computer-readable form and you have to type it in. Essentially the same as the first method, this method has the advantage of allowing you to use your editor to change the data and correct mistakes. You can then just rerun your M-file to re-enter the data.

- **Load data from an ASCII flat file.**

  A *flat file* stores the data in ASCII form, with fixed-length rows terminated with new lines (carriage returns) and with spaces separating the numbers. You can edit ASCII flat files using a normal text editor. Flat files can be read directly into MATLAB using the `load` command. The result is to create a variable with the same name as the filename.

  See the `load` function reference page for more information.

- **Read data using MATLAB I/O functions.**

  You can read data using `fopen`, `fread`, and MATLAB other low-level I/O functions. This method is useful for loading data files from other applications that have their own established file formats.

- **Write a MEX-file to read the data.**

  This is the method of choice if subroutines are already available for reading data files from other applications. See the section, “Introducing MEX-Files” on page 3-2, for more information.

- **Write a program to translate your data.**

  You can write a program in C or Fortran to translate your data into MAT-file format. You can then read the MAT-file into MATLAB using the `load` command. Refer to the section, “Reading and Writing MAT-Files” on page 1-5, for more information.

Exporting Data from MATLAB

There are several methods for getting MATLAB data back to the outside world:

- **Create a diary file.**

  For small matrices, use the `diary` command to create a diary file and display the variables, echoing them into this file. You can use your text editor to manipulate the diary file at a later time. The output of `diary` includes the MATLAB commands used during the session, which is useful for inclusion into documents and reports.

Ex.1049 / Page 265 of 748
Intel Corporation et al

- **Use the Save command.**

  Save the data in ASCII form using the `save` command with the `-ascii` option. For example,

  ```
  A = rand(4,3);
  save temp.dat A -ascii
  ```

  creates an ASCII file called `temp.dat` containing

  ```
  1.3889088e-001  2.7218792e-001  4.4509643e-001
  2.0276522e-001  1.9881427e-001  9.3181458e-001
  1.9872174e-001  1.5273927e-002  4.6599434e-001
  6.0379248e-001  7.4678568e-001  4.1864947e-001
  ```

  The `-ascii` option supports two-dimensional double and character arrays only. Multidimensional arrays, cell arrays, and structures are not supported.

  See the `save` function reference page for more information.

- **Use MATLAB I/O functions.**

  Write the data in a special format using `fopen`, `fwrite`, and the other low-level I/O functions. This method is useful for writing data files in the file formats required by other applications.

- **Develop a MEX-file to write the data.**

  You can develop a MEX-file to write the data. This is the method of choice if subroutines are already available for writing data files in the form needed by other applications. See the section, “Introducing MEX-Files” on page 3-2, for more information.

- **Translate data from a MAT-file.**

  You can write out the data as a MAT-file using the `save` command. You can then write a program in C or Fortran to translate the MAT-file into your own special format. See the section, “Reading and Writing MAT-Files” on page 1-5, for more information.

Exchanging Data Files Between Platforms

You may want to work with MATLAB implementations on several different computer systems, or need to transmit MATLAB applications to users on other systems. MATLAB applications consist of M-files containing functions and scripts, and MAT-files containing binary data.

Ex.1049 / Page 266 of 748
Intel Corporation et al

Both types of files can be transported directly between machines: M-files because they are platform independent and MAT-files because they contain a machine signature in the file header. MATLAB checks the signature when it loads a file and, if a signature indicates that a file is foreign, performs the necessary conversion.

Using MATLAB across several different machine architectures requires a facility for exchanging both binary and ASCII data between the various machines. Examples of this type of facility include FTP, NFS, Kermit, and other communication programs. When using these programs, be careful to transmit binary MAT-files in *binary file mode* and ASCII M-files in *ASCII file mode*. Failure to set these modes correctly corrupts the data.

Reading and Writing MAT-Files

The `save` command in MATLAB saves the MATLAB arrays currently in memory to a binary disk file called a MAT-file. The term MAT-file is used because these files have the extension `.mat`. The `load` command performs the reverse operation. It reads the MATLAB arrays from a MAT-file on disk back into MATLAB workspace.

A MAT-file may contain one or more of any of the data types supported in MATLAB 5 or later, including strings, matrices, multidimensional arrays, structures, and cell arrays. MATLAB writes the data sequentially onto disk as a continuous byte stream.

MAT-File Interface Library

The MAT-file interface library contains a set of routines for reading and writing MAT-files. You can call these routines from within your own C and Fortran programs. We recommend that you use these routines, rather than attempt to write your own code, to perform these operations. By using the routines in this library, you will be insulated from future changes to the MAT-file structure.

Ex.1049 / Page 267 of 748
Intel Corporation et al

The MAT-file library contains routines for reading and writing MAT-files. They all begin with the three-letter prefix `mat`. These tables list all the available MAT-functions and their purposes.

Table 1-1: C MAT-File Routines

MAT-Function	Purpose
`matOpen`	Open a MAT-file
`matClose`	Close a MAT-file
`matGetDir`	Get a list of MATLAB arrays from a MAT-file
`matGetFp`	Get an ANSI C file pointer to a MAT-file
`matGetVariable`	Read a MATLAB array from a MAT-file
`matGetVariableInfo`	Load a MATLAB array header from a MAT-file (no data)
`matGetNextVariable`	Read the next MATLAB array from a MAT-file
`matGetNextVariableInfo`	Load the next MATLAB array header from a MAT-file (no data)
`matPutVariable`	Write a MATLAB array to a MAT-file
`matPutVariableAsGlobal`	Put a MATLAB array into a MAT-file such that the `load` command will place it into the global workspace
`matDeleteVariable`	Remove a MATLAB array from a MAT-file

Ex.1049 / Page 268 of 748
Intel Corporation et al

Table 1-2: Fortran MAT-File Routines

MAT-Function	Purpose
`matOpen`	Open a MAT-file
`matClose`	Close a MAT-file
`matGetDir`	Get a list of MATLAB arrays from a MAT-file
`matGetVariable`	Get a named MATLAB array from a MAT-file
`matGetVariableInfo`	Get header for named MATLAB array from a MAT-file
`matGetNextVariable`	Get the next sequential MATLAB array from a MAT-file
`matGetNextVariableInfo`	Get header foro next sequential MATLAB array from a MAT-file
`matPutVariable`	Put a MATLAB array into a MAT-file
`matPutVariableAsGlobal`	Put a MATLAB array into a MAT-file
`matDeleteVariable`	Remove a MATLAB array from a MAT-file

Ex.1049 / Page 269 of 748
Intel Corporation et al

Finding Associated Files

A collection of files associated with reading and writing MAT-files is located on your disk. The following table, MAT-Function Subdirectories, lists the path to the required subdirectories for importing and exporting data using MAT-functions.

Table 1-3: MAT-Function Subdirectories

Platform	Contents	Directories
Windows	Include Files	`<matlab>\extern\include`
	Libraries	`<matlab>\bin\win32`
	Examples	`<matlab>\extern\examples/eng_mat`
UNIX	Include Files	`<matlab>/extern/include`
	Libraries	`<matlab>/extern/lib/$arch`
	Examples	`<matlab>/extern/examples/eng_mat`

Include Files

The `include` directory holds header files containing function declarations with prototypes for the routines that you can access in the API Library. These files are the same for both Windows and UNIX. Included in the subdirectory are:

- `matrix.h`, the header file that defines MATLAB array access and creation methods
- `mat.h`, the header file that defines MAT-file access and creation methods

Libraries

The subdirectory that contains shared (dynamically linkable) libraries for linking your programs is platform dependent.

Shared Libraries on Windows. The `bin` subdirectory contains the shared libraries for linking your programs:

- `libmat.dll`, the library of MAT-file routines (C and Fortran)
- `libmx.dll`, the library of array access and creation routines

Ex.1049 / Page 270 of 748
Intel Corporation et al

Shared Libraries on UNIX. The `extern/lib/$arch` subdirectory, where `$arch` is your machine's architecture, contains the shared libraries for linking your programs. For example, on `sol2`, the subdirectory is `extern/lib/sol2`:

- `libmat.so`, the library of MAT-file routines (C and Fortran)
- `libmx.so`, the library of array access and creation routines

Example Files

The `examples/eng_mat` subdirectory contains C and Fortran source code for a number of example files that demonstrate how to use the MAT-file routines. The source code files are the same for both Windows and UNIX.

Table 1-4: C and Fortran Examples

Library	Description
`matcreat.c`	Example C program that demonstrates how to use the library routines to create a MAT-file that can be loaded into MATLAB
`matdgns.c`	Example C program that demonstrates how to use the library routines to read and diagnose a MAT-file
`matdemo1.f`	Example Fortran program that demonstrates how to call the MATLAB MAT-file functions from a Fortran program
`matdemo2.f`	Example Fortran program that demonstrates how to use the library routines to read in the MAT-file created by `matdemo1.f` and describe its contents

For additional information about the MATLAB API files and directories, see "Additional Information" on page 3-41.

Ex.1049 / Page 271 of 748
Intel Corporation et al

Examples of MAT-Files

This section includes C and Fortran examples of writing, reading, and diagnosing MAT-files. The examples cover the following topics:

- "Creating a MAT-File in C" on page 1-10
- "Reading a MAT-File in C" on page 1-15
- "Creating a MAT-File in Fortran" on page 1-19
- "Reading a MAT-File in Fortran" on page 1-24

