text, images, etc.). Computer users can create or extract JAR files using the `jar` command that comes with a Java Development Kit (JDK).

Java-MATLAB Interface — Known as the *JMI Interface*, this is the Java interface built into MATLAB software.

JDK — The Java Development Kit is a free Oracle® product which provides the environment required for programming in Java.

JMI Interface — see *Java-MATLAB Interface*.

JRE — Java Run-Time Environment is the part of the Java Development Kit (JDK) required to run Java programs. It comprises the Java Virtual Machine, the Java platform core classes, and supporting files. It does not include the compiler, debugger, or other tools present in the JDK™. The JRE™ is the smallest set of executables and files that constitute the standard Java platform.

M

Magic Square — A square array of integers arranged so that their sum is the same when added vertically, horizontally, or diagonally.

MATLAB Runtime — An execution engine made up of the same shared libraries. MATLAB uses these libraries to enable the execution of MATLAB files on systems without an installed version of MATLAB.

MATLAB Runtime singleton — See *Shared MATLAB Runtime instance*.

MATLAB Runtime workers — A MATLAB Runtime session. Using MATLAB Production Server software, you have the option of specifying more than one MATLAB Runtime session, using the `--num-workers` options in the server configurations file.

MATLAB Production Server Client — In the MATLAB Production Server software, clients are applications written in a language supported by MATLAB Production Server that call deployed functions hosted on a server.

MATLAB Production Server Configuration — An instance of the MATLAB Production Server containing at least one server and one client. Each configuration of the software usually contains a unique set of values in the server configuration file, `main_config` (MATLAB Production Server).

MATLAB Production Server Server Instance — A logical server configuration created using the `mps-new` command in MATLAB Production Server software.

MATLAB Production Server Software — Product for server/client deployment of MATLAB programs within your production systems, enabling you to incorporate numerical analytics in enterprise applications. When you use this software, web, database, and enterprise applications connect to MATLAB programs running on MATLAB Production Server via a lightweight client library, isolating the MATLAB programs from your production system. MATLAB Production Server software consists of one or more servers and clients.

Marshaling — See *Data Marshaling*.

mbuild — MATLAB Compiler SDK command that compiles and links C and C++ source files into standalone applications or shared libraries. For more information, see the `mbuild` function reference page.

`mcc` — The MATLAB command that invokes the compiler. It is the command-line equivalent of using the compiler apps.

Method Attribute — In the context of .NET, a mechanism used to specify declarative information to a .NET class. For example, in the context of client programming with MATLAB Production Server software, you specify method attributes to define MATLAB structures for input and output processing.

`mxArray` interface — The MATLAB data type containing all MATLAB representations of standard mathematical data types.

`MWArray` interface — A proxy to `mxArray`. An application program interface (API) for exchanging data between your application and MATLAB. Using `MWArray`, you marshal data from traditional mathematical types to a form that can be processed and understood by MATLAB data type `mxArray`. There are different implementations of the `MWArray` proxy for each application programming language.

P

Package — The act of bundling the deployed MATLAB code, along with the MATLAB Runtime and other files, into an installer that can be distributed to others. The compiler apps place the installer in the `for_redistribution` subfolder. In addition to the installer, the compiler apps generate a number of lose artifacts that can be used for testing or building a custom installer.

PID File — See *Process Identification File (PID File)*.

Pool — A pool of threads, in the context of server management using MATLAB Production Server software. Servers created with the software do not allocate a unique thread to each client connection. Rather, when data is available on a connection, the required processing is scheduled on a pool, or group, of available threads. The server configuration file option `--num-threads` sets the size of that pool (the number of available request-processing threads) in the master server process.

Process Identification File (PID File) — A file that documents informational and error messages relating to a running server instance of MATLAB Production Server software.

Program — A bundle of code that is executed to achieve a purpose. Programs usually are written to automate repetitive operations through computer processing. Enterprise system applications usually consist of hundreds or even thousands of smaller programs.

Properties — For this definition in the context of .NET, see *Fields and Properties*.

Proxy — A software design pattern typically using a class, which functions as an interface to something else. For example, `MWArray` is a proxy for programmers who need to access the underlying type `mxArray`.

S

Server Instance — See MATLAB Production Server Server Instance.

Shared Library — Groups of files that reside in one space on disk or memory for fast loading into Windows applications. Dynamic-link libraries (DLLs) are Microsoft's implementation of the shared library concept for Microsoft Windows.

Shared MATLAB Runtime instance — When using MATLAB Compiler SDK, you can create a shared MATLAB Runtime instance, also known as a singleton. When you invoke MATLAB Compiler with the `-S` option through the compiler (using either `mcc` or a compiler app), a single MATLAB Runtime instance is created for each COM component or Java package in an application. You reuse this instance by sharing it among all subsequent class instances. Such sharing results in more efficient memory usage and eliminates the MATLAB Runtime startup cost in each subsequent class instantiation. All class instances share a single MATLAB workspace and share global variables in the deployed MATLAB files. MATLAB Compiler SDK creates singletons by default for .NET assemblies. MATLAB Compiler creates singletons by default for the COM components used by the Excel add-ins.

State — The present condition of MATLAB, or the MATLAB Runtime. MATLAB functions often carry state in the form of variable values. The MATLAB workspace itself also maintains information about global variables and path settings. When deploying functions that carry state, you must often take additional steps to ensure state retention when deploying applications that use such functions.

Structs — MATLAB Structures. Structs are MATLAB arrays with elements that you access using textual field designators. Fields are data containers that store data of a specific MATLAB type.

System Compiler — A key part of Interactive Development Environments (IDEs) such as Microsoft Visual Studio.

T

Thread — A portion of a program that can run independently of and concurrently with other portions of the program. See *pool* for additional information on managing the number of processing threads available to a server instance.

Type-safe interface — An API that minimizes explicit type conversions by hiding the `MWArray` type from the calling application.

W

Web Application Archive (WAR) —In computing, a Web Application Archive is a JAR file used to distribute a collection of `JavaServer` pages, servlets, Java classes, XML files, tag libraries, and static web pages that together constitute a web application.

Webfigure — A MathWorks representation of a MATLAB figure, rendered on the web. Using the WebFigures feature, you display MATLAB figures on a website for graphical manipulation by end users. This enables them to use their graphical applications from anywhere on the web, without the need to download MATLAB or other tools that can consume costly resources.

Windows Communication Foundation (WCF) — The Windows Communication Foundation™ is an application programming interface in the .NET Framework for building connected, service-oriented, web-centric applications. WCF is designed in accordance with service oriented architecture principles to support distributed computing where services are consumed by client applications.

Functions

<library>Initialize[WithHandlers]

Initialize MATLAB Runtime instance associated with *library*

Syntax

```
bool libraryInitialize(void)
bool libraryInitializeWithHandlers(
             mclOutputHandlerFcn error_handler,
             mclOutputHandlerFcn print_handler)
```

Description

Each generated library has its own MATLAB Runtime instance. These two functions, *library*`Initialize` and *library*`InitializeWithHandlers` initialize the MATLAB Runtime instance associated with *library*. Users must call one of these functions after calling `mclInitializeApplication` and before calling any of the compiled functions exported by the library. Each returns a boolean indicating whether or not initialization was successful. If they return `false`, calling any further compiled functions will result in unpredictable behavior. *library*`InitializeWithHandlers` allows users to specify how to handle error messages and printed text. The functions passed to *library*`InitializeWithHandlers` will be installed in the MATLAB Runtime instance and called whenever error text or regular text is to be output.

Examples

```
if (!libmatrixInitialize())
{
    fprintf(stderr,
            "An error occurred while initializing: \n %s ",
            mclGetLastErrorMessage());
    return -2;
}
```

See Also

`<library>Terminate`

Topics

“Library Initialization and Termination Functions” on page 2-30

Introduced in R2009a

mclGetLastErrorMessage

Last error message from unsuccessful function call

Syntax

```
const char* mclGetLastErrorMessage()
```

Description

This function returns a function error message (usually in the form of `false` or `-1`). It cannot catch the errors related to MATLAB Runtime initialization and can catch only errors thrown by MATLAB functions or code.

