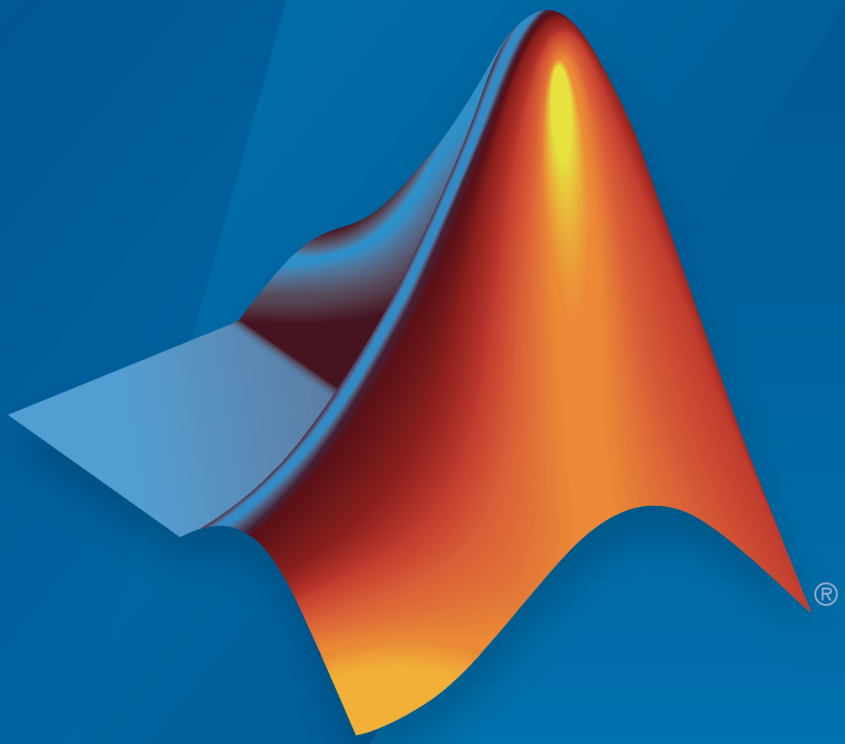
MATLAB® Compiler SDK™

MATLAB® Production Server™ C/C++ Client Guide



MATLAB®

R2015b

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™ MATLAB® Production Server™ C/C++ Client 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)

Contents

1 Client Programming

2 C/C++ Client Development

Client Programming

MATLAB Production Server Examples

Additional Client examples for MATLAB Production Server are available in the `client` folder of your MATLAB Production Server.

Create a C/C++ MATLAB Production Server Client

To create a MATLAB Production Server client:

1. Obtain the client runtime files installed in *`matlabroot`*`/toolbox/compiler_sdk/mps_client/c`.
2. In consultation with the MATLAB programmer, agree on the MATLAB function signatures that comprise the services in the application.
3. Configure your system with the appropriate software for working with C/C++.
4. Initialize the MATLAB Production Server client runtime using `mpsInitializeEx()`.

   ```
   mpsClientRuntime* mpsruntime = mpsInitializeEx(MPS_CLIENT_1_1);
   ```
5. Create the client runtime configuration.

   See “Configure the Client-Server Connection” on page 2-3.
6. Create the client runtime context.

   See “Create the Client Runtime Context” on page 2-2.
7. Create the required MATLAB data for the inputs and outputs of the function.

   See “Data Handling” on page 2-9.
8. Evaluate the MATLAB functions.
9. Handle any errors.

   See “Handle Function Processing Errors ” on page 2-15.
10. Free system resources by cleaning up all MATLAB data and terminating the client connection.

    See “Clean Up MATLAB Resources” on page 2-19.

For a complete example of an implementing a C++ client, see “Create a C++ Client”.

Create a C++ Client

This example shows how to write a MATLAB Production Server client using the C client API. The client application calls the `addmatrix` function you compiled in "Create a Deployable Archive for MATLAB Production Server" and deployed in "Share a Deployable Archive on the Server Instance".

Create a C++ MATLAB Production Server client application:

1. Create a file called `addmatrix_client.cpp`.
2. Using a text editor, open `addmatrix_client.cpp`.
3. Add the following include statements to the file:

```
#include <iostream>
#include <mps/client.h>
```

Note: The header files for the MATLAB Production Server C client API are located in the *`matlabroot`*`/toolbox/compiler_sdk/mps_client/c/include/mps` folder. folder.