Creating a MAT-File in C

This sample program illustrates how to use the library routines to create a MAT-file that can be loaded into MATLAB. The program also demonstrates how to check the return values of MAT-function calls for read or write failures.

```
/*
 * MAT-file creation program
 *
 * See the MATLAB API Guide for compiling information.
 *
 * Calling syntax:
 *
 *   matcreat
 *
 * Create a MAT-file which can be loaded into MATLAB.
 *
 * This program demonstrates the use of the following functions:
 *
 *  matClose
 *  matGetVariable
 *  matOpen
 *  matPutVariable
 *  matPutVariableAsGlobal
 *
 * Copyright 1984-2000 The MathWorks, Inc.
 * $Revision: 1.13 $
 */
#include <stdio.h>
#include <string.h> /* For strcmp() */
```

```
#include <stdlib.h> /* For EXIT_FAILURE, EXIT_SUCCESS */
#include "mat.h"

#define BUFSIZE 256

int main() {
  MATFile *pmat;
  mxArray *pa1, *pa2, *pa3;
  double data[9] = { 1.0, 4.0, 7.0, 2.0, 5.0, 8.0, 3.0, 6.0, 9.0 };
  const char *file = "mattest.mat";
  char str[BUFSIZE];
  int status;

  printf("Creating file %s...\n\n", file);
  pmat = matOpen(file, "w");
  if (pmat == NULL) {
    printf("Error creating file %s\n", file);
   printf("(Do you have write permission in this directory?)\n");
    return(EXIT_FAILURE);
  }

  pa1 = mxCreateDoubleMatrix(3,3,mxREAL);
  if (pa1 == NULL) {
      printf("%s : Out of memory on line %d\n", __FILE__,
              __LINE__);
      printf("Unable to create mxArray.\n");
      return(EXIT_FAILURE);
  }

  pa2 = mxCreateDoubleMatrix(3,3,mxREAL);
  if (pa2 == NULL) {
      printf("%s : Out of memory on line %d\n", __FILE__,
             __LINE__);
      printf("Unable to create mxArray.\n");
      return(EXIT_FAILURE);
  }
  memcpy((void *)(mxGetPr(pa2)), (void *)data, sizeof(data));

  pa3 = mxCreateString("MATLAB: the language of technical
                        computing");
```

```
if (pa3 == NULL) {
    printf("%s :  Out of memory on line %d\n", __FILE__,
            __LINE__);
    printf("Unable to create string mxArray.\n");
    return(EXIT_FAILURE);
}

status = matPutVariable(pmat, "LocalDouble", pa1);
if (status != 0) {
    printf("%s :  Error using matPutVariable on line %d\n",
            __FILE__, __LINE__);
    return(EXIT_FAILURE);
}

status = matPutVariableAsGlobal(pmat, "GlobalDouble", pa2);
if (status != 0) {
    printf("Error using matPutVariableAsGlobal\n");
    return(EXIT_FAILURE);
}

status = matPutVariable(pmat, "LocalString", pa3);
if (status != 0) {
    printf("%s :  Error using matPutVariable on line %d\n",
            __FILE__, __LINE__);
    return(EXIT_FAILURE);
}

/*
 * Ooops! we need to copy data before writing the array.  (Well,
 * ok, this was really intentional.) This demonstrates that
* matPutVariable will overwrite an existing array in a MAT-file.
 */
memcpy((void *)(mxGetPr(pa1)), (void *)data, sizeof(data));
status = matPutVariable(pmat, "LocalDouble", pa1);
if (status != 0) {
    printf("%s :  Error using matPutVariable on line %d\n",
            __FILE__, __LINE__);
    return(EXIT_FAILURE);
}
```

```
/* Clean up. */
mxDestroyArray(pa1);
mxDestroyArray(pa2);
mxDestroyArray(pa3);

if (matClose(pmat) != 0) {
  printf("Error closing file %s\n",file);
  return(EXIT_FAILURE);
}

/* Re-open file and verify its contents with matGetVariable. */
pmat = matOpen(file, "r");
if (pmat == NULL) {
  printf("Error reopening file %s\n", file);
  return(EXIT_FAILURE);
}

/* Read in each array we just wrote. */
pa1 = matGetVariable(pmat, "LocalDouble");
if (pa1 == NULL) {
  printf("Error reading existing matrix LocalDouble\n");
  return(EXIT_FAILURE);
}
if (mxGetNumberOfDimensions(pa1) != 2) {
  printf("Error saving matrix: result does not have two
          dimensions\n");
  return(EXIT_FAILURE);
}

pa2 = matGetVariable(pmat, "GlobalDouble");
if (pa2 == NULL) {
  printf("Error reading existing matrix GlobalDouble\n");
  return(EXIT_FAILURE);
}
if (!(mxIsFromGlobalWS(pa2))) {
  printf("Error saving global matrix: result is not global\n");
  return(EXIT_FAILURE);
}

pa3 = matGetVariable(pmat, "LocalString");
```

```
  if (pa3 == NULL) {
    printf("Error reading existing matrix LocalString\n");
    return(EXIT_FAILURE);
  }

  status = mxGetString(pa3, str, sizeof(str));
  if(status != 0) {
      printf("Not enough space. String is truncated.");
      return(EXIT_FAILURE);
  }
  if (strcmp(str, "MATLAB: the language of technical
             computing")) {
    printf("Error saving string: result has incorrect
           contents\n");
    return(EXIT_FAILURE);
  }

  /* Clean up before exit. */
  mxDestroyArray(pa1);
  mxDestroyArray(pa2);
  mxDestroyArray(pa3);

  if (matClose(pmat) != 0) {
    printf("Error closing file %s\n",file);
    return(EXIT_FAILURE);
  }
  printf("Done\n");
  return(EXIT_SUCCESS);
}
```

To produce an executable version of this example program, compile the file and link it with the appropriate library. Details on how to compile and link MAT-file programs on the various platforms are discussed in the section, “Compiling and Linking MAT-File Programs” on page 1-28.

Once you have compiled and linked your MAT-file program, you can run the stand-alone application you have just produced. This program creates a MAT-file, `mattest.mat`, that can be loaded into MATLAB. To run the application, depending on your platform, either double-click on its icon or enter `matcreat` at the system prompt.

```
matcreat
Creating file mattest.mat...
```

To verify that the MAT-file has been created, at the MATLAB prompt enter

```
whos -file mattest.mat
  Name               Size         Bytes  Class

  GlobalDouble       3x3             72  double array (global)
  LocalDouble        3x3             72  double array
  LocalString        1x43            86  char array

Grand total is 61 elements using 230 bytes
```

Reading a MAT-File in C

This sample program illustrates how to use the library routines to read and diagnose a MAT-file.

```
/*
 * MAT-file diagnose program
 *
 * Calling syntax:
 *
 *   matdgns <matfile.mat>
 *
 * It diagnoses the MAT-file named <matfile.mat>.
 *
 * This program demonstrates the use of the following functions:
 *
 *  matClose
 *  matGetDir
 *  matGetNextVariable
 *  matGetNextVariableInfo
 *  matOpen
 *
 *  Copyright (c) 1984-2000 The MathWorks, Inc.
 *  $Revision: 1.8 $
 */

#include <stdio.h>
```

```
#include <stdlib.h>
#include "mat.h"

int diagnose(const char *file) {
  MATFile *pmat;
  char **dir;
  const char *name;
  int ndir;
  int i;
  mxArray *pa;

  printf("Reading file %s...\n\n", file);

  /* Open file to get directory. */
  pmat = matOpen(file, "r");
  if (pmat == NULL) {
    printf("Error opening file %s\n", file);
    return(1);
  }

  /* Get directory of MAT-file. */
  dir = matGetDir(pmat, &ndir);
  if (dir == NULL) {
    printf("Error reading directory of file %s\n", file);
    return(1);
  } else {
    printf("Directory of %s:\n", file);
    for (i=0; i < ndir; i++)
      printf("%s\n", dir[i]);
  }
  mxFree(dir);

  /* In order to use matGetNextXXX correctly, reopen file to
     read in headers. */
  if (matClose(pmat) != 0) {
    printf("Error closing file %s\n",file);
    return(1);
  }
  pmat = matOpen(file, "r");
  if (pmat == NULL) {
```

```
  printf("Error reopening file %s\n", file);
  return(1);
}

/* Get headers of all variables. */
printf("\nExamining the header for each variable:\n");
for (i=0; i < ndir; i++) {
  pa = matGetNextVariableInfo(pmat, &name);
  if (pa == NULL) {
    printf("Error reading in file %s\n", file);
    return(1);
  }
  /* Diagnose header pa. */
 printf("According to its header, array %s has %d dimensions\n",
         name, mxGetNumberOfDimensions(pa));
  if (mxIsFromGlobalWS(pa))
    printf("  and was a global variable when saved\n");
  else
    printf("  and was a local variable when saved\n");
  mxDestroyArray(pa);
}

/* Reopen file to read in actual arrays. */
if (matClose(pmat) != 0) {
  printf("Error closing file %s\n",file);
  return(1);
}
pmat = matOpen(file, "r");
if (pmat == NULL) {
  printf("Error reopening file %s\n", file);
  return(1);
}

/* Read in each array. */
printf("\nReading in the actual array contents:\n");
for (i=0; i<ndir; i++) {
  pa = matGetNextVariable(pmat, &name);
  if (pa == NULL) {
    printf("Error reading in file %s\n", file);
    return(1);
```

```
    }
    /*
     * Diagnose array pa
     */
    printf("According to its contents, array %s has %d
            dimensions\n", name, mxGetNumberOfDimensions(pa));
    if (mxIsFromGlobalWS(pa))
      printf("  and was a global variable when saved\n");
    else
      printf("  and was a local variable when saved\n");
    mxDestroyArray(pa);
  }

  if (matClose(pmat) != 0) {
    printf("Error closing file %s\n",file);
    return(1);
  }
  printf("Done\n");
  return(0);
}


int main(int argc, char **argv)
{
  int result;

  if (argc > 1)
    result = diagnose(argv[1]);
  else{
    result = 0;
    printf("Usage: matdgns <matfile>");
    printf(" where <matfile> is the name of the MAT-file");
    printf(" to be diagnosed\n");
  }
  return (result==0) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

After compiling and linking this program, you can view its results.

```
matdgns mattest.mat
Reading file mattest.mat...

Directory of mattest.mat:
GlobalDouble
LocalString
LocalDouble

Examining the header for each variable:
According to its header, array GlobalDouble has 2 dimensions
  and was a global variable when saved
According to its header, array LocalString has 2 dimensions
  and was a local variable when saved
According to its header, array LocalDouble has 2 dimensions
  and was a local variable when saved

Reading in the actual array contents:
According to its contents, array GlobalDouble has 2 dimensions
  and was a global variable when saved
According to its contents, array LocalString has 2 dimensions
  and was a local variable when saved
According to its contents, array LocalDouble has 2 dimensions
  and was a local variable when saved
Done
```

Creating a MAT-File in Fortran

This example creates a MAT-file, `matdemo.mat`.

```
C     matdemo1.f
C     This is a simple program that illustrates how to call the
C     MATLAB MAT-file functions from a Fortran program.  This
C     demonstration focuses on writing MAT-files.
C
C     matdemo1 - Create a new MAT-file from scratch.
C
C     Copyright (c) 1984-2000 The MathWorks, Inc.
C     $Revision: 1.9 $
```

```
      program matdemo1
C-----------------------------------------------------------------
C     (pointer) Replace integer by integer*8 on the DEC alpha
C     64-bit platform
C
      integer matOpen, mxCreateDoubleMatrix, mxCreateString
      integer matGetVariable, mxGetPr
      integer mp, pa1, pa2, pa3, pa0
C-----------------------------------------------------------------

C     Other variable declarations here.
      integer status, matClose, mxIsFromGlobalWS
      integer matPutVariable, matPutVariableAsGlobal
      integer matDeleteVariable
      double precision dat(9)
      data dat / 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0 /

C     Open MAT-file for writing.
      write(6,*) 'Creating MAT-file matdemo.mat ...'
      mp = matOpen('matdemo.mat', 'w')
      if (mp .eq. 0) then
         write(6,*) 'Can''t open ''matdemo.mat'' for writing.'
         write(6,*) '(Do you have write permission in this
                      directory?)'
         stop
      end if

C     Create variables.
      pa0 = mxCreateDoubleMatrix(3,3,0)
      call mxCopyReal8ToPtr(dat, mxGetPr(pa0), 9)

      pa1 = mxCreateDoubleMatrix(3,3,0)

      pa2 = mxCreateString('MATLAB: The language of computing')

      pa3 = mxCreateString('MATLAB: The language of computing')

      status = matPutVariableAsGlobal(mp, 'NumericGlobal', pa0)
      if (status .ne. 0) then
```

```
         write(6,*) 'matPutVariableAsGlobal ''Numeric Global''
                     failed'
         stop
      end if
      status = matPutVariable(mp, 'Numeric', pa1)
      if (status .ne. 0) then
         write(6,*) 'matPutVariable ''Numeric'' failed'
         stop
      end if
      status = matPutVariable(mp, 'String', pa2)
      if (status .ne. 0) then
         write(6,*) 'matPutVariable ''String'' failed'
         stop
      end if
      status = matPutVariable(mp, 'String2', pa3)
      if (status .ne. 0) then
         write(6,*) 'matPutVariable ''String2'' failed'
         stop
      end if

C     Whoops! Forgot to copy the data into the first matrix --
C     it is now blank.  Well, ok, this was deliberate.  This
C     demonstrates that matPutVariable will overwrite existing
C     matrices.

      call mxCopyReal8ToPtr(dat, mxGetPr(pa1), 9)
      status = matPutVariable(mp, 'Numeric', pa1)
      if (status .ne. 0) then
         write(6,*) 'matPutVariable ''Numeric'' failed 2nd time'
         stop
      end if

C     We will delete String2 from the MAT-file.

      status = matDeleteVariable(mp, 'String2')
      if (status .ne. 0) then
         write(6,*) 'matDeleteVariable ''String2'' failed'
         stop
      end if
```

```
C     Finally, read back in MAT-file to make sure we know what
C     we put in it.

      status = matClose(mp)
      if (status .ne. 0) then
         write(6,*) 'Error closing MAT-file'
         stop
      end if

      mp = matOpen('matdemo.mat', 'r')
      if (mp .eq. 0) then
         write(6,*) 'Can''t open ''matdemo.mat'' for reading.'
         stop
      end if

      pa0 = matGetVariable(mp, 'NumericGlobal')
      if (mxIsFromGlobalWS(pa0) .eq. 0) then
       write(6,*) 'Invalid non-global matrix written to MAT-file'
         stop
      end if

      pa1 = matGetVariable(mp, 'Numeric')
      if (mxIsNumeric(pa1) .eq. 0) then
         write(6,*) 'Invalid non-numeric matrix written to
                    MAT-file'
         stop
      end if


      pa2 = matGetVariable(mp, 'String')
      if (mxIsString(pa2) .eq. 0) then
       write(6,*) 'Invalid non-string matrix written to MAT-file'
         stop
      end if

      pa3 = matGetVariable(mp, 'String2')
      if (pa3 .ne. 0) then
         write(6,*) 'String2 not deleted from MAT-file'
         stop
      end if
```

```
C     Clean up memory.
      call mxDestroyArray(pa0)
      call mxDestroyArray(pa1)
      call mxDestroyArray(pa2)
      call mxDestroyArray(pa3)

      status = matClose(mp)
      if (status .ne. 0) then
         write(6,*) 'Error closing MAT-file'
         stop
      end if

      write(6,*) 'Done creating MAT-file'
      stop
      end
```

Once you have compiled and linked your MAT-file program, you can run the stand-alone application you have just produced. This program creates a MAT-file, `matdemo.mat`, that can be loaded into MATLAB. To run the application, depending on your platform, either double-click on its icon or enter `matdemo1` at the system prompt.