Example

```
char *args[] = { "-nodisplay" };
if(!mclInitializeApplication(args, 1))
{
    fprintf(stderr,
            "An error occurred while initializing: \n %s ",
            mclGetLastErrorMessage());
    return -1;
}
```

See Also

`<library>Initialize[WithHandlers]` | `<library>Terminate` | `mclInitializeApplication` | `mclTerminateApplication`

Introduced in R2010b

mclGetLogFileName

Retrieve name of log file used by MATLAB Runtime

Syntax

```
const char* mclGetLogFileName()
```

Description

Use `mclGetLogFileName()` to retrieve the name of the log file used by the MATLAB Runtime. Returns a character string representing log file name used by MATLAB Runtime.

Examples

```
printf("Logfile name : %s\n",mclGetLogFileName());
```

Introduced in R2009a

mclInitializeApplication

Set up application state shared by all MATLAB Runtime instances created in current process

Syntax

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

Description

Set up the application state shared by all MATLAB Runtime instances created in current process. Call only once per process. The function takes an array of strings (possibly of zero length) and a count containing the size of the string array. The string array may contain the following MATLAB command line switches, which have the same meaning as they do when used in MATLAB:

- `-appendlogfile`
- `-Automation`
- `-beginfile`
- `-debug`
- `-defer`
- `-display`
- `-Embedding`
- `-endfile`
- `-fork`
- `-java`
- `-jdb`
- `-logfile`
- `-minimize`
- `-MLAutomation`

- `-noaccel`
- `-nodisplay`
- `-noFigureWindows`
- `-nojit`
- `-nojvm`
- `-noshelldde`
- `-nosplash`
- `-r`
- `-Regserver`
- `-shelldde`
- `-singleCompThread`
- `-student`
- `-Unregserver`
- `-useJavaFigures`
- `-mwvisual`
- `-xrm`

Caution `mclInitializeApplication` must be called once only per process. Calling `mclInitializeApplication` more than once may cause your application to exhibit unpredictable or undesirable behavior.

Caution When running on Mac, if `-nodisplay` is used as one of the options included in `mclInitializeApplication`, then the call to `mclInitializeApplication` must occur before calling `mclRunMain`.

Examples

To start all MATLAB Runtime in a given process with the `-nodisplay` option, for example, use the following code:

```
const char *args[] = { "-nodisplay" };
if (! mclInitializeApplication(args, 1))
{
```

```
    fprintf(stderr,
            "An error occurred while initializing: \n %s ",
            mclGetLastErrorMessage());
    return -1;
}
```

See Also

`mclTerminateApplication`

Topics

“Integrate a C/C++ Shared Library into an Application” on page 2-2

Introduced in R2009a

mclIsJVMEnabled

Determine if MATLAB Runtime was launched with instance of Java Virtual Machine (JVM)

Syntax

```
bool mclIsJVMEnabled()
```

Description

Use `mclIsJVMEnabled()` to determine if the MATLAB Runtime was launched with an instance of a Java Virtual Machine (JVM™). Returns `true` if MATLAB Runtime is launched with a JVM instance, else returns `false`.

Examples

```
printf("JVM initialized : %d\n", mclIsJVMEnabled());
```

Introduced in R2009a

mclIsMCRInitialized

Determine if MATLAB Runtime has been properly initialized

Syntax

```
bool mclIsMCRInitialized()
```

Description

Use `mclIsMCRInitialized()` to determine whether or not the MATLAB Runtime has been properly initialized. Returns

- `true` if MATLAB Runtime is already initialized
- `false` if the MATLAB Runtime is not initialized

Note This method can only be called once the MATLAB Runtime proxy library has been initiated.

Examples

```
printf("MCR initialized : %d\n", mclIsMCRInitialized());
```

Introduced in R2009a

mclIsNoDisplaySet

Determine if `-nodisplay` mode is enabled

Syntax

```
bool mclIsNoDisplaySet()
```

Description

Use `mclIsNoDisplaySet()` to determine if `-nodisplay` mode is enabled. Returns `true` if `-nodisplay` is enabled, else returns `false`.

Note Always returns `false` on Windows systems since the `-nodisplay` option is not supported on Windows systems.

Examples

```
printf("nodisplay set : %d\n",mclIsNoDisplaySet());
```

Introduced in R2009a

mclmcrInitialize

Initializes the MATLAB Runtime proxy library

Syntax

```
mclmcrInitialize();
```

Description

`mclmcrInitialize` is called before any other MATLAB APIs. It initializes the library used to create the MATLAB Runtime proxy used by all other MATLAB generated APIs.

See Also

`mclInitializeApplication`

Topics

“Integrate a C/C++ Shared Library into an Application” on page 2-2

Introduced in R2013b

mclRunMain

Mechanism for creating identical wrapper code across all platforms

Syntax

```
typedef int (*mclMainFcnType)(int, const char **);

int mclRunMain(mclMainFcnType run_main,
               int argc,
               const char **argv)
```

Description

As you need to provide wrapper code when creating an application which uses a C or C++ shared library created by MATLAB Compiler SDK, `mclRunMain` enables you with a mechanism for creating identical wrapper code across all MATLAB Compiler SDK platform environments.

`mclRunMain` is especially helpful in Macintosh OS X environments where a run loop must be created for correct MATLAB Runtime operation.

When a Mac OS X run loop is started, if `mclInitializeApplication` specifies the `-nojvm` or `-nodisplay` option, creating a run loop is a straight-forward process. Otherwise, you must create a Cocoa framework. The Cocoa frameworks consist of libraries, APIs, and MATLAB Runtime that form the development layer for all of Mac OS X.

Generally, the function pointed to by `run_main` returns with a pointer (return value) to the code that invoked it. When using Cocoa on the Macintosh, however, when the function pointed to by `run_main` returns, the MATLAB Runtime calls exit before the return value can be received by the application, due to the inability of the underlying code to get control when Cocoa is shut down.

Caution You should not use `mclRunMain` if your application brings up its own full graphical environment.

Note In non-Macintosh environments, `mclRunMain` acts as a wrapper and doesn't perform any significant processing.

Parameters

`run_main`

Name of function to execute after MATLAB Runtime set-up code.

`argc`

Number of arguments being passed to `run_main` function. Usually, `argc` is received by application at its `main` function.

`argv`

Pointer to an array of character pointers. Usually, `argv` is received by application at its `main` function.

Examples

Call using this basic structure:

```
int returncode = 0;
mclInitializeApplication(NULL,0);
returncode = mclRunMain((mclmainFcn)
                  my_main_function,0,NULL);
```

See Also

`mclInitializeApplication`

Introduced in R2010b

mclTerminateApplication

Close MATLAB Runtime-internal application state

Syntax

```
bool mclTerminateApplication(void)
```

Description

Call this function once at the end of your program to close MATLAB Runtime-internal application state. Call only once per process. After you have called this function, you cannot call any further MATLAB Compiler SDK-generated functions or any functions in any MATLAB library.

Caution `mclTerminateApplication` must be called once only per process. Calling `mclTerminateApplication` more than once may cause your application to exhibit unpredictable or undesirable behavior.

Caution `mclTerminateApplication` will close any visible or invisible figures before exiting. If you have visible figures that you would like to wait for, use `mclWaitForFiguresToDie`.

Examples

At the start of your program, call `mclInitializeApplication` to ensure your library was properly initialized:

```
mclInitializeApplication(NULL,0);
if (!libmatrixInitialize()){
    fprintf(stderr,
            "An error occurred while initializing: \n %s ",
            mclGetLastErrorMessage());
```

```
    return -1;
}
```

At your program's exit point, call `mclTerminateApplication` to properly shut the application down:

```
mxDestroyArray(in1); in1=0;
mxDestroyArray(in2); in2 = 0;
mclTerminateApplication();
return 0;
```

See Also

`mclInitializeApplication`

Introduced in R2009a

mclWaitForFiguresToDie

Enable deployed applications to process graphics events, enabling figure windows to remain displayed

Syntax

```
void mclWaitForFiguresToDie(HMCRINSTANCE instReserved)
```

Description

Calling `void mclWaitForFiguresToDie` enables the deployed application to process graphics events.

`NULL` is the only parameter accepted for the MATLAB Runtime instance (`HMCRINSTANCE instReserved`).

This function can only be called after *`library`*`Initialize` has been called and before *`library`*`Terminate` has been called.