---

4. Add the `main()` method to the application.

```
int main ( void )
{
}
```

5. Initialize the client runtime.

```
mpsClientRuntime* mpsruntime = mpsInitializeEx(MPS_CLIENT_1_1);
```

6. Create the client configuration.

```
mpsClientConfig* config;
mpsStatus status = mpsruntime->createConfig(&config);
```

7. Create the client context.

```
mpsClientContext* context;
status = mpsruntime->createContext(&context, config);
```

8. Create the MATLAB data to input to the function.

```
double a1[2][3] = {{1,2,3},{3,2,1}};
double a2[2][3] = {{4,5,6},{6,5,4}};

int numIn=2;
mpsArray** inVal = new mpsArray* [numIn];
```

```
inVal[0] = mpsCreateDoubleMatrix(2,3,mpsREAL);
inVal[1] = mpsCreateDoubleMatrix(2,3,mpsREAL);

double* data1 = (double *)( mpsGetData(inVal[0]) );
double* data2 = (double *)( mpsGetData(inVal[1]) );

for(int i=0; i<2; i++)
{
  for(int j=0; j<3; j++)
  {
    mpsIndex subs[] = { i, j };
    mpsIndex id = mpsCalcSingleSubscript(inVal[0], 2, subs);
    data1[id] = a1[i][j];
    data2[id] = a2[i][j];
  }
}
```

9 Create the MATLAB data to hold the output.

```
int numOut = 1;
mpsArray **outVal = new mpsArray* [numOut];
```

10 Call the deployed MATLAB function.

Specify the following as arguments:

- client context
- URL of the function
- Number of expected outputs
- Pointer to the `mpsArray` holding the outputs
- Number of inputs
- Pointer to the `mpsArray` holding the inputs

```
mpsStatus status = mpsruntime->feval(context,
    "http://localhost:9910/addmatrix/addmatrix",
    numOut, outVal, numIn, (const mpsArray**)inVal);
```

For more information about the `feval` function, see the reference material included in the *`matlabroot`*`/toolbox/compiler_sdk/mps_client` folder.

11 Verify that the function call was successful using an `if` statement.

```
if (status==MPS_OK)
{
```

```
}
```

12 Inside the `if` statement, add code to process the output.

```
double* out = mpsGetPr(outVal[0]);

for (int i=0; i<2; i++)
{
  for (int j=0; j<3; j++)
  {
    mpsIndex subs[] = {i, j};
    mpsIndex id = mpsCalcSingleSubscript(outVal[0], 2, subs);
    std::cout << out[id] << "\t";
  }
  std::cout << std::endl;
}
```

13 Add an `else` clause to the `if` statement to process any errors.

```
else
{
  mpsErrorInfo error;
  mpsruntime->getLastErrorInfo(context, &error);
  std::cout << "Error: " << error.message << std::endl;
  switch(error.type)
  {
    case MPS_HTTP_ERROR_INFO:
      std::cout << "HTTP: " << error.details.http.responseCode << ": "
          << error.details.http.responseMessage << std::endl;
    case MPS_MATLAB_ERROR_INFO:
      std::cout << "MATLAB: " << error.details.matlab.identifier
          << std::endl;
      std::cout << error.details.matlab.message << std::endl;
    case MPS_GENERIC_ERROR_INFO:
      std::cout << "Generic: " << error.details.general.genericErrorMsg
          << std::endl;
  }

  mpsruntime->destroyLastErrorInfo(&error);
}
```

14 Free the memory used by the inputs.

```
for (int i=0; i<numIn; i++)
  mpsDestroyArray(inVal[i]);
delete[] inVal;
```

15 Free the memory used by the outputs.

```
for (int i=0; i<numOut; i++)
  mpsDestroyArray(outVal[i]);
delete[] outVal;
```

16 Free the memory used by the client runtime.

```
mpsruntime->destroyConfig(config);
mpsruntime->destroyContext(context);
mpsTerminate();
```