```
matdemo1
Creating MAT-file matdemo.mat ...
Done creating MAT-file
```

To verify that the MAT-file has been created, at the MATLAB prompt enter

```
whos -file matdemo.mat
Name          Size         Bytes  Class

Numeric       3x3             72  double array
String        1x33            66  char array

Grand total is 42 elements using 138 bytes
```

Note For an example of a Windows stand-alone program (not MAT-file specific), see `engwindemo.c` in the `<matlab>\extern\examples\eng_mat` directory.

Reading a MAT-File in Fortran

This sample program illustrates how to use the library routines to read in the MAT-file created by `matdemo1.f` and describe its contents.

```
C     matdemo2.f
C
C     This is a simple program that illustrates how to call the
C     MATLAB MAT-file functions from a Fortran program.  This
C     demonstration focuses on reading MAT-files. It reads in
C     the MAT-file created by matdemo1.f and describes its
C     contents.
C
C     Copyright (c) 1984-2000 The MathWorks, Inc.
C     $Revision: 1.11 $

      program matdemo2
C-----------------------------------------------------------------
C     (pointer) Replace integer by integer*8 on the DEC Alpha
C     64-bit platform

      integer matOpen, matGetDir, matGetNextVariable
      integer matGetNextVariableInfo
      integer mp, dir, adir(100), pa
C-----------------------------------------------------------------

C     Other variable declarations here.
      integer   mxGetM, mxGetN, matClose
      integer   ndir, i, stat
      character*32 names(100), name

C     Open file and read directory.
      mp = matOpen('matdemo.mat', 'r')
      if (mp .eq. 0) then
```

```
         write(6,*) 'Can''t open ''matdemo.mat''.'
         stop
      end if

      dir = matgetdir(mp, ndir)
      if (dir .eq. 0) then
         write(6,*) 'Can''t read directory.'
         stop
      endif

C     Copy pointer into an array of pointers.
      call mxCopyPtrToPtrArray(dir, adir, ndir)

C     Copy pointer to character string.
      do 20 i=1,ndir
         call mxCopyPtrToCharacter(adir(i), names(i), 32)
   20 continue

      write(6,*) 'Directory of Mat-file:'
      do 30 i=1,ndir
         write(6,*) names(i)
   30 continue

      stat = matClose(mp)
      if (stat .ne. 0) then
         write(6,*) 'Error closing ''matdemo.mat''.'
         stop
      end if

C     Reopen file and read full arrays.
      mp = matOpen('matdemo.mat', 'r')
      if (mp .eq. 0) then
         write(6,*) 'Can''t open ''matdemo.mat''.'
         stop
      end if

C     Get Information on first array in mat file.
      write(6,*) 'Getting Header info from first array.'
      pa = matGetVariableInfo(mp, names(1))
      write(6,*) 'Retrieved ', names(1)
```


Ex.1049 / Page 287 of 748
Intel Corporation et al

```
      write(6,*) '  With size ', mxGetM(pa), '-by-', mxGetN(pa)
      call mxDestroyArray(pa)

      write(6,*) 'Getting Header info from next array.'
      pa = matGetNextVariableInfo(mp, name)
      write(6,*) 'Retrieved ', name
      write(6,*) '  With size ', mxGetM(pa), '-by-', mxGetN(pa)
      call mxDestroyArray(pa)

C     Read directory.
      write(6,*) 'Getting rest of array contents:'
      pa = matGetNextVariable(mp, name)

C     Copy name to character string.
      do while (pa .ne. 0)
         i=mxGetM(pa)
         write(*, *) i
         write(6,*) 'Retrieved ', name
        write(6,*) '  With size ', mxGetM(pa), '-by-', mxGetN(pa)
         call mxDestroyArray(pa)
         pa = matGetNextVariable(mp, name)
      end do

      stat = matClose(mp)
      if (stat .ne. 0) then
         write(6,*) 'Error closing ''matdemo.mat''.'
         stop
      end if
      stop
      end
```

After compiling and linking this program, you can view its results.

```
matdemo2
Directory of Mat-file:
String
Numeric
Getting full array contents:
  1
Retrieved String
  With size   1-by-  33
  3
Retrieved Numeric
  With size   3-by-  3
```


Ex.1049 / Page 289 of 748
Intel Corporation et al

Compiling and Linking MAT-File Programs

This section describes the steps required to compile and link MAT-file programs on UNIX and Windows systems. It begins by looking at a special consideration for compilers that do not mask floating-point exceptions. Topics covered are

- "Masking Floating Point Exceptions" on page 1-28
- "Compiling and Linking on UNIX" on page 1-29
- "Compiling and Linking on Windows" on page 1-30

Masking Floating Point Exceptions

Certain mathematical operations can result in nonfinite values. For example, division by zero results in the nonfinite IEEE value, `inf`. A floating-point exception occurs when such an operation is performed. Because MATLAB uses an IEEE model that supports nonfinite values such as `inf` and `NaN`, MATLAB disables, or *masks*, floating-point exceptions.

Some compilers do not mask floating-point exceptions by default. This causes MAT-file applications built with such compilers to terminate when a floating-point exception occurs. Consequently, you need to take special precautions when using these compilers to mask floating-point exceptions so that your MAT-file application will perform properly.

Note MATLAB based applications should never get floating-point exceptions. If you do get a floating-point exception, verify that any third party libraries that you link against do not enable floating-point exception handling.

The only compiler and platform on which you need to mask floating-point exceptions is the Borland C++ compiler on Windows.

Borland C++ Compiler on Windows

To mask floating-point exceptions when using the Borland C++ compiler on the Windows platform, you must add some code to your program. Include the following at the beginning of your `main()` or `WinMain()` function, before any calls to MATLAB API functions.

Ex.1049 / Page 290 of 748
Intel Corporation et al

```
#include <float.h>
    .
    .
    .
_control87(MCW_EM,MCW_EM);
    .
    .
    .
```

Compiling and Linking on UNIX

Under UNIX at runtime, you must tell the system where the API shared libraries reside. These sections provide the necessary UNIX commands depending on your shell and system architecture.

Setting Runtime Library Path

In C shell, the command to set the library path is

```
setenv LD_LIBRARY_PATH <matlab>/extern/lib/$Arch:$LD_LIBRARY_PATH
```

In Bourne shell, the commands to set the library path are

```
LD_LIBRARY_PATH=<matlab>/extern/lib/$Arch:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

where `<matlab>` is the MATLAB root directory and `$Arch` is your system architecture (`alpha`, `glnx86`, `sgi`, `sol2`, `hp700`, or `ibm_rs`).

Note that the environment variable (LD_LIBRARY_PATH in this example) varies on several platforms. The following table lists the different environment variable names you should use on these systems.

Table 1-5: Environment Variable Names

Architecture	Environment Variable
HP700	SHLIB_PATH
IBM RS/6000	LIBPATH

It is convenient to place these commands in a startup script such as `~/.cshrc` for C shell or `~/.profile` for Bourne shell.

Ex.1049 / Page 291 of 748
Intel Corporation et al

Using the Options File

MATLAB provides an options file, `matopts.sh`, that lets you use the `mex` script to easily compile and link MAT-file applications. For example, to compile and link the `matcreat.c` example, you can use

```
mex -f <matlab>/bin/matopts.sh <pathname>/matcreat.c
```

where `<pathname>` specifies the complete path to the specified file.

If you need to modify the options file for your particular compiler or platform, use the `-v` switch to view the current compiler and linker settings and then make the appropriate changes in a local copy of the `matopts.sh` file.

Compiling and Linking on Windows

To compile and link Fortran or C MAT-file programs, use the `mex` script with a MAT options file. Table 3-2, Options Files, on page 3-16 lists the MAT options files included in this release. All of these options files are located in `<matlab>\bin\win32\mexopts`.

As an example, to compile and link the stand-alone MAT application `matcreat.c` on Windows using MSVC (Version 5.0), use

```
mex -f <matlab>\bin\win32\mexopts\msvc50engmatopts.bat <pathname>\matcreat.c
```

where `<pathname>` specifies the complete path to the specified file. If you need to modify the options file for your particular compiler, use the `-v` switch to view the current compiler and linker settings and then make the appropriate changes in a local copy of the options file.

Ex.1049 / Page 292 of 748
Intel Corporation et al

2

MATLAB Interface to Generic DLLs

A shared library is a collection of functions that are available for use by one or more applications running on a system. On Windows systems, the library is precompiled into a dynamic link library (`.dll`) file. At run-time, the library is loaded into memory and made accessible to all applications. The MATLAB Interface to Generic DLLs enables you to interact with functions in dynamic link libraries directly from MATLAB.

This chapter covers the following topics.

Overview (p. 2-2)	Provides a brief introduction to the interface.
Loading and Unloading the Library (p. 2-3)	Describes functions to use in loading the library into MATLAB memory and later releasing that memory.
Getting Information About the Library (p. 2-4)	Shows several ways of obtaining information about the functions contained in a library.
Invoking Library Functions (p. 2-7)	Tells you how to make a call to any function in the library.
Passing Arguments (p. 2-8)	Explains how to construct MATLAB arguments that are compatible with the argument types found in the library functions.
Data Conversion (p. 2-11)	Describes how to convert MATLAB data to C data types when you need to do the conversion manually.

Overview

C programs built into external, shared libraries are easily accessed by MATLAB through a command line interface. This interface gives you the ability to load an external library into MATLAB memory space and then access any of the functions defined therein. Although data types differ between the two language environments, in most cases you can pass MATLAB types to the C functions without having to do the work of conversion. MATLAB does this for you.

This interface also supports libraries containing functions programmed in languages other than C, provided that the functions have a C interface.

Ex.1049 / Page 294 of 748
Intel Corporation et al

Loading and Unloading the Library

To give MATLAB access to external functions in a shared library, you must first load the library into memory. Once loaded, you can request information about any of the functions in the library and call them directly from MATLAB. When the library is no longer needed, you will need to unload it from memory to conserve memory usage.

Loading the Library

To load a shared library into MATLAB, use the `loadlibrary` function. The syntax for `loadlibrary` is

```
loadlibrary('shrlib', 'hfile')
```

where `shrlib` is the filename for the `.dll` shared library file, and `hfile` is the filename for the header file that contains the function prototypes. See the reference page for `loadlibrary` for variations in the syntax that you can use.