`mclWaitForFiguresToDie` blocks all open figures. This function runs until no visible figures remain. At that point, it displays a warning if there are invisible figures present. This function returns only when the last figure window is manually closed — therefore, this function should be called after the library launches at least one figure window. This function may be called multiple times.

If this function is not called, any figure windows initially displayed by the application briefly appear, and then the application exits.

Note `mclWaitForFiguresToDie` will block the calling program only for MATLAB figures. It will not block any Java GUIs, ActiveX controls, and other non-MATLAB GUIs unless they are embedded in a MATLAB figure window.

Examples

```
int run_main(int argc, const char** argv)
{
        int some_variable = 0;
        if (argc > 1)
             test_to_run = atoi(argv[1]);

    /* Initialize application */

    if( !mclInitializeApplication(NULL,0) )
    {
            fprintf(stderr,
                        "An error occurred while
                                initializing: \n %s ",
                        mclGetLastErrorMessage());
            return -1;
        }

        if (test_to_run == 1 || test_to_run == 0)
        {
            /* Initialize ax1ks library */
            if (!libax1ksInitialize())
            {
                    fprintf(stderr,
                            "An error occurred while
                                initializing: \n %s ",
                            mclGetLastErrorMessage());
                    return -1;
            }
        }

        if (test_to_run == 2 || test_to_run == 0)
        {
            /* Initialize simple library */
            if (!libsimpleInitialize())
            {
                    fprintf(stderr,
                            "An error occurred while
                                initializing: \n %s ",
                            mclGetLastErrorMessage());
                    return -1;
            }
        }
```

```
        /* your code here
        /* your code here
        /* your code here
        /* your code here
        /*
        /* Block on open figures */
            mclWaitForFiguresToDie(NULL);
        /* Terminate libraries */
            if (test_to_run == 1 || test_to_run == 0)
                libax1ksTerminate();
            if (test_to_run == 2 || test_to_run == 0)
                libsimpleTerminate();
        /* Terminate application */
            mclTerminateApplication();
      return(0);
}
```

See Also

Topics

"Terminating Figures by Force In an Application" (MATLAB Compiler)

Introduced in R2009a

<library>Terminate

Free all resources allocated by MATLAB Runtime instance associated with *library*

Syntax

```
void libraryTerminate(void)
```

Description

This function should be called after you finish calling the functions in this generated library, but before `mclTerminateApplication` is called.

Examples

Call `libmatrixInitialize` to initialize `libmatrix` library properly near the start of your program:

```
/* Call the library initialization routine and ensure the
*  library was initialized properly. */
if (!libmatrixInitialize())
{
    fprintf(stderr,
            "An error occurred while initializing: \n %s ",
            mclGetLastErrorMessage());
    return -2;
}
else
    ...
```

Near the end of your program (but before calling `mclTerminateApplication`) free resources allocated by the MATLAB Runtime instance associated with library `libmatrix`:

```
/* Call the library termination routine */
libmatrixTerminate();
/* Free the memory created */
```

```
mxDestroyArray(in1); in1=0;
mxDestroyArray(in2); in2 = 0;
}
```

See Also

`<library>Initialize[WithHandlers]`

Topics

"Library Initialization and Termination Functions" on page 2-30

Introduced in R2015a

C++ Utility Library Reference

Data Conversion Restrictions for the C++ MWArray API

Currently, returning a Java object to your application, from a compiled MATLAB function, is unsupported.

Primitive Types

The `mwArray` API supports all primitive types that can be stored in a MATLAB array. This table lists all the types.

Type	Description	mxClassID
`mxChar`	Character type	`mxCHAR_CLASS`
`mxLogical`	Logical or Boolean type	`mxLOGICAL_CLASS`
`mxDouble`	Double-precision floating-point type	`mxDOUBLE_CLASS`
`mxSingle`	Single-precision floating-point type	`mxSINGLE_CLASS`
`mxInt8`	1-byte signed integer	`mxINT8_CLASS`
`mxUint8`	1-byte unsigned integer	`mxUINT8_CLASS`
`mxInt16`	2-byte singed integer	`mxINT16_CLASS`
`mxUint16`	2-byte unsigned integer	`mxUINT16_CLASS`
`mxInt32`	4-byte signed integer	`mxINT32_CLASS`
`mxUint32`	4-byte unsigned integer	`mxUINT32_CLASS`
`mxInt64`	8-byte signed integer	`mxINT64_CLASS`
`mxUint64`	8-byte unsigned integer	`mxUINT64_CLASS`

C++ Utility Classes

- mwString
- mwException
- mwArray

mwString

String class used by the `mwArray` API to pass string data as output from certain methods

Description

The `mwString` class is a simple string class used by the `mwArray` API to pass string data as output from certain methods.

Required Headers

- `mclcppclass.h`
- `mclmcrrt.h`

Tip MATLAB Compiler SDK automatically includes these header files in the header file generated for your MATLAB functions.

Constructors

mwString()

Create an empty string.

mwString(char* str)

Create a new string and initialize the string's data with the supplied char buffer.

`char* str`	Null terminated character buffer

mwString(mwString& str)

Create a new string and initialize the string's data with the contents of the supplied string.

`mwString& str`	Initialized `mwString` instance

Methods

int Length() const

Return the number of characters in string.

```
mwString str("This is a string");
int len = str.Length();
```

Operators

operator const char* () const

Return a pointer to internal buffer of string.

```
mwString str("This is a string");
const char* pstr = (const char*)str;
```

mwString& operator=(const mwString& str)

Copy the contents of one string into a new string.

`mwString& str`	Initialized `mwString` instance to copy

```
mwString str("This is a string");
mwString new_str = str;
```

mwString& operator=(const char* str)

Copy the contents of a null terminated character buffer into a new string.

`char* str`	Null terminated character buffer to copy

```
const char* pstr = "This is a string";
mwString str = pstr;
```

bool operator==(const mwString& str) const

Test two `mwString` instances for equality. If the characters in the string are the same, the instances are equal.

`mwString& str`	Initialized `mwString` instance

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str == str2);
```

bool operator!=(const mwString& str) const

Test two `mwString` instances for inequality. If the characters in the string are not the same, the instances are inequal.

`mwString& str`	Initialized `mwString` instance

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str != str2);
```

bool operator<(const mwString& str) const

Compare two strings and return `true` if the first string is lexicographically less than the second string.

`mwString& str`	Initialized `mwString` instance

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str < str2);
```

bool operator<=(const mwString& str) const

Compare two strings and return `true` if the first string is lexicographically less than or equal to the second string.

`mwString& str`	Initialized `mwString` instance

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str <= str2);
```

bool operator>(const mwString& str) const

Compare two strings and return `true` if the first string is lexicographically greater than the second string.

`mwString& str`	Initialized `mwString` instance

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str > str2);
```

bool operator>=(const mwString& str) const

Compare two strings and return `true` if the first string is lexicographically greater than or equal to the second string.

`mwString& str`	Initialized `mwString` instance

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str >= str2);
```

friend std::ostream& operator<<(std::ostream& os, const mwString& str)

Copy contents of input string to specified `ostream`.

`std::ostream& os`	Initialized `ostream` instance to copy string into
`mwString& str`	Initialized `mwString` instance to copy

```
#include <ostream>
mwString str("This is a string");
std::cout << str << std::endl;
```

Introduced in R2013b

mwException

Exception type used by the `mwArray` API and the C++ interface functions

Description

The `mwException` class is the basic exception type used by the `mwArray` API and the C++ interface functions. All errors created during calls to the `mwArray` API and to generated C++ interface functions are thrown as `mwExceptions`.

Required Headers

- `mclcppclass.h`
- `mclmcrrt.h`

Tip MATLAB Compiler SDK automatically includes these header files in the header file generated for your MATLAB functions.

Constructors

mwException()

Construct new `mwException` with default error message.

mwException(char* msg)

Create an `mwException` with a specified error message.

`char* msg`	Null terminated character buffer to use as the error message

mwException(mwException& e)

Create a copy of an `mwException`.

`mwException& e`	Initialized `mwException` instance to copy

mwException(std::exception& e)

Create new `mwException` from existing `std::exception`.