17 Save the file.

The completed program should resemble the following:

```
#include <iostream>
#include <mps/client.h>

int main ( void )
{
  mpsClientRuntime* mpsruntime = mpsInitializeEx(MPS_CLIENT_1_1);

  mpsClientConfig* config;
  mpsStatus status = mpsruntime->createConfig(&config);

  mpsClientContext* context;
  status = mpsruntime->createContext(&context, config);

  double a1[2][3] = {{1,2,3},{3,2,1}};
  double a2[2][3] = {{4,5,6},{6,5,4}};

  int numIn=2;
  mpsArray** inVal = new mpsArray* [numIn];
  inVal[0] = mpsCreateDoubleMatrix(2,3,mpsREAL);
  inVal[1] = mpsCreateDoubleMatrix(2,3,mpsREAL);
  double* data1 = (double *)( mpsGetData(inVal[0]) );
  double* data2 = (double *)( mpsGetData(inVal[1]) );
  for(int i=0; i<2; i++)
  {
    for(int j=0; j<3; j++)
    {
      mpsIndex subs[] = { i, j };
      mpsIndex id = mpsCalcSingleSubscript(inVal[0], 2, subs);
      data1[id] = a1[i][j];
      data2[id] = a2[i][j];
    }
  }

  int numOut = 1;
  mpsArray **outVal = new mpsArray* [numOut];

  status = mpsruntime->feval(context,
              "http://localhost:9910/addmatrix/addmatrix",
              numOut, outVal, numIn, (const mpsArray **)inVal);

  if (status==MPS_OK)
  {
    double* out = mpsGetPr(outVal[0]);

    for (int i=0; i<2; i++)
```

```
    {
      for (int j=0; j<3; j++)
      {
        mpsIndex subs[] = {i, j};
        mpsIndex id = mpsCalcSingleSubscript(outVal[0], 2, subs);
        std::cout << out[id] << "\t";
      }
      std::cout << std::endl;
    }
  }
  else
  {
    mpsErrorInfo error;
    mpsruntime->getLastErrorInfo(context, &error);
    std::cout << "Error: " << error.message << std::endl;

    switch(error.type)
    {
    case MPS_HTTP_ERROR_INFO:
      std::cout << "HTTP: "
          << error.details.http.responseCode
          << ": " << error.details.http.responseMessage
          << std::endl;
    case MPS_MATLAB_ERROR_INFO:
      std::cout << "MATLAB: " << error.details.matlab.identifier
          << std::endl;
      std::cout << error.details.matlab.message << std::endl;
    case MPS_GENERIC_ERROR_INFO:
      std::cout << "Generic: "
          << error.details.general.genericErrorMsg
          << std::endl;
    }
    mpsruntime->destroyLastErrorInfo(&error);
  }

  for (int i=0; i<numIn; i++)
    mpsDestroyArray(inVal[i]);
  delete[] inVal;

  for (int i=0; i<numOut; i++)
    mpsDestroyArray(outVal[i]);
  delete[] outVal;

  mpsruntime->destroyConfig(config);
  mpsruntime->destroyContext(context);
  mpsTerminate();
}
```

18 Compile the application.

To compile your client code, the compiler needs access to `client.h`. This header file is stored in *`matlabroot`*`/toolbox/compiler_sdk/mps_client/c/include/mps/`.

To link your application, the linker needs access to the following files stored in :

Files Required for Linking

Windows®	UNIX®/Linux	Mac OS X
`$arch\lib\mpsclient.lib`	`$arch/lib/libprotobuf.so`	`$arch/lib/libprotobuf.dylib`
	`$arch/lib/libcurl.so`	`$arch/lib/libcurl.dylib`
	`$arch/lib/libmwmpsclient.so`	`$arch/lib/libmwmpsclient.dylib`
	`$arch/lib/libmwcpp11compat.so`	

19 Run the application.

To run your application, add the following files stored in to the application's path:

Files Required for Running

Windows	UNIX/Linux	Mac OS X
`$arch\lib\mpsclient.dll`	`$arch/lib/libprotobuf.so`	`$arch/lib/libprotobuf.dylib`
`$arch\lib\libprotobuf.dll`	`$arch/lib/libcurl.so`	`$arch/lib/libcurl.dylib`
`$arch\lib\libcurl.dll`	`$arch/lib/libmwmpsclient.so`	`$arch/lib/libmwmpsclient.dylib`
	`$arch/lib/libmwcpp11compat.so`	

The client invokes `addmatrix` function on the server instance and returns the following matrix at the console:

```
5.0 7.0 9.0
9.0 7.0 5.0
```

Unsupported MATLAB Data Types for Client and Server Marshaling

These data types are not supported for marshaling between MATLAB Production Server instances and clients:

- MATLAB function handles

C/C++ Client Development

Create the Client Runtime Context

The client runtime context encapsulates the information required by the client-server connection.