Note The header file provides signatures for the functions in the library and is a required argument for `loadlibrary`.

As an example, you can use `loadlibrary` to load the `libmx` library that defines the MATLAB `mx` routines. The first statement below forms the directory specification for the `matrix.h` header file for the `mx` routines. The second loads the library from `libmx.dll`, also specifying the header file:

```
hfile = [matlabroot '\extern\include\matrix.h'];
loadlibrary('libmx', hfile)
```

There are also several optional arguments that you can use with `loadlibrary`. See the `loadlibrary` reference page for more information.

Unloading the Library

To unload the library and free up the memory that it occupied, use the `unloadlibrary` function. For example,

```
unloadlibrary libmx
```

Getting Information About the Library

You can use either of two functions to get information on the functions available in a library that you have loaded:

```
libfunctions('libname')

libfunctionsview('libname')
```

The main difference is that `libfunctions` displays the information in the MATLAB Command Window (and you can assign its output to a variable), and `libfunctionsview` displays the information as a graphical display in a new window.

To see what functions are available in the `libmx` library, use `libfunctions`, specifying the library filename as the only argument. Note that you can use the MATLAB command syntax (with no parentheses or quotes required) when specifying no output variables:

```
libfunctions libmx

Methods for class lib.libmx:
mxAddField               mxGetFieldNumber  mxIsLogicalScalarTrue
mxArrayToString          mxGetImagData     mxIsNaN
mxCalcSingleSubscript    mxGetInf          mxIsNumeric
mxCalloc                 mxGetIr           mxIsObject
mxClearScalarDoubleFlag  mxGetJc           mxIsOpaque
mxCreateCellArray       mxGetLogicals    mxIsScalarDoubleFlagSet
       .                       .                 .
       .                       .                 .
```

To list the functions along with their signatures, use the `-full` switch with `libfunctions`. This shows the MATLAB syntax for calling functions written in C. The data types used in the argument lists and return values match MATLAB types, not C types. See the section “Data Conversion” on page 2-11 for more information on these data types.

```
libfunctions libmx -full

Methods for class lib.libmx:
[mxClassID, MATLAB array] mxGetClassID(MATLAB array)
[lib.pointer, MATLAB array] mxGetData(MATLAB array)
[MATLAB array, voidPtr] mxSetData(MATLAB array, voidPtr)
[lib.pointer, MATLAB array] mxGetPr(MATLAB array)
[MATLAB array, doublePtr] mxSetPr(MATLAB array, doublePtr)
uint8 mxIsFinite(double)
uint8 mxIsInf(double)
          .
          .
```

Viewing Functions in a GUI Interface

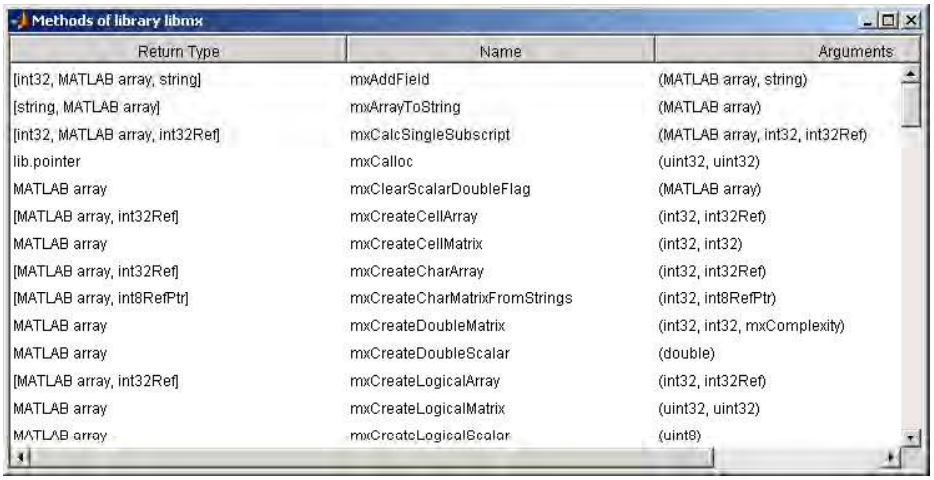
The `libfunctionsview` function creates a new window that displays all of the functions defined in a specific library. For each method, the following information is shown.

Heading	Description
Return Type	Data types that the method returns
Name	Function name
Arguments	Valid data types for input arguments
Inherited From	Not relevant for shared library functions

Ex.1049 / Page 297 of 748
Intel Corporation et al

The following command opens the window shown below for the `libmx` library:

```
libfunctionsview libmx
```



As was true for the `libfunctions` function, the data types displayed here are MATLAB types. See the section “Data Conversion” on page 2-11 for more information on these data types.

Ex.1049 / Page 298 of 748
Intel Corporation et al

Invoking Library Functions

Once a shared library has been loaded into MATLAB, use the `calllib` function to call any of the functions from that library. Specify the library name, function name, and any arguments that get passed to the function:

```
calllib('libname', 'funcname', arg1, ..., argN)
```

This example calls functions from the `libmx` library to test the value stored in `y`.

```
hfile = [matlabroot '\extern\include\matrix.h'];
loadlibrary('libmx', hfile)

y = rand(4, 7, 2);

calllib('libmx', 'mxGetNumberOfElements', y)
ans =
    56

calllib('libmx', 'mxGetClassID', y)
ans =
   mxDOUBLE_CLASS
```


Ex.1049 / Page 299 of 748
Intel Corporation et al

Passing Arguments

To determine which MATLAB data types to use when passing arguments to library functions, see the output of `libfunctionsview` or `libfunctions -full`. These functions list all of the functions found in a particular library along with a specification of the data types required for each argument.

A sample external library called `shrlibsample` is supplied with MATLAB. The `.dll` file for the `shrlibsample` library resides in the directory, `extern\examples\shrlib`. To use the `shrlibsample` library, you first need to either add this directory to your MATLAB path with the command,

```
addpath([matlabroot '\extern\examples\shrlib'])
```

or make this your current working directory with the command,

```
cd([matlabroot '\extern\examples\shrlib'])
```

The following example loads the `shrlibsample` library and displays some of the functions that come with the library:

```
loadlibrary shrlibsample shrlibsample.h
libfunctions shrlibsample -full

doublePtr multDoubleArray(doublePtr, int32)
double addMixedTypes(int16, int32, double)
[double, doublePtr] addDoubleRef(double, doublePtr, double)
[string, string] stringToUpper(string)
string readEnum(Enum1)
double addStructFields(c_struct)
[lib.pointer, doublePtr] multDoubleRef(doublePtr)
[double, c_structPtr] addStructByRef(c_structPtr)
c_structPtrPtr allocateStruct(c_structPtrPtr)
voidPtr deallocateStruct(voidPtr)
int16Ptr multiplyShort(int16Ptr, int32)
```

These functions are all written in C. What you see here is the MATLAB syntax for calling the C functions.

Ex.1049 / Page 300 of 748
Intel Corporation et al

Some General Rules

There are a few interesting things to note about the input and output arguments shown in the function listing above:

- Many of the arguments (like `int32`, `double`) are very similar to their C counterparts. In these cases, you only need to pass in the MATLAB data types shown for these arguments.
- Some arguments in C (like `**double`, or predefined structures), are quite different from standard MATLAB data types. In these cases, you usually have the option of either passing a standard MATLAB type and letting MATLAB convert it for you, or converting the data yourself using MATLAB functions like `libstruct` and `libpointer`. See the next section on "Data Conversion" on page 2-11.
- C input arguments are often passed by reference. Although MATLAB does not support passing by reference, you can create MATLAB arguments that are compatible with C references. In the listing shown above, these are the arguments with names ending in `Ptr` and `PtrPtr`. See "Creating References" on page 2-21.
- C functions often return data in input arguments passed by reference. MATLAB creates additional output arguments to return these values. Note in the listing shown above that all input arguments ending in `Ptr` or `PtrPtr` are also listed as outputs.

A few general guidelines on passing arguments:

- Nonscalar arguments must be declared as passed by reference in the library functions.
- If the library function uses single subscript indexing to reference a two-dimensional matrix, keep in mind that C programs process matrices row by row while MATLAB processes matrices by column. To get C behavior from the function, transpose the input matrix before calling the function, and then transpose the function output.

Ex.1049 / Page 301 of 748
Intel Corporation et al

- When passing an array having more than two dimensions, the shape of the array may be altered by MATLAB. To ensure that the array retains its shape, store the size of the array before making the call, and then use this same size to reshape the output array to the correct dimensions:

```
vs = size(vin)                  % Store the original dimensions
vs =
     2     5     2

vout = calllib('shrlibsample','multDoubleArray', vin, 20);

size(vout)                      % Dimensions have been altered
ans =
     2    10

vout = reshape(vout, vs);      % Restore the array to 2-by-5-by-2

size(vout)
ans =
     2     5     2
```

- Use an empty array, `[]`, to pass a `NULL` parameter to a library function that supports optional input arguments. This is valid only when the argument is declared as a `Ptr` or `PtrPtr` as shown by `libfunctions` or `libfunctionsview`.

Passing References

Many functions in external libraries use arguments that are passed by reference. To enable you to interact with these functions, MATLAB passes what is called a *pointer object* to these arguments. This should not be confused with "passing by reference" in the typical sense of the term. See "Creating References" on page 2-21 for more information.

Ex.1049 / Page 302 of 748
Intel Corporation et al

Data Conversion

This section contains information on how MATLAB handles conversion of argument data and how to convert data yourself when you decide that would be more efficient.

Under most conditions, data passed to and from external library functions is automatically converted by MATLAB to the data type expected by the external function. However, you may choose, at times, to convert some of your argument data manually. Circumstances under which you might find this advantageous are

- When you pass the same piece of data to a series of library functions, it probably makes more sense to convert it once manually at the start rather than having MATLAB convert it automatically on every call. This saves time on unnecessary copy and conversion operations.
- When you pass large structures, you can save memory by creating MATLAB structures that match the shape of the C structures used in the external function instead of using generic MATLAB structures. The `libstruct` function creates a MATLAB structure modeled from a C structure taken from the library.
- When an argument to an external function uses more than one level of referencing (e.g., `double **`), you will need to pass a reference (constructed by the `libpointer` function) rather than have MATLAB convert the data type automatically.

Primitive Types

All standard scalar C data types are supported by the shared library interface. These are shown in the two tables below along with their equivalent MATLAB types. MATLAB uses the type from the right column for arguments having the C type shown in the left column.

(The second table shows *extended* MATLAB types in the right column. These are instances of the MATLAB `lib.pointer` class rather than standard

MATLAB data types. See “Creating References” on page 2-21 for information on the `lib.pointer` class.)

C Type (on a 32-bit computer)	Equivalent MATLAB Type
`char`, `byte`	`int8`
`unsigned char`, `byte`	`uint8`
`short`	`int16`
`unsigned short`	`uint16`
`int`, `long`	`int32`
`unsigned int`, `unsigned long`	`uint32`
`float`	`single`
`double`	`double`
`char *`	`string (1xn char array)`

C Type (on a 32-bit computer)	Extended MATLAB Type
`integer pointer types (int *)`	`(u)int(`*`size`*`)Ptr`
`float *`	`singlePtr`
`double *`	`doublePtr`
`mxArray *`	`MATLAB array`
`void *`	`voidPtr`
`type` `**`	Same as *`type`*`Ptr` with an added `Ptr` (e.g., `double **` is `doublePtrPtr`)

Converting to Other Primitive Types

For primitive types, MATLAB automatically converts any argument to the data type expected by the external function. This means that you can pass a `double` to a function that expects to receive a `byte` (8-bit integer) and MATLAB does the conversion for you.