`std::exception& e`	`std::exception` to copy

Methods

char* what() const throw()

Return the error message contained in this exception.

```
try
{
    ...
}
catch (const std::exception& e)
{
    std::cout << e.what() << std::endl;
}
```

void print_stack_trace()

Print the stack trace to `std::cerr`.

Operators

mwException& operator=(const mwException& e)

Copy the contents of one exception into a new exception.

`mwException& e`	An initialized `mwException` instance to copy

```
try
{
    ...
}
catch (const mwException& e)
{
    mwException e2 = e;
    throw e2;
}
```

mwException& operator=(const std::exception& e)

Copy the contents of one exception into a new exception.

`std::exception& e`	`std::exception` to copy

```
try
{
    ...
}
catch (const std::exception& e)
{
    mwException e2 = e;
    throw e2;
}
```

Introduced in R2013b

mwArray

Class used to pass input/output arguments to C++ functions generated by MATLAB Compiler SDK

Description

Use the `mwArray` class to pass input/output arguments to generated C++ interface functions. This class consists of a thin wrapper around a MATLAB array. All data in MATLAB is represented by arrays. The `mwArray` class provides the necessary constructors, methods, and operators for array creation and initialization, as well as simple indexing.

Note Arithmetic operators, such as addition and subtraction, are no longer supported as of Release 14.

Required Headers

- `mclcppclass.h`
- `mclmcrrt.h`

Tip MATLAB Compiler SDK automatically includes these header files in the header file generated for your MATLAB functions.

Constructors

mwArray()

Construct empty array of type `mxDOUBLE_CLASS`.

mwArray(mxClassID mxID)

Construct empty array of specified type.

`mxClassID mxID`	Valid `mxClassID` specifying the type of array to construct. See "Work with mxArrays" (MATLAB) for more information on `mxClassID`.

mwArray(mwSize num_rows, mwSize num_cols, mxClassID mxID, mxComplexity cmplx = mxREAL)

Create a 2–D matrix of the specified type and complexity. For nonnumeric types, `mxComplexity` will be ignored. For numeric types, pass `mxCOMPLEX` for the last argument to create a complex matrix; otherwise, the matrix will be real. All elements are initialized to zero. For cell matrices, all elements are initialized to empty cells.

`mwSize num_rows`	Number of rows in the array
`mwSize num_cols`	Number of columns in the array
`mxClassID mxID`	Valid `mxClassID` specifying the type of array to construct. See "Work with mxArrays" (MATLAB) for more information on `mxClassID`.
`mxComplexity cmplx`	Complexity of the array to create. Valid values are `mxREAL` and `mxCOMPLEX`. The default value is `mxREAL`.

mwArray(mwSize num_dims, const mwSize* dims, mxClassID mxID, mxComplexity cmplx = mxREAL)

Create an `n`-dimensional array of the specified type and complexity. For nonnumeric types, `mxComplexity` will be ignored. For numeric types, pass `mxCOMPLEX` for the last argument to create a complex matrix; otherwise, the array will be real. All elements are initialized to zero. For cell arrays, all elements are initialized to empty cells.

`mwSize num_dims`	Number of dimensions in the array
`const mwSize* dims`	Dimensions of the array
`mxClassID mxID`	Valid `mxClassID` specifying the type of array to construct. See "Work with mxArrays" (MATLAB) for more information on `mxClassID`.
`mxComplexity cmplx`	Complexity of the array to create. Valid values are `mxREAL` and `mxCOMPLEX`. The default value is `mxREAL`.

mwArray(const char* str)

Create a 1-by-*n* array of type `mxCHAR_CLASS`, with `n = strlen(str)`, and initialize the array's data with the characters in the supplied string.

`const char* str`	Null-terminated character buffer used to initialize the array

mwArray(mwSize num_strings, const char** str)

Create a matrix of type `mxCHAR_CLASS`, and initialize the array's data with the characters in the supplied strings. The created array has dimensions `m`-by-`max`, where `m` is the number of strings and `max` is the length of the longest string in `str`.

`mwSize num_strings`	Number of strings in the input array
`const char** str`	Array of null-terminated strings

mwArray(mwSize num_rows, mwSize num_cols, int num_fields, const char** fieldnames)

Create a matrix of type `mxSTRUCT_CLASS`, with the specified field names. All elements are initialized with empty cells.

`mwSize num_rows`	Number of rows in the array
`mwSize num_cols`	Number of columns in the array
`int num_fields`	Number of fields in the `struct` matrix.
`const char** fieldnames`	Array of null-terminated strings representing the field names

mwArray(mwSize num_dims, const mwSize* dims, int num_fields, const char** fieldnames)

Create an `n`-dimensional array of type `mxSTRUCT_CLASS`, with the specified field names. All elements are initialized with empty cells.

`mwSize num_dims`	Number of dimensions in the array
`const mwSize* dims`	Dimensions of the array
`int num_fields`	Number of fields in the `struct` matrix.
`const char** fieldnames`	Array of null-terminated strings representing the field names

mwArray(const mwArray& arr)

Create a deep copy of an existing array.

`mwArray& arr`	`mwArray` to copy

mwArray(<type> re)

Create a real scalar array.

The scalar array is created with the type of the input argument.

`<type> re`	Scalar value to initialize the array. `<type>` can be any of the following: • `mxDouble` • `mxSingle` • `mxInt8` • `mxUint8` • `mxInt16` • `mxUint16` • `mxInt32` • `mxUint32` • `mxInt64` • `mxUint64` • `mxLogical`

mwArray(<type> re, <type> im)

Create a complex scalar array.

The scalar array is created with the type of the input argument.

`<type> re`	Scalar value to initialize the real part of the array
`<type> im`	Scalar value to initialize the imaginary part of the array

`<type>` can be any of the following:

- `mxDouble`
- `mxSingle`
- `mxInt8`
- `mxUint8`
- `mxInt16`

- mxUint16
- mxInt32
- mxUint32
- mxInt64
- mxUint64
- mxLogical

Methods

mwArray Clone() const

Create a new array representing deep copy of array.

```
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray b = a.Clone();
```

mwArray SharedCopy() const

Create a shared copy of an existing array. The new array and the original array both point to the same data.

```
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray b = a.SharedCopy();
```

mwArray Serialize() const

Serialize an array into bytes. A 1-by-`n` numeric matrix of type `mxUINT8_CLASS` is returned containing the serialized data. The data can be deserialized back into the original representation by calling `mwArray::Deserialize()`.

```
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray b = a.Serialize();
```

mxClassID ClassID() const

Determine the type of the array. See “Work with mxArrays” (MATLAB) for more information on `mxClassID`.

```
mwArray a(2, 2, mxDOUBLE_CLASS);
mxClassID id = a.ClassID();
```

size_t ElementSize() const

Determine the size, in bytes, of an element of array type. If the array is complex, the return value will represent the size, in bytes, of the real part of an element.

```
mwArray a(2, 2, mxDOUBLE_CLASS);
int size = a.ElementSize();
```

mwSize NumberOfElements() const

Determine the total size of the array.

```
mwArray a(2, 2, mxDOUBLE_CLASS);
int n = a.NumberOfElements();
```

mwSize NumberOfNonZeros() const

Determine the size of the array's data. If the underlying array is not sparse, this returns the same value as `NumberOfElements()`.

```
mwArray a(2, 2, mxDOUBLE_CLASS);
int n = a.NumberOfNonZeros();
```

mwSize MaximumNonZeros() const

Determine the allocated size of the array's data. If the underlying array is not sparse, this returns the same value as `NumberOfElements()`.

```
mwArray a(2, 2, mxDOUBLE_CLASS);
int n = a.MaximumNonZeros();
```

mwSize NumberOfDimensions() const

Determine the dimensionality of the array.

```
mwArray a(2, 2, mxDOUBLE_CLASS);
int n = a.NumberOfDimensions();
```

int NumberOfFields() const

Determine the number of fields in a `struct` array. If the underlying array is not of type `struct`, zero is returned.

```
const char* fields[] = {"a", "b", "c"};
mwArray a(2, 2, 3, fields);
int n = a.NumberOfFields();
```

mwString GetFieldName(int index)

Determine the name of a given field in a `struct` array. If the underlying array is not of type `struct`, an exception is thrown.