You create the context using the `mpsClientRuntime` `createContext()` function. The `createContext()` function takes a pointer to a unitialized `mpsClientContext` variable and an initialized client configuration.

```
mpsClientRuntime* mpsruntime = mpsInitializeEx(MPS_CLIENT_1_1);
mpsClientConfig* config;
mpsStatus status = mpsruntime->createConfig(&config);
mpsClientContext* context;
status = mpsruntime->createContext(&context, config);
```

For information on creating a client configuration, see “Configure the Client-Server Connection” on page 2-3.

Note: Do not share an instance of `mpsClientContext` across multiple threads at the same time. In a multi-threaded environment, each thread should get its own instance of `mpsClientContext`.

Configure the Client-Server Connection

In this section...

You configure the client-server connection using a structure of type `mpsClientConfig`. The structure has fields to configure:

- amount of time, in milliseconds, the client waits for a response before timing out.
- maximum size, in bytes, of the response a client accepts.
- security parameters.

You can use methods provided by the `mpsClientConfig` structure, to change the values before you create the client context.

Create a Connection with the Default Configuration

When you create the client configuration using the runtime API `createConfig()` function, it is populated with default values:

- `responseTimeOut` = `120000`
- `reponseSizeLimit` = `64*1024*1024` (64 MB)

```
mpsClientConfig* config;
mpsStatus status = mpsruntime->createConfig(&config);
```

Change the Response Time Out

To change the amount of time the client waits for a response use the `setTimeOutSec()` function provided by the `mpsClientRuntime` structure.

This code sample creates a client connection with a time out value of 1000 ms:

```
mpsClientConfig* config;
mpsStatus status = mpsruntime->createConfig(&config);
mpsruntime->setResponseTimeOutSec(config, 1000);
```

Tip Setting the response time out to `0` specifies that the client will wait indefinitely for a response.

Change the Response Size Limit

To change the amount of data a client will accept in a response use the `setResponseSizeLimit()` function provided by the `mpsClientConfig` structure.

This code sample creates a client connection that accepts a maximum of 4 MB in a response:

```
mpsClientConfig* config;
mpsStatus status = mpsruntime->createConfig(&config);
config->setResponseSizeLimit(4*1024*1024);
```

Access Secure Programs Using HTTPS

In this section...

It is possible to connect to a secure server instance by simply using an HTTPS address when calling `feval()`.

The resulting connection will be encrypted, but not secure. Neither party performs any authentication. Neither can determine if it communicating with a valid actor or a malignant actor.

To establish a secure connection you must:

- install valid certificate authorities for server instance authentication
- configure your client application code to use the installed certificate authorities to authenticate the server instance

To ensure an added level of security, you can also verify the server instance host name against the certificate common name.

These steps allow your client to ensure that it is communicating with a valid MATLAB Production Server instance.

In some environments, server instances will also require client authentication. In these environments, you will need to

Configure the Client’s Environment for SSL

At a minimum the client requires the server's root CA (Certificate Authority) in one of the application's certificate stores.

To connect to a server that requires client-side authentication, the client needs a signed certificate in one of the application's certificate stores.

To manage the certificate authorities and certificates on the client machine, use OpenSSL.

Make a Secure Request

To configure your client to authenticate server instances you need to add the following to the client runtime configuration:

- server root CA
- private key

  If the private key is encrypted, you also need to provide the private key password.
- that the client authenticates the server instance

In addition to the minimum requirements you can also specify:

- certificate revocation list to check against
- if the client needs to verify the server instance hostname against the certificates common name

You do this using setters on the `mpsClientRuntime` structure:

- specifies the client certificate
- `setPrivateKeyFile(mpsClientConfig* sslCfg, const char* pkFile)` specifies the private key
- `setPrivateKeyPasswd(mpsClientConfig* sslCfg, const char* passwd)` specifies the private key password
- `setCAFile(mpsClientConfig* sslCfg, const char* caFile)` specifies the certificate authority
- `setRevocationListFile(mpsClientConfig* sslCfg, const char* crlFile)` specifies the certificate revocation list
- `setVerifyHost(mpsClientConfig* sslCfg, mpsLogical verifyHost)` specifies if the client verifies the server instance hostname
- `setVerifyPeer(mpsClientConfig* sslCfg, mpsLogical verifyPeer)` specifies if the client authenticates the server instance