For example, the C function shown here takes arguments that are of types `short`, `int`, and `double`:

```
double addMixedTypes(short x, int y, double z)
{
   return (x + y + z);
}
```

You can simply pass all of the arguments as type `double` from MATLAB. MATLAB determines what type of data is expected for each argument and performs the appropriate conversions:

```
calllib('shrlibsample', 'addMixedTypes', 127, 33000, pi)
ans =
  3.3130e+004
```

Converting to a Reference

MATLAB also automatically converts an argument passed by value into an argument passed by reference when the external function prototype defines the argument as a reference. So a MATLAB `double` argument passed to a function that expects `double *` is converted to a `double` reference by MATLAB.

`addDoubleRef` is a C function that takes an argument of type `double *`:

```
double addDoubleRef(double x, double *y, double z)
{
    return (x + *y + z);
}
```

Call the function with three arguments of type `double`, and MATLAB handles the conversion:

```
calllib('shrlibsample', 'addDoubleRef', 1.78, 5.42, 13.3)
ans =
   20.5000
```


Ex.1049 / Page 305 of 748
Intel Corporation et al

Strings

For arguments that require `char *`, you can pass a MATLAB string (i.e., character array).

This C function takes a `char *` input argument:

```
char* stringToUpper(char *input) {
   char *p = input;

   if (p != NULL)
      while (*p!=0)
         *p++ = toupper(*p);
   return input;
}
```

`libfunctions` shows that you can use a MATLAB `string` for this input.

```
libfunctions shrlibsample -full
   [string, string] stringToUpper(string)
```

Create a MATLAB character array, `str`, and pass it as the input argument:

```
str = 'This was a Mixed Case string';
calllib('shrlibsample', 'stringToUpper', str)
ans =
   THIS WAS A MIXED CASE STRING
```

Note Although the input argument that MATLAB passes to `stringToUpper` resembles a reference to type `char`, it is not a true reference data type. That is, it does not contain the address of the MATLAB character array, `str`. So, when the function executes, it returns the correct result but does not modify the value in `str`. If you now examine `str`, you find that its original value is unchanged:

```
str
 str =
   This was a Mixed Case string
```

Ex.1049 / Page 306 of 748
Intel Corporation et al

Enumerated Types

For arguments defined as C enumerated types, you can pass either the enumeration string or its integer equivalent.

The `readEnum` function from the `shrlibsample` library returns the enumeration string that matches the argument passed in. Here is the `Enum1` definition and the `readEnum` function in C:

```
enum Enum1 {en1 = 1, en2, en4 = 4} TEnum1;

char* readEnum(TEnum1 val) {
   switch (val) {
   case 1 :return "You chose en1";
   case 2: return "You chose en2";
   case 4: return "You chose en4";
   default : return "enum not defined";
   }
}
```

In MATLAB, you can express an enumerated type as either the enumeration string or its equivalent numeric value. The `TEnum1` definition above declares enumeration `en4` to be equal to `4`. Call `readEnum` first with a string:

```
calllib('shrlibsample', 'readEnum', 'en4')
ans =
   You chose en4
```

Now call it with the equivalent numeric argument, `4`:

```
calllib('shrlibsample', 'readEnum', 4)
ans =
   You chose en4
```

Structures

For library functions that take structure arguments, you need to pass structures that have field names that are the same as those in the structure definitions in the library. To determine the names and also the data types of structure fields, you can do one of the following:

Ex.1049 / Page 307 of 748
Intel Corporation et al

- Consult the documentation that was provided with the library.
- Look for the structure definition in the header file that you used to load the library into MATLAB.

You can also determine the field names of an externally defined structure from within MATLAB using the following procedure. When you create and initialize the structure, you don't necessarily have to match the data types of numeric fields. MATLAB converts to the correct numeric type for you when you make the call using the `calllib` function. To get the structure field names:

1. Use `libfunctionsview` to display the signatures for all functions in the library you are using. `libfunctionsview` shows the names of the structures used by each function. For example, when you type

   ```
   libfunctionsview shrlibsample
   ```

   MATLAB opens a new window displaying function signatures for the `shrlibsample` library. The line showing the `addStructFields` function reads:

   ```
   double addStructFields (c_struct)
   ```

2. If the function you are interested in takes a structure argument, get the structure type from the `libfunctionsview` display, and invoke the `libstruct` function on that type. `libstruct` returns an object that is modeled on the structure as defined in the library:

   ```
   s = libstruct('c_struct');
   ```

3. Get the names of the structure fields from the object returned by `libstruct`:

   ```
   get(s)
       p1: 0
       p2: 0
       p3: 0
   ```

4. Initialize the fields to the values you want to pass to the library function and make the call using `calllib`:

   ```
   s.p1 = 476;   s.p2 = -299;   s.p3 = 1000;
   calllib('shrlibsample', 'addStructFields', s)
   ```


Ex.1049 / Page 308 of 748
Intel Corporation et al

Specifying Structure Field Names

Here are a few general guidelines that apply to structures passed to external library functions:

- These structures may contain fewer fields than defined in the library structure. MATLAB sets any undefined or uninitialized fields to zero.
- Any structure field name that you use must match a field name in the structure definition. Structure names are case sensitive.
- Structures cannot contain additional fields that are not in the library structure definition.

Passing a MATLAB Structure

As with other data types, when an external function takes a structure argument (such as a C structure), you can pass a MATLAB structure to the function in its place. Structure field names must match field names defined in the library, but data types for numeric fields do not have to match. MATLAB converts each numeric field of the MATLAB structure to the correct data type.

Example of Passing a MATLAB Structure. The sample shared library, `shrlibsample`, defines the following C structure and function:

```
struct c_struct {
    double p1;
    short p2;
    long  p3;
};

double addStructFields(struct c_struct st)
{
    double t = st.p1 + st.p2 + st.p3;
    return t;
}
```

The following code passes a MATLAB structure, `sm`, with three `double` fields to `addStructFields`. MATLAB converts the fields to the `double`, `short`, and `long` data types defined in the C structure, `c_struct`.

```
sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;

calllib('shrlibsample', 'addStructFields', sm)
ans =
        1177
```

Passing a libstruct Object

When you pass a structure to an external function, MATLAB makes sure that the structure being passed matches the library's definition for that structure type. It must contain all the necessary fields defined for that type and each field must be of the expected data type. For any fields that are missing in the structure being passed, MATLAB creates an empty field of that name and initializes its value to zero. For any fields that have a data type that doesn't match the structure definition, MATLAB converts the field to the expected type.

When working with small structures, it is efficient enough to have MATLAB do this work for you. You can pass the original MATLAB structure with `calllib` and let MATLAB handle the conversions automatically. However, when working with repeated calls that pass one or more large structures, it may be to your advantage to convert the structure manually before making any calls to external functions. In this way, you save processing time by converting the structure data only once at the start rather than at each function call. You can also save memory if the fields of the converted structure take up less space than the original MATLAB structure.

Using the libstruct Function. You can convert a MATLAB structure to a C-like structure derived from a specific type definition in the library in one step. Call the `libstruct` function, passing in the name of the structure type from the library, and the original structure from MATLAB. The syntax for `libstruct` is

```
s = libstruct('structtype', mlstruct)
```

The value `s` returned by this function is called a *libstruct object*. Although it is truly a MATLAB object, it handles much like a MATLAB structure. The fields of this new "structure" are derived from the external structure type specified by `structtype` in the syntax above.

Ex.1049 / Page 310 of 748
Intel Corporation et al

For example, to convert a MATLAB structure, `sm`, to a libstruct object, `sc`, that is derived from the `c_struct` structure type, use

```
sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;
sc = libstruct('c_struct', sm);
```

The original structure, `sm`, has fields that are all of type `double`. The object, `sc`, returned from the `libstruct` call has fields that match the `c_struct` structure type. These fields are `double`, `short`, and `long`.

Note You can only use `libstruct` on scalar structures.

Creating an Empty libstruct Object. You can also create an empty libstruct object by calling `libstruct` with only the `structtype` argument. This constructs an object with all the required fields and with each field initialized to zero.

```
s = libstruct('structtype')
```

libstruct Requirements for Structures. when converting a MATLAB structure to a libstruct object, the structure to be converted must adhere to the same guidelines that were documented for MATLAB structures passed directly to external functions. See “Specifying Structure Field Names” on page 2-17.

Using the Structure As an Object

The value returned by `libstruct` is not a true MATLAB structure. It is actually an instance of a class called `lib.c_struct`, as seen by examining the output of `whos`:

```
whos sc
  Name      Size                    Bytes  Class

  sc      1x1                           lib.c_struct

Grand total is 1 element using 0 bytes
```


Ex.1049 / Page 311 of 748
Intel Corporation et al

The fields of this structure are implemented as properties of the `lib.c_struct` class. You can read and modify any of these fields using the MATLAB object-oriented functions, `set` and `get`:

```
sc = libstruct('c_struct');

set(sc, 'p1', 100, 'p2', 150, 'p3', 200);

get(sc)
    p1: 100
    p2: 150
    p3: 200
```

You can also read and modify the fields by simply treating them like any other MATLAB structure fields:

```
sc.p1 = 23;

sc.p1
ans =
   23
```

Example of Passing a libstruct Object

Repeat the `addStructFields` example, this time converting the structure to one of type `c_struct` before calling the function:

```
sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;
sc = libstruct('c_struct', sm);

get(sc)
    p1: 476
    p2: -299
    p3: 1000
```

Now call the function, passing the structure `sc`:

```
calllib('shrlibsample', 'addStructFields', sc)
ans =
    1177
```

Note When passing manually converted structures, the structure passed must be of the same type as that used by the external function. For example, you cannot pass a structure of type `records` to a function that expects type `c_struct`.

Creating References

You can pass most arguments to an external function by value, even when the prototype for that function declares the argument to be a reference. However, you may find times when it is useful to pass a MATLAB argument that is the equivalent of a C reference.

Using the libpointer Function

To construct a reference, use the function `libpointer` with this syntax:

```
p = libpointer('type', 'value')
```

To give an example, create a pointer `pv` to an `int16` value. In the first argument to `libpointer`, enter the `type` of pointer you are creating. The type name is always the data type (`int16`, in this case) suffixed by the letters `Ptr`:

```
v = int16(485);
pv = libpointer('int16Ptr', v);
```

The value returned, `pv`, is actually an instance of a MATLAB class called `lib.pointer`. The `lib.pointer` class has the properties `Value` and `DataType`. You can read and modify these properties with the MATLAB `get` and `set` functions:

```
get(pv)
        Value: 485
    DataType: 'int16Ptr'
```

The `lib.pointer` class also has two methods, `setdatatype` and `reshape`, that are described in the next section, "Obtaining the Function's Return Values" on page 2-23:

```
methods(pv)

Methods for class lib.pointer:
   setdatatype  reshape
```

Creating a Reference to a Primitive Type

Here is a simple example that illustrates how to construct and pass a pointer to type `double`, and how to interpret the output data. The function `multDoubleRef` takes one input that is a reference to a `double` and returns the same.

Here is the C function:

```
double *multDoubleRef(double *x)
{
     *x *= 5;
     return x;
}
```

Construct a reference, `xp`, to input data, `x`, and verify its contents:

```
x = 15;
xp = libpointer('doublePtr', x);

get(xp)
       Value: 15
    DataType: 'doublePtr'
```

Now call the function and check the results:

```
calllib('shrlibsample', 'multDoubleRef', xp);

get(xp, 'Value')
ans =
   75
```

Note It is important to note that reference `xp` is not a true pointer as it would be in a language like C. That is, even though it was constructed as a reference to `x`, it does not contain the address of `x`. So, when the function executes, it modifies the `Value` property of `xp`, but it does not modify the value in `x`. If you now examine `x`, you find that its original value is unchanged:

```
x
x =
    15
```

Obtaining the Function's Return Values. In the last example, the result of the function called from MATLAB could be obtained by examining the modified input reference. But this function also returns data in its output arguments that may be useful.