`int index`	Index of the field to name. Indexing starts at zero.

```
const char* fields[] = {"a", "b", "c"};
mwArray a(2, 2, 3, fields);
```

```
mwString tempname = a.GetFieldName(1);
const char* name = (const char*)tempname;
```

mwArray GetDimensions() const

Determine the size of each dimension in the array. The size of the returned array is 1-by-`NumberOfDimensions()`.

```
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray dims = a.GetDimensions();
```

bool IsEmpty() const

Determine if an array is empty.

```
mwArray a;
bool b = a.IsEmpty();
```

bool IsSparse() const

Determine if an array is sparse.

```
mwArray a(2, 2, mxDOUBLE_CLASS);
bool b = a.IsSparse();
```

bool IsNumeric() const

Determine if an array is numeric.

```
mwArray a(2, 2, mxDOUBLE_CLASS);
bool b = a.IsNumeric();
```

bool IsComplex() const

Determine if an array is complex.

```
mwArray a(2, 2, mxDOUBLE_CLASS, mxCOMPLEX);
bool b = a.IsComplex();
```

bool Equals(const mwArray& arr) const

Returns `true` if the input array is byte-wise equal to this array. This method makes a byte-wise comparison of the underlying arrays. Therefore, arrays of the same type should be compared. Arrays of different types will not in general be equal, even if they are initialized with the same data.

`mwArray& arr`	Array to compare to array.

```
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray b(1, 1, mxDOUBLE_CLASS);
a = 1.0;
b = 1.0;
bool c = a.Equals(b);
```

int CompareTo(const mwArray& arr) const

Compares this array with the specified array for order. This method makes a byte-wise comparison of the underlying arrays. Therefore, arrays of the same type should be compared. Arrays of different types will, in general, not be ordered equivalently, even if they are initialized with the same data.

`mwArray& arr`	Array to compare to array.

```
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray b(1, 1, mxDOUBLE_CLASS);
a = 1.0;
b = 1.0;
int n = a.CompareTo(b);
```

int HashCode() const

Constructs a unique hash value from the underlying bytes in the array. Therefore, arrays of different types will have different hash codes, even if they are initialized with the same data.

```
mwArray a(1, 1, mxDOUBLE_CLASS);
int n = a.HashCode();
```

mwString ToString() const

Returns a string representation of the underlying array. The string returned is the same one that is returned by typing a variable's name at the MATLAB command prompt.

```
mwArray a(1, 1, mxDOUBLE_CLASS, mxCOMPLEX);
a.Real() = 1.0;
a.Imag() = 2.0;
printf("%s\n", (const char*)(a.ToString()));
```

mwArray RowIndex() const

Returns an array representing the row indices (first dimension) of the elements of this array in column-major order. For sparse arrays, the indices are returned for just the non-zero elements and the size of the array returned is 1-by-`NumberOfNonZeros()`. For nonsparse arrays, the size of the array returned is 1-by-`NumberOfElements()`, and the row indices of all of the elements are returned.

```
#include <stdio.h>
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray rows = a.RowIndex();
```

mwArray ColumnIndex() const

Returns an array representing the column indices (second dimension) of the elements of this array in column-major order. For sparse arrays, the indices are returned for just the

non-zero elements and the size of the array returned is 1-by-`NumberOfNonZeros()`. For nonsparse arrays, the size of the array returned is 1-by-`NumberOfElements()`, and the column indices of all of the elements are returned.

```
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray rows = a.ColumnIndex();
```

void MakeComplex()

Convert a numeric array that has been previously allocated as `real` to `complex`. If the underlying array is of a nonnumeric type, an `mwException` is thrown.

```
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double idata[4] = {10.0, 20.0, 30.0, 40.0};
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(rdata, 4);
a.MakeComplex();
a.Imag().SetData(idata, 4);
```

mwArray Get(mwSize num_indices, ...)

Fetches a single element at a specified index. The number of indices is passed, followed by a comma-separated list of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing) or `NumberOfDimensions()` (multiple subscript indexing). In single subscript indexing the element at the specified 1-based offset is returned, accessing data in column-major order. In multiple subscript indexing the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the *i*th index has the valid range: `1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

`mwSize num_indices`	Number of indices passed in
`...`	Comma-separated list of input indices. Number of items must equal `num_indices` but should not exceed 32.

```
double data[4] = {1.0, 2.0, 3.0, 4.0};
double x;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(data, 4);
x = a.Get(1,1);
x = a.Get(2, 1, 2);
x = a.Get(2, 2, 2);
```

mwArray Get(const char* name, mwSize num_indices, ...)

Fetches a single element at a specified field name and index. This method may only be called on an array that is of type `mxSTRUCT_CLASS`. An `mwException` is thrown if the underlying array is not a `struct` array. The field name passed must be a valid field name in the `struct` array. The index is passed by first passing the number of indices followed by a comma-separated list of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing) or `NumberOfDimensions()` (multiple subscript indexing). In single subscript indexing the element at the specified 1-based offset is returned, accessing data in column-wise order. In multiple subscript indexing the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the ith index has the valid range:
`1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

`char* name`	Null-terminated character buffer containing the name of the field
`mwSize num_indices`	Number of indices passed in
`...`	Comma-separated list of input indices. Number of items must equal `num_indices` but should not exceed 32.

```
const char* fields[] = {"a", "b", "c"};

mwArray a(1, 1, 3, fields);
mwArray b = a.Get("a", 1, 1);
mwArray b = a.Get("b", 2, 1, 1);
```

mwArray Get(mwSize num_indices, const mwIndex* index)

Fetches a single element at a specified index. The index is passed by first passing the number of indices, followed by an array of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing) or `NumberOfDimensions()` (multiple subscript indexing). In single subscript indexing the element at the specified 1-based offset is returned, accessing data in column-wise order. In multiple subscript indexing the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the ith index has the valid range:
`1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

`mwSize num_indices`	Size of index array
`mwIndex* index`	Array of at least size `num_indices` containing the indices

```
double data[4] = {1.0, 2.0, 3.0, 4.0};
int index[2] = {1, 1};
double x;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(data, 4);
x = a.Get(1, index);
x = a.Get(2, index);
index[0] = 2;
index[1] = 2;
x = a.Get(2, index);
```

mwArray Get(const char* name, mwSize num_indices, const mwIndex* index)

Fetches a single element at a specified field name and index. This method may only be called on an array that is of type `mxSTRUCT_CLASS`. An `mwException` is thrown if the underlying array is not a `struct` array. The field name passed must be a valid field name in the `struct` array. The index is passed by first passing the number of indices followed by an array of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing) or `NumberOfDimensions()` (multiple subscript

indexing). In single subscript indexing the element at the specified 1-based offset is returned, accessing data in column-wise order. In multiple subscript indexing the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the ith index has the valid range: `1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

`char* name`	Null-terminated character buffer containing the name of the field
`mwSize num_indices`	Number of indices passed in
`mwIndex* index`	Array of at least size `num_indices` containing the indices

```
const char* fields[] = {"a", "b", "c"};
int index[2] = {1, 1};
mwArray a(1, 1, 3, fields);
mwArray b = a.Get("a", 1, index);
mwArray b = a.Get("b", 2, index);
```

mwArray Real()

Accesses the real part of a complex array. The returned `mwArray` is considered real and has the same dimensionality and type as the original.

Complex arrays consist of Complex numbers, which are 1-by-2 vectors (pairs). For example, if the number is `3+5i`, then the pair is `(3,5i)`. An array of Complex numbers is therefore two dimensional `(N-by-2)`, where *N* is the number of complex numbers in the array. `2+4i, 7-3i, 8+6i` would be represented as `(2,4i) (7,3i) (8,6i)`. Complex numbers have two components, real and imaginary.

```
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double idata[4] = {10.0, 20.0, 30.0, 40.0};
mwArray a(2, 2, mxDOUBLE_CLASS, mxCOMPLEX);
a.Real().SetData(rdata, 4);
```

mwArray Imag()

Accesses the imaginary part of a complex array. The returned `mwArray` is considered real and has the same dimensionality and type as the original.