The following code configures the client to fully authenticate the server instance. It also configures the client to verify that the server instance hostname matches the certificate common name.

```
mpsClientRuntime* mpsruntime = mpsInitializeEx(MPS_CLIENT_1_1);
```

```
mpsClientConfig* config;
mpsStatus status1 = mpsruntime->createConfig(&config);

const std::string caFile("CERT_AUTH_FILE");
mpsruntime->setCAFile(config, caFile.c_str());

const std::string crlFile("CERT_REVOCATION_LIST_FILE");
mpsruntime->setRevocationListFile(config, crlFile.c_str());

mpsruntime->setVerifyHost(config, static_cast<mpsLogical>(true));
mpsruntime->setVerifyPeer(config, static_cast<mpsLogical>(true));

mpsClientContext* context;
status = mpsruntime->createContext(&context, config);

...
status = mpsruntime->feval(context,
              "https://localhost:9911/addmatrix/addmatrix",
              numOut, outVal, numIn, (const mpsArray **)inVal);
```

When the client attempts to evaluate the function, it will exchange certificates with the server instance. The client will verify the server instance certificate against the configured CA. If the certificate is valid, the client will then verify that the server instance hostname matches the common name stored in the server instance certificate. If either check fails, the connection is rejected.

If the server instance is configured to perform client authentication, the connection will also be rejected since the client is not configured with a valid certificate to exchange with the server instance.

Make a Secure Request Using Client Authentication

In some environments, server instances require that clients provide a certificate for authentication. To enable the client to connect with a server instance requiring client authentication:

- set the client cert file property using the `setClientCertFile()` setter of the `mpsClientRuntime` structure.
- Set the private key properties to access the client certificate.

```
const std::string certFile("CERTFILE");
mpsruntime->setClientCertFile(config, certFile.c_str());
```

```
const std::string pkFile("PRIVATE_KEY_FILE");
mpsruntime->setPrivateKeyFile(config, pkFile.c_str());

const std::string pkPass("PRIVATE_KEY_PASSWORD");
mpsruntime->setPrivateKeyPasswd(config, pkPass.c_str());
```

External Websites

- OpenSSL

Data Handling

In this section...

The MATLAB Array

The MATLAB Runtime works with a single object type: the MATLAB array. All MATLAB variables (including scalars, vectors, matrices, strings, cell arrays, structures, and objects) are stored as MATLAB arrays. In the MATLAB Production Server C/C++ client API, the MATLAB array is declared to be of type `mpsArray`. The `mpsArray` structure contains the following information about the array:

- Type
- Dimensions
- Data associated with the array
- If numeric, whether the variable is real or complex
- If sparse, its indices and nonzero maximum elements
- If a structure or object, the number of fields and field names

To access the `mpsArray` structure, use the `mpsArray` API functions. These functions enable you to create, read, and query information about the MATLAB data used by the client.

Note: The `mpsArray` API mirrors the `mxArray` API used by MATLAB Compiler SDK™ and MATLAB external interfaces.

Data Storage

MATLAB stores data in a column-major (columnwise) numbering scheme. MATLAB internally stores data elements from the first column first, then data elements from the second column second, and so on, through the last column.

For example, given the matrix:

```
a=['house'; 'floor'; 'porch']
a =
   house
   floor
   porch
```

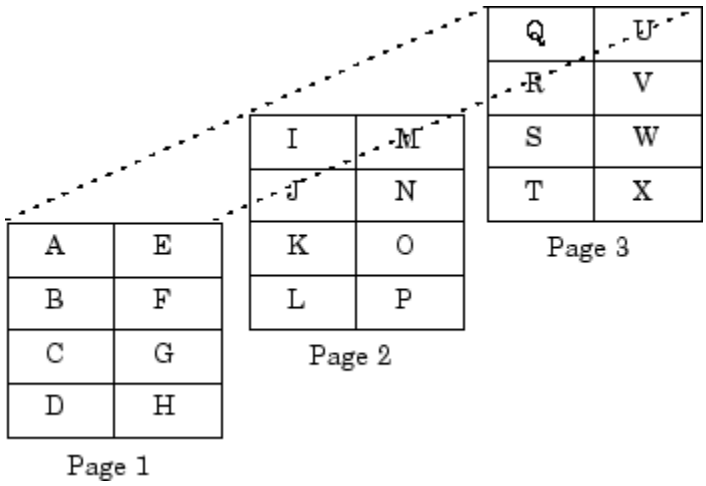
its dimensions are:

```
size(a)
ans =
     3     5
```

and its data is stored as:

h	f	p	o	l	o	u	o	r	s	o	c	e	r	h

If a matrix is N-dimensional, MATLAB represents the data in N-major order. For example, consider a three-dimensional array having dimensions 4-by-2-by-3. Although you can visualize the data as:



MATLAB internally represents the data for this three-dimensional array in the following order:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

The `mpsCalcSingleSubscript()` function creates the offset from the first element of an array to the desired element, using N-dimensional subscripting.