The MATLAB prototype for this function (returned by `libfunctions -full`), indicates that MATLAB returns two outputs. The first is an object of class `lib.pointer`; the second is the `Value` property of the `doublePtr` input argument:

```
libfunctions shrlibsample -full
   [lib.pointer, doublePtr] multDoubleRef(doublePtr)
```

Run the example once more, but this time check the output values returned:

```
x = 15;
xp = libpointer('doublePtr', x);

[xobj, xval] = calllib('shrlibsample', 'multDoubleRef', xp)
xobj =
   lib.pointer
xval =
    75
```

Like the input reference argument, the first output, `xobj`, is an object of class `lib.pointer`. You can examine this output, but first you need to initialize its data type and size as these factors are undefined when returned by the function. Use the `setdatatype` method defined by class `lib.pointer` to set the data type to `doublePtr` and the size to `1`-by-`1`.

Once initialized, you can examine the `xobj` output:

```
setdatatype(xobj, 'doublePtr', 1, 1)

get(xobj)
ans =
       Value: 75
    DataType: 'doublePtr'
```

The second output, `xval`, is a `double` copied from the `Value` of input `xp`.

Creating a Structure Reference

Creating a reference argument to a structure is not much different than using a reference to a primitive type. The function shown here takes a reference to a structure of type `c_struct` as its only input. It returns an output argument that is the sum of all fields in the structure. It also modifies the fields of the input argument:

```
double addStructByRef(struct c_struct *st)
{
    double t = st->p1 + st->p2 + st->p3;
    st->p1 = 5.5;
    st->p2 = 1234;
    st->p3 = 12345678;
    return t;
}
```

Passing the Structure Itself. Although this function expects to receive a structure reference input, it is easier to pass the structure itself and let MATLAB do the conversion to a reference. This example passes a MATLAB structure, `sm`, to the function `addStructByRef`. MATLAB returns the correct value in the output, `x`, but does not modify the contents of the input, `sm`, since `sm` is not a reference:

```
sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;

x = calllib('shrlibsample', 'addStructByRef', sm)
x =
        1177
```


Ex.1049 / Page 316 of 748
Intel Corporation et al

Passing a Structure Reference. The second part of this example passes the structure by reference. This time, the function receives a pointer to the structure and is then able to modify the structure fields.

```
sp = libpointer('c_struct', sm);
calllib('shrlibsample', 'addStructByRef', sp)
ans =
        1177

get(sp, 'Value')
ans =
    p1: 5.5000
    p2: 1234
    p3: 12345678
```

Reference Pointers

Arguments that have more than one level of referencing (e.g., `uint16 **`) are referred to here as reference pointers. In MATLAB, these argument types are named with the suffix `PtrPtr` (for example, `uint16PtrPtr`). See the output of `libfunctionsview` or `methods -full` for examples of this type.

When calling a function that takes a reference pointer argument, you can use a reference argument instead and MATLAB will convert it to the reference pointer. For example, the external `allocateStruct` function expects a `c_structPtrPtr` argument:

```
libfunctions shrlibsample -full
   c_structPtrPtr allocateStruct(c_structPtrPtr)
```

Here is the C function:

```
void allocateStruct(struct c_struct **val)
{
    *val=(struct c_struct*) malloc(sizeof(struct c_struct));
    (*val)->p1 = 12.4;
    (*val)->p2 = 222;
    (*val)->p3 = 333333;
}
```

Although the prototype says that a `c_structPtrPtr` is required, you can use a `c_structPtr` and let MATLAB do the second level of conversion. Create a reference to an empty structure argument and pass it to `allocateStruct`:

```
sp = libpointer('c_structPtr');
calllib('shrlibsample', 'allocateStruct', sp)

get(sp)
ans =
       Value: [1x1 struct]
    DataType: 'c_structPtr'

get(sp, 'Value')
ans =
    p1: 12.4000
    p2: 222
    p3: 333333
```

When you are done, return the memory that you had allocated:

```
calllib('shrlibsample', 'deallocateStruct', sp)
```

3

Calling C and Fortran Programs from MATLAB

Although MATLAB® is a complete, self-contained environment for programming and manipulating data, it is often useful to interact with data and programs external to the MATLAB environment. MATLAB provides an interface to external programs written in the C and Fortran languages.

Introducing MEX-Files (p. 3-2)	Using MEX-files, mx routines, and mex routines
MATLAB Data (p. 3-5)	Data types you can use in MEX-files
Building MEX-Files (p. 3-10)	Compiling and linking your MEx-file
Custom Building MEX-Files (p. 3-19)	Platform-specific instructions on custom building
Troubleshooting (p. 3-31)	Troubleshooting some of the more common problems you may encounter
Additional Information (p. 3-41)	Files you should know about, example programs, where to get help

Introducing MEX-Files

You can call your own C or Fortran subroutines from MATLAB as if they were built-in functions. MATLAB callable C and Fortran programs are referred to as MEX-files. MEX-files are dynamically linked subroutines that the MATLAB interpreter can automatically load and execute.

MEX-files have several applications:

- Large pre-existing C and Fortran programs can be called from MATLAB without having to be rewritten as M-files.
- Bottleneck computations (usually `for`-loops) that do not run fast enough in MATLAB can be recoded in C or Fortran for efficiency.

MEX-files are not appropriate for all applications. MATLAB is a high-productivity system whose specialty is eliminating time-consuming, low-level programming in compiled languages like Fortran or C. In general, most programming should be done in MATLAB. Don't use the MEX facility unless your application requires it.

Using MEX-Files

MEX-files are subroutines produced from C or Fortran source code. They behave just like M-files and built-in functions. While M-files have a platform-independent extension, `.m`, MATLAB identifies MEX-files by platform-specific extensions. This table lists the platform-specific extensions for MEX-files.

Table 3-1: MEX-File Extensions

Platform	MEX-File Extension
Alpha	`mexaxp`
HP, Version 10.20	`mexhp7`
HP, Version 11.x	`mexhpux`
IBM RS/6000	`mexrs6`
Linux	`mexglx`
SGI, SGI64	`mexsg`

Ex.1049 / Page 320 of 748
Intel Corporation et al

Table 3-1: MEX-File Extensions (Continued)

Platform	MEX-File Extension
Solaris	`mexsol`
Windows	`dll`

You can call MEX-files exactly as you would call any M-function. For example, a MEX-file called `conv2.mex` on your disk in the MATLAB `datafun` toolbox directory performs a 2-D convolution of matrices. `conv2.m` only contains the help text documentation. If you invoke the function `conv2` from inside MATLAB, the interpreter looks through the list of directories on the MATLAB search path. It scans each directory looking for the first occurrence of a file named `conv2` with the corresponding filename extension from the table or `.m`. When it finds one, it loads the file and executes it. MEX-files take precedence over M-files when like-named files exist in the same directory. However, help text documentation is still read from the `.m` file.

The Distinction Between mx and mex Prefixes

Routines in the API that are prefixed with `mx` allow you to create, access, manipulate, and destroy `mxArrays`. Routines prefixed with `mex` perform operations back in the MATLAB environment.

mx Routines

The array access and creation library provides a set of array access and creation routines for manipulating MATLAB arrays. These subroutines, which are fully documented in the online API reference pages, always start with the prefix `mx`. For example, `mxGetPi` retrieves the pointer to the imaginary data inside the array.

Although most of the routines in the array access and creation library let you manipulate the MATLAB array, there are two exceptions — the IEEE routines and memory management routines. For example, `mxGetNaN` returns a `double`, not an `mxArray`.

Ex.1049 / Page 321 of 748
Intel Corporation et al

mex Routines

Routines that begin with the `mex` prefix perform operations back in the MATLAB environment. For example, the `mexEvalString` routine evaluates a string in the MATLAB workspace.

Note `mex` routines are only available in MEX-functions.

Ex.1049 / Page 322 of 748
Intel Corporation et al

MATLAB Data

Before you can program MEX-files, you must understand how MATLAB represents the many data types it supports. This section discusses the following topics:

- "The MATLAB Array"
- "Data Storage"
- "Data Types in MATLAB"
- "Using Data Types"

The MATLAB Array

The MATLAB language works with only 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 C, the MATLAB array is declared to be of type `mxArray`. The `mxArray` structure contains, among other things:

- Its type
- Its dimensions
- The data associated with this 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

Data Storage

All MATLAB data is stored columnwise, which is how Fortran stores matrices. MATLAB uses this convention because it was originally written in Fortran. For example, given the matrix

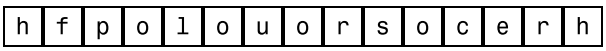
```
a=['house'; 'floor'; 'porch']
a =
   house
   floor
   porch
```


Ex.1049 / Page 323 of 748
Intel Corporation et al

its dimensions are

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

and its data is stored as



Data Types in MATLAB

Complex Double-Precision Matrices

The most common data type in MATLAB is the complex double-precision, nonsparse matrix. These 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 real-only, double-precision matrix is one whose `pi` is `NULL`.

Numeric Matrices

MATLAB also supports other types of numeric matrices. These are single-precision floating-point and 8-, 16-, and 32-bit integers, both signed and unsigned. The data is stored in two vectors in the same manner as double-precision matrices.

MATLAB Strings

MATLAB strings are of type `char` and are stored the same way as unsigned 16-bit integers except there is no imaginary data component. Unlike C, MATLAB strings are not null terminated.

Sparse Matrices

Sparse matrices have a different storage convention than full matrices in MATLAB. The parameters `pr` and `pi` are still arrays of double-precision numbers, but there are three additional parameters, `nzmax`, `ir`, and `jc`:

- `nzmax` is an integer that contains the length of `ir`, `pr`, and, if it exists, `pi`. It is the maximum possible 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` that contains column index information. For `j`, in the range `0 ≤ j ≤ N-1`, `jc[j]` is the index in `ir` and `pr` (and `pi` if it exists) of the first nonzero entry in the `j`th column and `jc[j+1] - 1` index of the last nonzero entry. As a result, `jc[N]` is also equal to `nnz`, the number of nonzero entries in the matrix. If `nnz` is less than `nzmax`, then more nonzero entries can be inserted in the array without allocating additional storage.

Cell Arrays

Cell arrays are a collection of MATLAB arrays where each `mxArray` is referred to as a cell. This allows 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 `mxArrays`. Members of this vector are called cells. Each cell can be of any supported data type, even another cell array.

Structures

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 `mxArray`.

Objects

Objects are stored and accessed the same way as structures. In MATLAB, objects are named structures with registered methods. Outside MATLAB, an object is a structure that contains storage for an additional classname that identifies the name of the object.

Multidimensional Arrays

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

Ex.1049 / Page 325 of 748
Intel Corporation et al

Empty Arrays

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

Using Data Types

The six fundamental data types in MATLAB are `double`, `char`, `logical`, `uint8`, `cell`, and `struct`. You can write MEX-files, MAT-file applications, and engine applications in C that accept any data type supported by MATLAB. In Fortran, only the creation of double-precision n-by-m arrays and strings are supported. You can treat C and Fortran MEX-files, once compiled, exactly like M-functions.

The explore Example

There is an example MEX-file included with MATLAB, called `explore`, that identifies the data type of an input variable. The source file for this example is in the `<matlab>/extern/examples/mex` directory, where `<matlab>` represents the top-level directory where MATLAB is installed on your system.

Note In platform independent discussions that refer to directory paths, this book uses the UNIX convention. For example, a general reference to the `mex` directory is `<matlab>/extern/examples/mex`.