Complex arrays consist of Complex numbers, which are 1-by-2 vectors (pairs). For example, if the number is `3+5i`, then the pair is `(3,5i)`. An array of Complex numbers is therefore two dimensional `(N-by-2)`, where *N* is the number of complex numbers in the array. `2+4i, 7-3i, 8+6i` would be represented as `(2,4i) (7,3i) (8,6i)`. Complex numbers have two components, real and imaginary.

```
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double idata[4] = {10.0, 20.0, 30.0, 40.0};
mwArray a(2, 2, mxDOUBLE_CLASS, mxCOMPLEX);
a.Imag().SetData(idata, 4);
```

void Set(const mwArray& arr)

Assign shared copy of input array to currently referenced cell for arrays of type `mxCELL_CLASS` and `mxSTRUCT_CLASS`.

`mwArray& arr`	`mwArray` to assign to currently referenced cell

```
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray b(2, 2, mxINT16_CLASS);
mwArray c(1, 2, mxCELL_CLASS);
c.Get(1,1).Set(a);
c.Get(1,2).Set(b);
```

void GetData(<numeric-type>* buffer, mwSize len) const

Copies the array's data into supplied numeric buffer.

The data is copied in column-major order. If the underlying array is not of the same type as the input buffer, the data is converted to this type as it is copied. If a conversion cannot be made, an `mwException` is thrown.

<numeric-type>* buffer	Buffer to receive copy. Valid types for <numeric-type> are: • mxDOUBLE_CLASS • mxSINGLE_CLASS • mxINT8_CLASS • mxUINT8_CLASS • mxINT16_CLASS • mxUINT16_CLASS • mxINT32_CLASS • mxUINT32_CLASS • mxINT64_CLASS • mxUINT64_CLASS
mwSize len	Maximum length of buffer. A maximum of len elements will be copied.

```
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double data_copy[4] ;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(rdata, 4);
a.GetData(data_copy, 4);
```

void GetLogicalData(mxLogical* buffer, mwSize len) const

Copies the array's data into supplied mxLogical buffer.

The data is copied in column-major order. If the underlying array is not of type mxLOGICAL_CLASS, the data is converted to this type as it is copied. If a conversion cannot be made, an mwException is thrown.

mxLogical* buffer	Buffer to receive copy
mwSize len	Maximum length of buffer. A maximum of len elements will be copied.

```
mxLogical data[4] = {true, false, true, false};
mxLogical data_copy[4] ;
mwArray a(2, 2, mxLOGICAL_CLASS);
a.SetLogicalData(data, 4);
a.GetLogicalData(data_copy, 4);
```

void GetCharData(mxChar* buffer, mwSize len) const

Copies the array's data into supplied `mxChar` buffer.

The data is copied in column-major order. If the underlying array is not of type `mxCHAR_CLASS`, the data is converted to this type as it is copied. If a conversion cannot be made, an `mwException` is thrown.

`mxChar** buffer`	Buffer to receive copy
`mwSize len`	Maximum length of buffer. A maximum of `len` elements will be copied.

```
mxChar data[6] = {'H', 'e' , `l' , 'l' , 'o' , '\0'};
mxChar data_copy[6] ;
mwArray a(1, 6, mxCHAR_CLASS);
a.SetCharData(data, 6);
a.GetCharData(data_copy, 6);
```

void SetData(<numeric-type>* buffer, mwSize len)

Copies the data from supplied numeric buffer into the array.

The data is copied in column-major order. If the underlying array is not of the same type as the input buffer, the data is converted to this type as it is copied. If a conversion cannot be made, an `mwException` is thrown.

`<numeric-type>* buffer`	Buffer containing data to copy. Valid types for `<numeric-type>` are: • `mxDOUBLE_CLASS` • `mxSINGLE_CLASS` • `mxINT8_CLASS` • `mxUINT8_CLASS` • `mxINT16_CLASS` • `mxUINT16_CLASS` • `mxINT32_CLASS` • `mxUINT32_CLASS` • `mxINT64_CLASS` • `mxUINT64_CLASS`
`mwSize len`	Maximum length of buffer. A maximum of `len` elements will be copied.

```
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double data_copy[4] ;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(rdata, 4);
a.GetData(data_copy, 4);
```

void SetLogicalData(mxLogical* buffer, mwSize len)

Copies the data from the supplied `mxLogical` buffer into the array.

The data is copied in column-major order. If the underlying array is not of type `mxLOGICAL_CLASS`, the data is converted to this type as it is copied. If a conversion cannot be made, an `mwException` is thrown.

`mxLogical* buffer`	Buffer containing data to copy
`mwSize len`	Maximum length of buffer. A maximum of `len` elements will be copied.

```
mxLogical data[4] = {true, false, true, false};
mxLogical data_copy[4] ;
mwArray a(2, 2, mxLOGICAL_CLASS);
a.SetLogicalData(data, 4);
a.GetLogicalData(data_copy, 4);
```

void SetCharData(mxChar* buffer, mwSize len)

Copies the data from the supplied `mxChar` buffer into the array.

The data is copied in column-major order. If the underlying array is not of type `mxCHAR_CLASS`, the data is converted to this type as it is copied. If a conversion cannot be made, an `mwException` is thrown.

`mxChar** buffer`	Buffer containing data to copy
`mwSize len`	Maximum length of buffer. A maximum of `len` elements will be copied.

```
mxChar data[6] = {'H', 'e' , `l' , 'l' , 'o' , '\0'};
mxChar data_copy[6] ;
mwArray a(1, 6, mxCHAR_CLASS);
a.SetCharData(data, 6);
a.GetCharData(data_copy, 6);
```

static mwArray Deserialize(const mwArray& arr)

Deserializes an array that has been serialized with `mwArray::Serialize()`. The input array must be of type `mxUINT8_CLASS` and contain the data from a serialized array. If the input data does not represent a serialized `mwArray`, the behavior of this method is undefined.

`mwArray& arr`	`mwArray` that has been obtained by calling `mwArray::Serialize`

```
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
mwArray a(1,4,mxDOUBLE_CLASS);
```

```
a.SetData(rdata, 4);
mwArray b = a.Serialize();
a = mwArray::Deserialize(b);
```

static mwArray NewSparse(mwSize rowindex_size, const mwIndex* rowindex, mwSize colindex_size, const mwIndex* colindex, mwSize data_size, const mxDouble* rdata, mwSize num_rows, mwSize num_cols, mwSize nzmax)

Creates real sparse matrix of type `double` with specified number of rows and columns.

The lengths of input row, column index, and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated throughout the construction of the matrix.

If the same row/column pair occurs more than once, the data value assigned to that element is the sum of all values associated with that pair. If any element of the `rowindex` or `colindex` array is greater than the specified values in `num_rows` or `num_cols` respectively, an exception is thrown.

`mwSize rowindex_size`	Size of `rowindex` array
`mwIndex* rowindex`	Array of row indices of non-zero elements
`mwSize colindex_size`	Size of `colindex` array
`mwIndex* colindex`	Array of column indices of non-zero elements
`mwSize data_size`	Size of data array
`mxDouble* rdata`	Data associated with non-zero row and column indices
`mwSize num_rows`	Number of rows in matrix
`mwSize num_cols`	Number of columns in matrix
`mwSize nzmax`	Reserved storage for sparse matrix. If `nzmax` is zero, storage will be set to `max{rowindex_size, colindex_size, data_size}`.

This example constructs a sparse 4-by-4 tridiagonal matrix:

```
2 -1  0  0
-1  2 -1  0
0 -1  2 -1
0  0 -1  2
```

The following code, when run:

```
double rdata[]        =
           {2.0, -1.0, -1.0, 2.0, -1.0,
            -1.0, 2.0, -1.0, -1.0, 2.0};
mwIndex row_tridiag[] =
           {1,    2,    1,   2,    3,
            2,   3,    4,    3,   4  };
mwIndex col_tridiag[] =
           {1,    1,    2,   2,    2,
              3,   3,    3,    4,   4  };

mwArray mysparse =
           mwArray::NewSparse(10, row_tridiag,
                              10, col_tridiag,
                              10, rdata, 4, 4, 10);
std::cout << mysparse << std::endl;
```

will display the following output to the screen:

```
 (1,1)        2
 (2,1)       -1
 (1,2)       -1
 (2,2)        2
 (3,2)       -1
 (2,3)       -1
 (3,3)        2
 (4,3)       -1
 (3,4)       -1
 (4,4)        2
```

static mwArray NewSparse(mwSize rowindex_size, const mwIndex* rowindex, mwSize colindex_size, const mwIndex* colindex, mwSize data_size, const mxDouble* rdata, mwSize nzmax)

Creates real sparse matrix of type `double` with number of rows and columns inferred from input data.