Note: MATLAB indexing starts at 1 where C indexing starts at 0.

MATLAB Types

Complex Double-Precision Matrices

Complex double-precision, non-sparse matrices are of type double and have dimensions m-by-n, where `m` is the number of rows and `n` is the number of columns. The data is stored as two vectors of double-precision numbers—one contains the real data and one contains the imaginary data. The pointers to this data are referred to as `pr` (pointer to real data) and `pi` (pointer to imaginary data), respectively. A non-complex matrix is one whose `pi` is `NULL`.

Numeric Matrices

Numeric matrices are single-precision floating-point integers that can be 8-, 16-, 32, and 64-bit, both signed and unsigned. The data is stored in two vectors in the same manner as double-precision matrices.

Logical Matrices

The `logical` data type represents a logical `true` or `false` state using the numbers `1` and `0`, respectively. Certain MATLAB functions and operators return logical `1` or logical

0 to indicate whether a certain condition was found to be true or not. For example, the statement (5 * 10) > 40 returns a logical 1 value.

MATLAB Strings

MATLAB strings are of type char and are stored in a similar manner as unsigned 16-bit integers, except there is no imaginary data component. Unlike C, MATLAB strings are not null terminated.

Cell Arrays

Cell arrays are a collection of MATLAB arrays where each mpsArray is referred to as a cell, enabling MATLAB arrays of different types to be stored together. Cell arrays are stored in a similar manner to numeric matrices, except the data portion contains a single vector of pointers to mpsArrays. Members of this vector are called cells. Each cell can be of any supported data type, even another cell array.

Structures

Structures are MATLAB arrays with elements accessed by textual field designators.

Following is an example of how structures are created in MATLAB:

```
S.name = 'Ed Plum';
S.score = 83;
S.grade = 'B+'
```

creates a scalar structure with three fields:

```
S =
    name: 'Ed Plum'
    score: 83
    grade: 'B+'
```

A 1-by-1 structure is stored in the same manner as a 1-by-n cell array where n is the number of fields in the structure. Members of the data vector are called fields. Each field is associated with a name stored in the mpsArray.

Multidimensional Arrays

A multidimensional array is a vector of integers where each element is the size of the corresponding dimension. The storage of the data is the same as matrices. MATLAB arrays of any type can be multidimensional.

Empty Arrays

MATLAB arrays of any type can be empty. An empty `mpsArray` is one with at least one dimension equal to zero. For example, a double-precision `mpsArray` of type `double`, where `m` and `n` equal `0` and `pr` is `NULL`, is an empty array.

Sparse Matrices

Sparse matrices have a different storage convention from that of full matrices in MATLAB. The parameters `pr` and `pi` are still arrays of double-precision numbers, but these arrays contain only nonzero data elements. There are three additional parameters:

- `nzmax` is an integer that contains the length of `ir`, `pr`, and, if it exists, `pi`. It is the maximum number of nonzero elements in the sparse matrix.
- `ir` points to an integer array of length `nzmax` containing the row indices of the corresponding elements in `pr` and `pi`.
- `jc` points to an integer array of length `n+1`, where n is the number of columns in the sparse matrix. The `jc` array contains column index information. If the `j`th column of the sparse matrix has any nonzero elements, `jc[j]` is the index in `ir` and `pr` (and `pi` if it exists) of the first nonzero element in the `j`th column, and `jc[j+1] - 1` is the index of the last nonzero element in that column. For the `j`th column of the sparse matrix, `jc[j]` is the total number of nonzero elements in all preceding columns. The last element of the `jc` array, `jc[n]`, is equal to `nnz`, the number of nonzero elements in the entire sparse matrix. If `nnz` is less than `nzmax`, more nonzero entries can be inserted into the array without allocating more storage.

Using Data Types

- "Declaring Data Structures" on page 2-14
- "Manipulating Data" on page 2-14

You can write MATLAB Production Server client applications in C/C++ that accept any class or data type supported by MATLAB (see "MATLAB Types" on page 2-11).

Caution The MATLAB Runtime does not check the validity of MATLAB data structures created in C/C++. Using invalid syntax to create a MATLAB data structure can result in unexpected behavior.