For example, typing

```
cd([matlabroot '/extern/examples/mex']);
x = 2;
explore(x);
```

produces this result

```
------------------------------------------------
Name: x
Dimensions: 1x1
Class Name: double
------------------------------------------------
   (1,1) = 2
```


Ex.1049 / Page 326 of 748
Intel Corporation et al

`explore` accepts any data type. Try using `explore` with these examples.

```
explore([1 2 3 4 5])
explore 1 2 3 4 5
explore({1 2 3 4 5})
explore(int8([1 2 3 4 5]))
explore {1 2 3 4 5}
explore(sparse(eye(5)))
explore(struct('name', 'Joe Jones', 'ext', 7332))
explore(1, 2, 3, 4, 5)
```


Ex.1049 / Page 327 of 748
Intel Corporation et al

Building MEX-Files

This section covers the following topics:

- "Compiler Requirements"
- "Testing Your Configuration on UNIX"
- "Testing Your Configuration on Windows"
- "Specifying an Options File"

Compiler Requirements

Your installed version of MATLAB contains all the tools you need to work with the API. MATLAB includes a C compiler for the PC called Lcc, but does not include a Fortran compiler. If you choose to use your own C compiler, it must be an ANSI C compiler. Also, if you are working on a Microsoft Windows platform, your compiler must be able to create 32-bit windows dynamically linked libraries (DLLs).

MATLAB supports many compilers and provides preconfigured files, called options files, designed specifically for these compilers. The Options Files table lists all supported compilers and their corresponding options files. The purpose of supporting this large collection of compilers is to provide you with the flexibility to use the tool of your choice. However, in many cases, you simply can use the provided Lcc compiler with your C code to produce your applications.

The MathWorks also maintains a list of compilers supported by MATLAB at the following location on the web:
`http://www.mathworks.com/support/tech-notes/v5/1600/1601.shtml`.

Note The MathWorks provides an option (`setup`) for the `mex` script that lets you easily choose or switch your compiler.

The following sections contain configuration information for creating MEX-files on UNIX and Windows systems. More detailed information about the `mex` script is provided in "Custom Building MEX-Files" on page 3-19. In addition, there is a section on "Troubleshooting" on page 3-31, if you are having difficulties creating MEX-files.

Ex.1049 / Page 328 of 748
Intel Corporation et al

Testing Your Configuration on UNIX

The quickest way to check if your system is set up properly to create MEX-files is by trying the actual process. There is C source code for an example, `yprime.c`, and its Fortran counterpart, `yprimef.F` and `yprimefg.F`, included in the `<matlab>/extern/examples/mex` directory, where `<matlab>` represents the top-level directory where MATLAB is installed on your system.

To compile and link the example source files, `yprime.c` or `yprimef.F` and `yprimefg.F`, on UNIX, you must first copy the file(s) to a local directory, and then change directory (`cd`) to that local directory.

At the MATLAB prompt, type

```
mex yprime.c
```

This uses the system compiler to create the MEX-file called `yprime` with the appropriate extension for your system.

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

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

To try the Fortran version of the sample program with your Fortran compiler, at the MATLAB prompt, type

```
mex yprimef.F yprimefg.F
```

In addition to running the `mex` script from the MATLAB prompt, you can also run the script from the system prompt.

Selecting a Compiler

To change your default compiler, you select a different options file. You can do this anytime by using the command

```
mex -setup

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


Ex.1049 / Page 329 of 748
Intel Corporation et al

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

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

The options files available for mex are:

  1: <matlab>/bin/gccopts.sh :
         Template Options file for building gcc MEXfiles

  2: <matlab>/bin/mexopts.sh :
         Template Options file for building MEXfiles using the
          system ANSI compiler

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

Select the proper options file for your system by entering its number and pressing **Return**. If an options file doesn't exist in your MATLAB directory, the system displays a message stating that the options file is being copied to your user-specific `matlab` directory. If an options file already exists in your `matlab` directory, the system prompts you to overwrite it.

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

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

Testing Your Configuration on Windows

Before you can create MEX-files on the Windows platform, you must configure the default options file, `mexopts.bat`, for your compiler. The switch, `setup`, provides an easy way for you to configure the default options file. To configure or change the options file at anytime, run

```
mex -setup
```

from either the MATLAB or DOS command prompt.

Selecting a Compiler

MATLAB includes a C compiler, Lcc, that you can use to create C MEX-files. The `mex` script will use the Lcc compiler automatically if you do not have a C or C++ compiler of your own already installed on your system and you try to compile a C MEX-file. Naturally, if you need to compile Fortran programs, you must supply your own supported Fortran compiler.

The `mex` script uses the filename extension to determine the type of compiler to use for creating your MEX-files. For example,

```
mex test1.f
```

would use your Fortran compiler and

```
mex test2.c
```

would use your C compiler.

On Systems without a Compiler. If you do not have your own C or C++ compiler on your system, the `mex` utility automatically configures itself for the included Lcc compiler. So, to create a C MEX-file on these systems, you can simply enter

```
mex filename.c
```

This simple method of creating MEX-files works for the majority of users.

If using the included Lcc compiler satisfies your needs, you can skip ahead in this section to “Building the MEX-File on Windows” on page 3-15.

On Systems with a Compiler. On systems where there is a C, C++, or Fortran compiler, you can select which compiler you want to use. Once you choose your compiler, that compiler becomes your default compiler and you no longer have

Ex.1049 / Page 331 of 748
Intel Corporation et al

to select one when you compile MEX-files. To select a compiler or change to existing default compiler, use `mex -setup`.

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

```
mex -setup

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

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

Select a compiler:
[1] Compaq Visual Fortran version 6.6
[2] Lcc C version 2.4
[3] Microsoft Visual C/C++ version 6.0

[0] None

Compiler: 3

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

Please verify your choices:

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

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

The default options file:
"C:\WINNT\Profiles\username\ApplicationData\MathWorks\MATLAB\R13
\mexopts.bat" is being updated from ...
```


Ex.1049 / Page 332 of 748
Intel Corporation et al

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

```
The default location for compiler-name is directory-name,
but that directory does not exist on this machine.

Use directory-name anyway [y]/n?
```

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

Building the MEX-File on Windows

There is example C source code, `yprime.c`, and its Fortran counterpart, `yprimef.f` and `yprimefg.f`, included in the `<matlab>\extern\examples\mex` directory, where `<matlab>` represents the top-level directory where MATLAB is installed on your system.

To compile and link the example source file on Windows, at the MATLAB prompt, type

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

This should create the MEX-file called `yprime` with the `.DLL` extension, which corresponds to the Windows platform.

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

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

To try the Fortran version of the sample program with your Fortran compiler, switch to your Fortran compiler using `mex -setup`. Then, at the MATLAB prompt, type

```
cd([matlabroot '\extern\examples\mex'])
mex yprimef.f yprimefg.f
```

In addition to running the `mex` script from the MATLAB prompt, you can also run the script from the system prompt.

Specifying an Options File

You can use the `-f` option to specify an options file on either UNIX or Windows. To use the `-f` option, at the MATLAB prompt type