The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated through out the construction of the matrix.

If the same row/column pair occurs more than once, the data value assigned to that element is the sum of all values associated with that pair. The number of rows and columns in the created matrix are calculated from the input `rowindex` and `colindex` arrays as `num_rows = max{rowindex}, num_cols = max{colindex}`.

`mwSize rowindex_size`	Size of `rowindex` array
`mwIndex* rowindex`	Array of row indices of non-zero elements
`mwSize colindex_size`	Size of `colindex` array
`mwIndex* colindex`	Array of column indices of non-zero elements
`mwSize data_size`	Size of data array
`mxDouble* rdata`	Data associated with non-zero row and column indices
`mwSize nzmax`	Reserved storage for sparse matrix. If `nzmax` is zero, storage will be set to `max{rowindex_size, colindex_size, data_size}`.

In this example, we construct a sparse 4-by-4 identity matrix. The value of 1.0 is copied to each non-zero element defined by row and column index arrays:

```
double one = 1.0;
mwIndex row_diag[] = {1, 2, 3, 4};
mwIndex col_diag[] = {1, 2, 3, 4};
```

```
mwArray mysparse =
  mwArray::NewSparse(4, row_diag,
                     4, col_diag,
                     1, &one,
                     0);
std::cout << mysparse << std::endl;

(1,1)         1
(2,2)         1
(3,3)         1
(4,4)         1
```

static mwArray NewSparse(mwSize rowindex_size, const mwIndex* rowindex, mwSize colindex_size, const mwIndex* colindex, mwSize data_size, const mxDouble* rdata, const mxDouble* idata, mwSize num_rows, mwSize num_cols, mwSize nzmax)

Creates complex sparse matrix of type `double` with specified number of rows and columns.

The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated through out the construction of the matrix.

If the same row/column pair occurs more than once, the data value assigned to that element is the sum of all values associated with that pair. If any element of the `rowindex` or `colindex` array is greater than the specified values in `num_rows`, `num_cols`, respectively, then an exception is thrown.

`mwSize rowindex_size`	Size of `rowindex` array
`mwIndex* rowindex`	Array of row indices of non-zero elements
`mwSize colindex_size`	Size of `colindex` array
`mwIndex* colindex`	Array of column indices of non-zero elements
`mwSize data_size`	Size of data array
`mxDouble* rdata`	Real part of data associated with non-zero row and column indices

`mxDouble* idata`	Imaginary part of data associated with non-zero row and column indices
`mwSize num_rows`	Number of rows in matrix
`mwSize num_cols`	Number of columns in matrix
`mwSize nzmax`	Reserved storage for sparse matrix. If `nzmax` is zero, storage will be set to `max{rowindex_size, colindex_size, data_size}`.

This example constructs a complex tridiagonal matrix:

```
double rdata[] =
  {2.0, -1.0, -1.0, 2.0, -1.0, -1.0, 2.0, -1.0, -1.0, 2.0};
double idata[] =
  {20.0, -10.0, -10.0, 20.0, -10.0, -10.0, 20.0, -10.0,
                                              -10.0, 20.0};
mwIndex row_tridiag[] =
  {1,    2,    1,   2,    3,    2,   3,    4,    3,   4};
mwIndex col_tridiag[] =
  {1,    1,    2,   2,    2,    3,   3,    3,    4,   4};

mwArray mysparse = mwArray::NewSparse(10, row_tridiag,
                                      10, col_tridiag,
                                      10, rdata,
                                      idata, 4, 4, 10);
std::cout << mysparse << std::endl;
```

It displays the following output to the screen:

```
(1,1)      2.0000 +20.0000i
(2,1)     -1.0000 -10.0000i
(1,2)     -1.0000 -10.0000i
(2,2)      2.0000 +20.0000i
(3,2)     -1.0000 -10.0000i
(2,3)     -1.0000 -10.0000i
(3,3)      2.0000 +20.0000i
(4,3)     -1.0000 -10.0000i
(3,4)     -1.0000 -10.0000i
(4,4)      2.0000 +20.0000i
```

static mwArray NewSparse(mwSize rowindex_size, const mwIndex* rowindex, mwSize colindex_size, const mwIndex* colindex, mwSize data_size, const mxDouble* rdata, const mxDouble* idata, mwSize nzmax)

Creates complex sparse matrix of type `double` with number of rows and columns inferred from input data.

The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated through out the construction of the matrix.

If the same row/column pair occurs more than once, the data value assigned to that element is the sum of all values associated with that pair. The number of rows and columns in the created matrix are calculated form the input `rowindex` and `colindex` arrays as `num_rows = max{rowindex}, num_cols = max{colindex}`.

`mwSize rowindex_size`	Size of `rowindex` array
`mwIndex* rowindex`	Array of row indices of non-zero elements
`mwSize colindex_size`	Size of `colindex` array
`mwIndex* colindex`	Array of column indices of non-zero elements
`mwSize data_size`	Size of data array
`mxDouble* rdata`	Real part of data associated with non-zero row and column indices
`mxDouble* idata`	Imaginary part of data associated with non-zero row and column indices
`mwSize nzmax`	Reserved storage for sparse matrix. If `nzmax` is zero, storage will be set to `max{rowindex_size, colindex_size, data_size}`.

This example constructs a complex matrix by inferring dimensions and storage allocation from the input data.

```
mwArray mysparse =
   mwArray::NewSparse(10, row_tridiag,
                      10, col_tridiag,
                      10, rdata, idata,
                      0);
std::cout << mysparse << std::endl;

(1,1)       2.0000 +20.0000i
(2,1)      -1.0000 -10.0000i
(1,2)      -1.0000 -10.0000i
(2,2)       2.0000 +20.0000i
(3,2)      -1.0000 -10.0000i
(2,3)      -1.0000 -10.0000i
(3,3)       2.0000 +20.0000i
(4,3)      -1.0000 -10.0000i
(3,4)      -1.0000 -10.0000i
(4,4)       2.0000 +20.0000i
```

static mwArray NewSparse(mwSize rowindex_size, const mwIndex* rowindex, mwSize colindex_size, const mwIndex* colindex, mwSize data_size, const mxLogical* rdata, mwSize num_rows, mwSize num_cols, mwSize nzmax)

Creates logical sparse matrix with specified number of rows and columns.

The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated throughout the construction of the matrix.

If the same row/column pair occurs more than once, the data value assigned to that element is the sum of all values associated with that pair. If any element of the `rowindex` or `colindex` array is greater than the specified values in `num_rows`, `num_cols`, respectively, then an exception is thrown.

`mwSize rowindex_size`	Size of `rowindex` array
`mwIndex* rowindex`	Array of row indices of non-zero elements
`mwSize colindex_size`	Size of `colindex` array

`mwIndex* colindex`	Array of column indices of non-zero elements
`mwSize data_size`	Size of data array
`mxLogical* rdata`	Data associated with non-zero row and column indices
`mwSize num_rows`	Number of rows in matrix
`mwSize num_cols`	Number of columns in matrix
`mwSize nzmax`	Reserved storage for sparse matrix. If `nzmax` is zero, storage will be set to `max{rowindex_size, colindex_size, data_size}`.

This example creates a sparse logical 4-by-4 tridiagonal matrix, assigning `true` to each non-zero value:

```
mxLogical one = true;
mwIndex row_tridiag[] =
      {1,    2,    1,   2,    3,
       2,   3,    4,    3,   4};
mwIndex col_tridiag[] =
      {1,    1,    2,   2,    2,
       3,   3,    3,    4,   4};

mwArray mysparse =
      mwArray::NewSparse(10, row_tridiag,
                         10, col_tridiag,
                          1, &one,
                          4, 4, 10);
std::cout << mysparse << std::endl;

(1,1)        1
(2,1)        1
(1,2)        1
(2,2)        1
(3,2)        1
(2,3)        1
(3,3)        1
(4,3)        1
(3,4)        1
(4,4)        1
```

static mwArray NewSparse(mwSize rowindex_size, const mwIndex* rowindex, mwSize colindex_size, const mwIndex* colindex, mwSize data_size, const mxLogical* rdata, mwSize nzmax)

Creates logical sparse matrix with number of rows and columns inferred from input data.