Declaring Data Structures

To handle MATLAB arrays, use type `mpsArray`. The following statement declares an `mpsArray` named `myData`:

```
mpsArray *myData;
```

To define the values of `myData`, use one of the `mpsCreate*` functions. Some useful array creation routines are `mpsCreateNumericArray()`, `mpsCreateCellArray()`, and `mpsCreateCharArray()`. For example, the following statement allocates an `m`-by-1 floating-point `mpsArray` initialized to `0`:

```
myData = mpsCreateDoubleMatrix(m, 1, mpsREAL);
```

C/C++ programmers should note that data in a MATLAB array is in column-major order. (For an illustration, see "Data Storage" on page 2-9.) Use the `mpsGet*` array access routines to read data from an `mpsArray`.

Manipulating Data

The `mpsGet*` array access routines get references to the data in an `mpsArray`. Use these routines to modify data in your client application. Each function provides access to specific information in the `mpsArray`. Some useful functions are `mpsGetData()`, `mpsGetPr()`, `mpsGetM()`, and `mpsGetString()`. The following statements read the input string `prhs[0]` into a C-style string `buf`:

```
char *buf;
int buflen;
int status;
buflen = mpsGetN(prhs[0])*sizeof(mpsChar)+1;
buf = malloc(buflen);
status = mpsGetString(prhs[0], buf, buflen);
```

Handle Function Processing Errors

In this section...

To handle errors that occur when processing MATLAB functions:

1. Evaluate the status returned by the `feval()` function to determine if the function was successfully processed.
2. Get the error information using the `getLastErrorInfo()` function.
3. Interrogate the `type` field of the error detail to determine the type of error.
4. Process the error information appropriately.
5. Clean-up the resources used by the error information using the `destroyLastErrorInfo()` function.

Determine if an Error Occurred

The `feval()` function returns a value of type `mpsStatus`, which signifies if an error occurred while the function was being processed. The status can have one of two values:

- `MPS_OK` indicates that the function processed successfully.
- `MPS_FAILURE` indicates that an error occurred.

For example, to check if an error occurred while evaluating a MATLAB function, use an if-then statement.

```
status = mpsruntime->feval(context,funUrl,outArgs,outVal,inArgs,inVal);
if (status==MPS_OK)
{
  ...
}
else
```

```
{
  ...
}
```

Get the Error Information

If a call to the `feval()` function returns a value of `MPS_FAILURE`, you can get the details of the error by calling the `getLastErrorInfo()` function. It returns an `mpsErrorInfo` structure that contains these fields:

- `message` — String containing general information about the error
- `type` — Character identifying the type of error. The type identifier is used to select the correct data type for the detailed error information.
- `details` — Structure containing details, such as the MATLAB stack, about the error and its underlying cause

To get the error information and print the basic error message:

```
mpsErrorInfo error;
mpsruntime->getLastErrorInfo(context, &error);
std::cout << "Error: " << error.message << std::endl;
```

Determine the Type of Error

Before you can process the detailed error information, you need to determine what type of error occurred. This is done by interrogating the `type` field of the `mpsErrorInfo` structure. It can have one of three values:

- `MPS_HTTP_ERROR_INFO` — Non-200 HTTP error occurred and the details are stored in an `mpsErrorInfoHTTP` structure
- `MPS_MATLAB_ERROR_INFO` — MATLAB error occurred and the details are stored in an `mpsErrorInfoMATLAB` structure
- `MPS_GENERIC_ERROR_INFO` — Indeterminate error occurred and the details are stored in an `mpsErrorInfoGeneric` structure

Once you determine the type of error, you can process the detailed information. To determine the error type using a `switch` statement:

```
mpsErrorInfo error;
mpsruntime->getLastErrorInfo(context, &error);
```

```
switch(error.type)
{
case MPS_HTTP_ERROR_INFO:
  ...
case MPS_MATLAB_ERROR_INFO:
  ...
case MPS_MATLAB_ERROR_INFO:
  ...
}
```

Process HTTP Errors

The details of an HTTP errors are stored in an `mpsErrorInfoHTTP` structure. This structure has two fields:

- `responseCode` — HTTP error code
- `responseMessage` — String containing the message returned with the error

For example, if you attempt to access a function using an invalid URL, the client may return an `mpsErrorInfoHTTP` structure with the following values:

- `responseCode` — 404
- `responseMessage` — Not Found

Process MATLAB Errors

If the error occurs while the MATLAB Runtime is evaluating the function the client returns an `mpsErrorInfoMATLAB` structure. This structure has the following fields:

- `message` — Error message returned by the MATLAB Runtime
- `identifier` — MATLAB error ID
- `matlabStack` — MATLAB Runtime stack
- `matlabStackDepth` — Number of entries in the MATLAB Runtime stack