```
mex filename -f <optionsfile>
```

and specify the name of the options file along with its pathname. The Options Files table, below, contains a list of the options files included with MATLAB.

There are several situations when it may be necessary to specify an options file every time you use the `mex` script. These include:

- *(Windows and UNIX)* You want to use a different compiler (and not use the `-setup` option), or you want to compile MAT or engine stand-alone programs.
- *(UNIX)* You do not want to use the system C compiler.

Preconfigured Options Files

MATLAB includes some preconfigured options files that you can use with particular compilers. The Options Files table lists the compilers whose options files are included with this release of MATLAB.

Table 3-2: Options Files

Platform	Compiler	Options File
Windows	Borland C++, Version 5.0 & 5.2	`bccopts.bat`
	Borland C++Builder 3.0 (Borland C++, Version 5.3)	`bcc53opts.bat`
	Borland C++Builder 4.0 (Borland C++, Version 5.4)	`bcc54opts.bat`
	Borland C++Builder 5.0 (Borland C++, Version 5.5)	`bcc55opts.bat`
	Lcc C Compiler, bundled with MATLAB	`lccopts.bat`
	Microsoft C/C++, Version 5.0	`msvc50opts.bat`
	Microsoft C/C++, Version 6.0	`msvc60opts.bat`

Ex.1049 / Page 334 of 748
Intel Corporation et al

Table 3-2: Options Files (Continued)

Platform	Compiler	Options File
	Watcom C/C++, Version 10.6	`watcopts.bat`
	Watcom C/C++, Version 11	`wat11copts.bat`
	DIGITAL Visual Fortran, Version 5.0	`df50opts.bat`
	Compaq Visual Fortran, Version 6.1	`df61opts.bat`
	Compaq Visual Fortran, Version 6.6	`df66opts.bat`
	Borland C, Version 5.0 & 5.2, for Engine and MAT stand-alone programs	`bccengmatopts.bat`
	Borland C, Version 5.3, for Engine and MAT stand-alone programs	`bcc53engmatopts.bat`
	Borland C, Version 5.4, for Engine and MAT stand-alone programs	`bcc54engmatopts.bat`
	Borland C, Version 5.5, for Engine and MAT stand-alone programs	`bcc55engmatopts.bat`
	Lcc C compiler for Engine and MAT stand-alone programs,	`lccengmatopts.bat`
	Microsoft Visual C for Engine and MAT stand-alone programs, Version 5.0	`msvc50engmatopts.bat`
	Microsoft Visual C for Engine and MAT stand-alone programs, Version 6.0	`msvc60engmatopts.bat`
	Watcom C for Engine and MAT stand-alone programs, Version 10.6	`watengmatopts.bat`

Ex.1049 / Page 335 of 748
Intel Corporation et al

Table 3-2: Options Files (Continued)

Platform	Compiler	Options File
	Watcom C for Engine and MAT stand-alone programs, Version 11	`wat11engmatopts.bat`
	DIGITAL Visual Fortran for MAT stand-alone programs, Version 5.0	`df50engmatopts.bat`
	Compaq Visual Fortran for MAT stand-alone programs, Version 6.1	`df60engmatopts.bat`
UNIX	System ANSI Compiler	`mexopts.sh`
	GCC	`gccopts.sh`
	System ANSI Compiler for Engine stand-alone programs	`engopts.sh`
	System ANSI Compiler for MAT stand-alone programs	`matopts.sh`

An up-to-date list of options files is available from our FTP server, `ftp://ftp.mathworks.com/pub/tech-support/docexamples/apiguide/R12/bin`. For a list of all the compilers supported by MATLAB, access the MathWorks Technical Support Web site at `http://www.mathworks.com/support`.

Note The next section, "Custom Building MEX-Files" on page 3-19, contains specific information on how to modify options files for particular systems.

Ex.1049 / Page 336 of 748
Intel Corporation et al

Custom Building MEX-Files

This section discusses in detail the process that the MEX-file build script uses. It covers the following topics:

- "Who Should Read This Chapter"
- "MEX Script Switches"
- "Default Options File on UNIX"
- "Default Options File on Windows"
- "Custom Building on UNIX"
- "Custom Building on Windows"

Who Should Read This Chapter

In general, the defaults that come with MATLAB should be sufficient for building most MEX-files. There are reasons that you might need more detailed information, such as:

- You want to use an Integrated Development Environment (IDE), rather than the provided script, to build MEX-files.
- You want to create a new options file, for example, to use a compiler that is not directly supported.
- You want to exercise more control over the build process than the script uses.

The script, in general, uses two stages (or three, for Microsoft Windows) to build MEX-files. These are the compile stage and the link stage. In between these two stages, Windows compilers must perform some additional steps to prepare for linking (the prelink stage).

MEX Script Switches

The `mex` script has a set of switches (also called options) that you can use to modify the link and compile stages. The MEX Script Switches table lists the available switches and their uses. Each switch is available on both UNIX and Windows unless otherwise noted.

For customizing the build process, you should modify the options file, which contains the compiler-specific flags corresponding to the general compile, prelink, and link steps required on your system. The options file consists of a

Ex.1049 / Page 337 of 748
Intel Corporation et al

series of variable assignments; each variable represents a different logical piece of the build process.

Table 3-3: MEX Script Switches

Switch	Function
`@<rsp_file>`	Include the contents of the text file `<rsp_file>` as command line arguments to the `mex` script.
`-argcheck`	Perform argument checking on MATLAB API functions (C functions only).
`-c`	Compile only; do not link.
`-D<name>[#<def>]`	Define C preprocessor macro `<name>` [as having value `<def>`]. (Note: UNIX also allows `-D<name>[=<def>]`.)
`-f <file>`	Use `<file>` as the options file; `<file>` is a full pathname if it is not in current directory.
`-g`	Build an executable with debugging symbols included.
`-h[elp]`	Help; lists the switches and their functions.
`-I<pathname>`	Include `<pathname>` in the compiler `include` search path.
`-inline`	Inlines matrix accessor functions (`mx*`). The generated MEX-function may not be compatible with future versions of MATLAB.
`-l<file>`	(UNIX) Link against library `lib<file>`.
`-L<pathname>`	(UNIX) Include `<pathname>` in the list of directories to search for libraries.

Table 3-3: MEX Script Switches (Continued)

Switch	Function
`<name>#<def>`	Override options file setting for variable `<name>`. This option is equivalent to `<ENV_VAR>#<val>`, which temporarily sets the environment variable `<ENV_VAR>` to `<val>` for the duration of the call to `mex`. `<val>` can refer to another environment variable by prepending the name of the variable with a `$`, e.g., `COMPFLAGS#"$COMPFLAGS -myswitch"`.
`<name>=<def>`	(UNIX) Override options file setting for variable `<name>`.
`-O`	Build an optimized executable.
`-outdir <name>`	Place all output files in directory `<name>`.
`-output <name>`	Create an executable named `<name>`. (An appropriate executable extension is automatically appended.)
`-setup`	Set up default options file. This switch should be the only argument passed.
`-U<name>`	Undefine C preprocessor macro `<name>`.
`-v`	Verbose; print all compiler and linker settings.
`-V5`	Compile MATLAB 5-compatible MEX-file.

Default Options File on UNIX

The default MEX options file provided with MATLAB is located in `<matlab>/bin`. The `mex` script searches for an options file called `mexopts.sh` in the following order:

- The current directory
- The directory returned by the `prefdir` function
- The directory specified by `[matlabroot '/bin']`

Ex.1049 / Page 339 of 748
Intel Corporation et al

mex uses the first occurrence of the options file it finds. If no options file is found, mex displays an error message. You can directly specify the name of the options file using the -f switch.

For specific information on the default settings for the MATLAB supported compilers, you can examine the options file in fullfile(matlabroot, 'bin', 'mexopts.sh'), or you can invoke the mex script in verbose mode (-v). Verbose mode will print the exact compiler options, prelink commands (if appropriate), and linker options used in the build process for each compiler. "Custom Building on UNIX" on page 3-23 gives an overview of the high-level build process.

Default Options File on Windows

The default MEX options file is placed in your user profile directory after you configure your system by running mex -setup. The mex script searches for an options file called mexopts.bat in the following order:

- The current directory
- The user profile directory (returned by the prefdir function)
- The directory specified by [matlabroot '\bin\win32\mexopts']

mex uses the first occurrence of the options file it finds. If no options file is found, mex searches your machine for a supported C compiler and automatically configures itself to use that compiler. Also, during the configuration process, it copies the compiler's default options file to the user profile directory. If multiple compilers are found, you are prompted to select one.

For specific information on the default settings for the MATLAB supported compilers, you can examine the options file, mexopts.bat, or you can invoke the mex script in verbose mode (-v). Verbose mode will print the exact compiler options, prelink commands, if appropriate, and linker options used in the build process for each compiler. "Custom Building on Windows" on page 3-25 gives an overview of the high-level build process.

The User Profile Directory

The Windows user profile directory is a directory that contains user-specific information such as desktop appearance, recently used files, and **Start** menu items. The mex and mbuild utilities store their respective options files, mexopts.bat and compopts.bat, which are created during the setup process,

in a subdirectory of your `user profile` directory, named `Application Data\MathWorks\MATLAB`.

Custom Building on UNIX

On UNIX systems, there are two stages in MEX-file building: compiling and linking.

Compile Stage

The compile stage must:

- Add `<matlab>/extern/include` to the list of directories in which to find header files (`-I<matlab>/extern/include`)
- Define the preprocessor macro `MATLAB_MEX_FILE` (`-DMATLAB_MEX_FILE`)
- (C MEX-files only) Compile the source file, which contains version information for the MEX-file, `<matlab>/extern/src/mexversion.c`

Link Stage

The link stage must:

- Instruct the linker to build a shared library
- Link all objects from compiled source files (including `mexversion.c`)
- (Fortran MEX-files only) Link in the precompiled versioning source file, `<matlab>/extern/lib/$Arch/version4.o`
- Export the symbols `mexFunction` and `mexVersion` (these symbols represent functions called by MATLAB)

For Fortran MEX-files, the symbols are all lower case and may have appended underscores. For specific information, invoke the `mex` script in verbose mode and examine the output.

Build Options

For customizing the build process, you should modify the options file. The options file contains the compiler-specific flags corresponding to the general steps outlined above. The options file consists of a series of variable assignments; each variable represents a different logical piece of the build process. The options files provided with MATLAB are located in `<matlab>/bin`.

Ex.1049 / Page 341 of 748
Intel Corporation et al

The section, "Default Options File on UNIX" on page 3-21, describes how the `mex` script looks for an options file.

To aid in providing flexibility, there are two sets of options in the options file that can be turned on and off with switches to the `mex` script. These sets of options correspond to building in *debug mode* and building in *optimization mode*. They are represented by the variables `DEBUGFLAGS` and `OPTIMFLAGS`, respectively, one pair for each *driver* that is invoked (`CDEBUGFLAGS` for the C compiler, `FDEBUGFLAGS` for the Fortran compiler, and `LDDEBUGFLAGS` for the linker; similarly for the `OPTIMFLAGS`).

- If you build in optimization mode (the default), the `mex` script will include the `OPTIMFLAGS` options in the compile and link stages.
- If you build in debug mode, the `mex` script will include the `DEBUGFLAGS` options in the compile and link stages, but will not include the `OPTIMFLAGS` options.
- You can include both sets of options by specifying both the optimization and debugging flags to the `mex` script (`-O` and `-g`, respectively).

Aside from these special variables, the `mex` options file defines the executable invoked for each of the three modes (C compile, Fortran compile, link) and the flags for each stage. You can also provide explicit lists of libraries that must be linked in to all MEX-files containing source files of each language.

The variables can be summed up as follows.

Variable	C Compiler	Fortran Compiler	Linker
Executable	`CC`	`FC`	`LD`
Flags	`CFLAGS`	`FFLAGS`	`LDFLAGS`
Optimization	`COPTIMFLAGS`	`FOPTIMFLAGS`	`LDOPTIMFLAGS`
Debugging	`CDEBUGFLAGS`	`FDEBUGFLAGS`	`LDDEBUGFLAGS`
Additional libraries	`CLIBS`	`FLIBS`	----

Ex.1049 / Page 342 of 748
Intel Corporation et al

For specifics on the default settings for these variables, you can:

- Examine the options file in `<matlab>/bin/mexopts.sh` (or the options file you are using), or
- Invoke the `mex` script in verbose mode

Custom Building on Windows

There are three stages to MEX-file building for both C and Fortran on Windows – compiling, prelinking, and linking.

Compile Stage

For the compile stage, a `mex` options file must:

- Set up paths to the compiler using the `COMPILER` (e.g., Watcom), `PATH`, `INCLUDE`, and `LIB` environment variables. If your compiler always has the environment variables set (e.g., in `AUTOEXEC.BAT`), you can remark them out in the options file.
- Define the name of the compiler, using the `COMPILER` environment variable, if needed.
- Define the compiler switches in the `COMPFLAGS` environment variable.
  a. The switch to create a DLL is required for MEX-files.
  b. For stand-alone programs, the switch to create an `exe` is required.
  c. The `-c` switch (compile only; do not link) is recommended.
  d. The switch to specify 8-byte alignment.
  e. Any other switch specific to the environment can be used.
- Define preprocessor `macro`, with `-D`, `MATLAB_MEX_FILE` is required.
- Set up optimizer switches and/or debug switches using `OPTIMFLAGS` and `DEBUGFLAGS`. These are mutually exclusive: the `OPTIMFLAGS` are the default, and the `DEBUGFLAGS` are used if you set the `-g` switch on the `mex` command line.

Ex.1049 / Page 343 of 748
Intel Corporation et al

Prelink Stage

The prelink stage dynamically creates import libraries to import the required function into the MEX, MAT, or engine file. All MEX-files link against MATLAB only. MAT stand-alone programs link against `libmx.dll` (array access library) and `libmat.dll` (MAT-functions). Engine stand-alone programs link against `libmx.dll` (array access library) and `libeng.dll` for engine functions. MATLAB and each DLL have corresponding `.def` files of the same names located in the `<matlab>\extern\include` directory.

Link Stage

Finally, for the link stage, a `mex` options file must:

- Define the name of the linker in the `LINKER` environment variable.
- Define the `LINKFLAGS` environment variable that must contain:
  - The switch to create a DLL for MEX-files, or the switch to create an `exe` for stand-alone programs.
  - Export of the entry point to the MEX-file as `mexFunction` for C or `MEXFUNCTION@16` for DIGITAL Visual Fortran.
  - The import library(s) created in the `PRELINK_CMDS` stage.
  - Any other link switch specific to the compiler that can be used.
- Define the linking optimization switches and debugging switches in `LINKEROPTIMFLAGS` and `LINKDEBUGFLAGS`. As in the compile stage, these two are mutually exclusive: the default is optimization, and the `-g` switch invokes the debug switches.
- Define the link-file identifier in the `LINK_FILE` environment variable, if needed. For example, Watcom uses `file` to identify that the name following is a file and not a command.
- Define the link-library identifier in the `LINK_LIB` environment variable, if needed. For example, Watcom uses `library` to identify the name following is a library and not a command.
- Optionally, set up an output identifier and name with the output switch in the `NAME_OUTPUT` environment variable. The environment variable `MEX_NAME` contains the name of the first program in the command line. This must be set for `-output` to work. If this environment is not set, the compiler default is to use the name of the first program in the command line. Even if this is set, it can be overridden by specifying the `mex -output` switch.

Linking DLLs to MEX-Files

To link a DLL to a MEX-file, list the DLL's `.lib` file on the command line.

Versioning MEX-Files

The `mex` script can build your MEX-file with a resource file that contains versioning and other essential information. The resource file is called `mexversion.rc` and resides in the `extern\include` directory. To support versioning, there are two new commands in the options files, `RC_COMPILER` and `RC_LINKER`, to provide the resource compiler and linker commands. It is assumed that:

- If a compiler command is given, the compiled resource will be linked into the MEX-file using the standard link command.
- If a linker command is given, the resource file will be linked to the MEX-file after it is built using that command.

Compiling MEX-Files with the Microsoft Visual C++ IDE

Note This section provides information on how to compile MEX-files in the Microsoft Visual C++ (MSVC) IDE; it is not totally inclusive. This section assumes that you know how to use the IDE. If you need more information on using the MSVC IDE, refer to the corresponding Microsoft documentation.

To build MEX-files with the Microsoft Visual C++ integrated development environment:

1. Create a project and insert your MEX source and `mexversion.rc` into it.
2. Create a `.DEF` file to export the MEX entry point. For example

```
LIBRARY MYFILE.DLL
EXPORTS mexFunction           <-- for a C MEX-file
   or
EXPORTS _MEXFUNCTION@16        <-- for a Fortran MEX-file
```

3. Add the `.DEF` file to the project.

Ex.1049 / Page 345 of 748
Intel Corporation et al