The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated through out the construction of the matrix.

The number of rows and columns in the created matrix are calculated form the input `rowindex` and `colindex` arrays as `num_rows = max {rowindex}, num_cols = max {colindex}`.

`mwSize rowindex_size`	Size of `rowindex` array
`mwIndex* rowindex`	Array of row indices of non-zero elements
`mwSize colindex_size`	Size of `colindex` array
`mwIndex* colindex`	Array of column indices of non-zero elements
`mwSize data_size`	Size of data array
`mxLogical* rdata`	Data associated with non-zero row and column indices
`mwSize nzmax`	Reserved storage for sparse matrix. If `nzmax` is zero, storage will be set to `max{rowindex_size, colindex_size, data_size}`.

This example uses the data from the first example, but allows the number of rows, number of columns, and allocated storage to be calculated from the input data:

```
mwArray mysparse =
    mwArray::NewSparse(10, row_tridiag,
                       10, col_tridiag,
                       1, &one,
                       0);
std::cout << mysparse << std::endl;
```

```
(1,1)         1
(2,1)         1
(1,2)         1
(2,2)         1
(3,2)         1
(2,3)         1
(3,3)         1
(4,3)         1
(3,4)         1
(4,4)         1
```

static mwArray NewSparse (mwSize num_rows, mwSize num_cols, mwSize nzmax, mxClassID mxID, mxComplexity cmplx = mxREAL)

Creates an empty sparse matrix. All elements in an empty sparse matrix are initially zero, and the amount of allocated storage for non-zero elements is specified by `nzmax`.

`mwSize num_rows`	Number of rows in matrix
`mwSize num_cols`	Number of columns in matrix
`mwSize nzmax`	Reserved storage for sparse matrix
`mxClassID mxID`	Type of data to store in matrix. Currently, sparse matrices of type `double` precision and `logical` are supported. Pass `mxDOUBLE_CLASS` to create a `double` precision sparse matrix. Pass `mxLOGICAL_CLASS` to create a `logical` sparse matrix.
`mxComplexity cmplx`	Complexity of matrix. Pass `mxCOMPLEX` to create a `complex` sparse matrix and `mxREAL` to create a `real` sparse matrix. This argument may be omitted, in which case the default complexity is `real`

This example constructs a real 3-by-3 empty sparse matrix of type `double` with reserved storage for 4 non-zero elements:

```
mwArray mysparse = mwArray::NewSparse
                (3, 3, 4, mxDOUBLE_CLASS);
std::cout << mysparse << std::endl;

All zero sparse: 3-by-3
```

static double GetNaN()

Get value of `NaN` (Not-a-Number).

Call `mwArray::GetNaN` to return the value of `NaN` for your system. `NaN` is the IEEE arithmetic representation for Not-a-Number. Certain mathematical operations return `NaN` as a result, for example:

- 0.0/0.0
- `Inf-Inf`

The value of `NaN` is built in to the system; you cannot modify it.

```
double x = mwArray::GetNaN();
```

static double GetEps()

Returns the value of the MATLAB `eps` variable. This variable is the distance from 1.0 to the next largest floating-point number. Consequently, it is a measure of floating-point accuracy. The MATLAB `pinv` and `rank` functions use `eps` as a default tolerance.

```
double x = mwArray::GetEps();
```

static double GetInf()

Returns the value of the MATLAB internal `Inf` variable. `Inf` is a permanent variable representing IEEE arithmetic positive infinity. The value of `Inf` is built into the system; you cannot modify it.

Operations that return `Inf` include

- Division by 0. For example, 5/0 returns `Inf`.
- Operations resulting in overflow. For example, `exp(10000)` returns `Inf` because the result is too large to be represented on your machine.

```
double x = mwArray::GetInf();
```

static bool IsFinite(double x)

Determine whether or not a value is finite. A number is finite if it is greater than `-Inf` and less than `Inf`.

`double x`	Value to test for finiteness

```
bool x = mwArray::IsFinite(1.0);
```

static bool IsInf(double x)

Determines whether or not a value is equal to infinity or minus infinity. MATLAB stores the value of infinity in a permanent variable named `Inf`, which represents IEEE arithmetic positive infinity. The value of the variable, `Inf`, is built into the system; you cannot modify it.

Operations that return infinity include

- Division by 0. For example, 5/0 returns infinity.
- Operations resulting in overflow. For example, `exp(10000)` returns infinity because the result is too large to be represented on your machine. If the value equals `NaN` (Not-a-Number), then `mxIsInf` returns `false`. In other words, `NaN` is not equal to infinity.

`double x`	Value to test for infiniteness

```
bool x = mwArray::IsInf(1.0);
```

static bool IsNaN(double x)

Determines whether or not the value is `NaN`. `NaN` is the IEEE arithmetic representation for Not-a-Number. `NaN` is obtained as a result of mathematically undefined operations such as

- `0.0/0.0`
- `Inf-Inf`

The system understands a family of bit patterns as representing `NaN`. In other words, `NaN` is not a single value, rather it is a family of numbers that the MATLAB software (and other IEEE-compliant applications) use to represent an error condition or missing data.

`double x`	Value to test for `NaN`

```
bool x = mwArray::IsNaN(1.0);
```

Operators

mwArray operator()(mwIndex i1, mwIndex i2, mwIndex i3, ...,)

Fetches a single element at a specified index. The index is passed as a comma-separated list of 1-based indices. This operator is overloaded to support 1 through 32 indices. The valid number of indices that can be passed in is either 1 (single subscript indexing) or `NumberOfDimensions()` (multiple subscript indexing). In single subscript indexing the element at the specified 1-based offset is returned, accessing data in column-wise order. In multiple subscript indexing the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the ith index has the valid range: `1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

`mwIndex i1, mwIndex i2, mwIndex i3, ...,`	Comma-separated list of input indices

```
double data[4] = {1.0, 2.0, 3.0, 4.0};
double x;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(data, 4);
x = a(1,1);
x = a(1,2);
x = a(2,2);
```

mwArray operator()(const char* name, mwIndex i1, mwIndex i2, mwIndex i3, ...,)

Fetches a single element at a specified field name and index. This method may only be called on an array that is of type `mxSTRUCT_CLASS`. An `mwException` is thrown if the underlying array is not a `struct` array. The field name passed must be a valid field name in the `struct` array. The index is passed by first passing the number of indices, followed by an array of 1-based indices. This operator is overloaded to support 1 through 32 indices. The valid number of indices that can be passed in is either 1 (single subscript indexing) or `NumberOfDimensions()` (multiple subscript indexing). In single subscript indexing the element at the specified 1-based offset is returned, accessing data in column-wise order. In multiple subscript indexing the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the ith index has the valid range: `1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

`char* name`	Null terminated string containing the field name to get
`mwIndex i1, mwIndex i2, mwIndex i3, ...,`	Comma-separated list of input indices

```
const char* fields[] = {"a", "b", "c"};
int index[2] = {1, 1};
mwArray a(1, 1, 3, fields);
mwArray b = a("a", 1, 1);
mwArray b = a("b", 1, 1);
```

mwArray& operator=(const <type>& x)

Sets a single scalar value. This operator is overloaded for all numeric and logical types.

`const <type>& x`	Value to assign

```
mwArray a(2, 2, mxDOUBLE_CLASS);
a(1,1) = 1.0;
a(1,2) = 2.0;
a(2,1) = 3.0;
a(2,2) = 4.0;
```

const mwArray operator()(mwIndex i1, mwIndex i2, mwIndex i3, ...,) const

Fetches a single scalar value. This operator is overloaded for all numeric and logical types.

`mwIndex i1, mwIndex i2, mwIndex i3, ...,`	Comma-separated list of input indices

```
double data[4] = {1.0, 2.0, 3.0, 4.0};
double x;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(data, 4);
x = (double)a(1,1);
x = (double)a(1,2);
x = (double)a(2,1);
x = (double)a(2,2);
```

std::ostream::operator<<(const mwArray &)

Write `mwArray` to output stream. The output has the same format as the output when a variable's name is typed at the MATLAB command prompt. See `ToString()`.

Introduced in R2013b