The entries in the MATLAB Runtime stack have the following fields:

- `file` — Name of the MATLAB file that caused the error
- `function` — Name of the MATLAB function that caused the error
- `line` — Line number in the MATLAB file that caused the error

To print the contents of a MATLAB error:

```
mpsErrorInfo error;
mpsruntime->getLastErrorInfo(context, &error);
switch(error.type)
{
case MPS_HTTP_ERROR_INFO:
  ...
case MPS_MATLAB_ERROR_INFO:
  std::cout << "MATLAB: " << error.details.matlab.identifier
    << std::endl;
  std::cout << error.details.matlab.message << std::endl;
  for (int i=0; i < error.details.matlab.matlabStackDepth; i++)
  {
    std::cout << "in " <<  error.details.matlab.matlabStack[i].file
     << " at " << error.details.matlab.matlabStack[i].function
     << " line number " << error.details.matlab.matlabStack[i].line
     << std::endl;
  }
case MPS_MATLAB_ERROR_INFO:
  ...
}
```

Process Generic Errors

If an error other than a non-200 HTTP response or a MATLAB Runtime exception occurs, the client returns an `mpsErrorInfoGeneric` structure containing a `genericErrorMessage` field.

Clean Up Error Information

The error information created by the MATLAB Production Server client runtime is opaque. Once you have processed the error, clean up the resources used by the error using the `mpsClientRuntime` `destroyLastErrorInfo()` function. It takes a pointer to the error information returned from `getLastErrorInfo()`.

```
mpsClientRuntime* mpsruntime = mpsInitialize();
mpsErrorInfo error;
mpsruntime->getLastErrorInfo(context, &error);
...
mpsruntime->destroyLastErrorInfo(&error);
```

Clean Up MATLAB Resources

In this section...

Clean Up Client Configuration

You can clean up the client configuration any time after it is used to create the client context. The context copies the required configuration values when it is created.

To clean up the client configuration, use the `mpsClientRuntime` `destroyConfig()` function with a pointer to the client configuration data.

```
mpsClientRuntime* mpsruntime = mpsInitialize();
mpsClientConfig* config;
mpsStatus status = mpsruntime->createConfig(&config);
mpsClientContext* context;
status = mpsruntime->createContext(&context, config);
...
mpsruntime->destroyConfig(config);
```

Clean Up Client Context

The client context encapsulates the connection framework between the client and a server instance. It is required to evaluate MATLAB functions. The context also performs a number of tasks to optimize the connections to server instances.

The client context should not be cleaned up until the client is done evaluating MATLAB functions.

Clean up the client context using the `mpsClientRuntime` `destroyContext()` function with a pointer to the client context data.

```
mpsClientRuntime* mpsruntime = mpsInitialize();
mpsClientConfig* config;
mpsStatus status = mpsruntime->createConfig(&config);
```

```
mpsClientContext* context;
status = mpsruntime->createContext(&context, config);
...
mpsruntime->destroyContext(context);
```

Clean Up Client Runtime

When you are finished using the client API, clean up the runtime resources using the `mpsTerminate()` function.

Note: `mpsTerminate()` does not clean up the client context or the client configuration. They must be cleaned up before calling `mpsTerminate()`.

Clean Up MATLAB Arrays

MATLAB arrays stored in `mpsArray` variables are opaque. They contain a number of fields used to marshal data between your C client code and the MATLAB Runtime. Variables containing MATLAB arrays can be large.

Clean up variables containing MATLAB arrays using the `mpsDestroyArray()` function. The `mpsDestroyArray()` function takes a pointer to the MATLAB array being cleaned up. It frees all of the memory used by the array.

Note: When cleaning up the arrays used as inputs and outputs of an `feval()` call, you must clean up all of the MATLAB arrays contained in the array before you destroy the MATLAB array pointing to the inputs or outputs.

Clean up the data used in an `feval()` call.

```
const mpsArray** const inVal = new const mpsArray* [numIn];
...
mpsArray **outVal = new mpsArray* [numOut];
...
status = mpsruntime->feval(context,funUrl, numOut, outVal,
                           numIn, inVal);
if (status==MPS_OK)
{
  ...
```

```
  for (int i=0; i<numOut; i++)
    mpsDestroyArray(outVal[i]);
  delete[] outVal;
}
for (int i=0; i<numIn; i++)
  mpsDestroyArray(inVal[i]);
delete[] inVal;
```