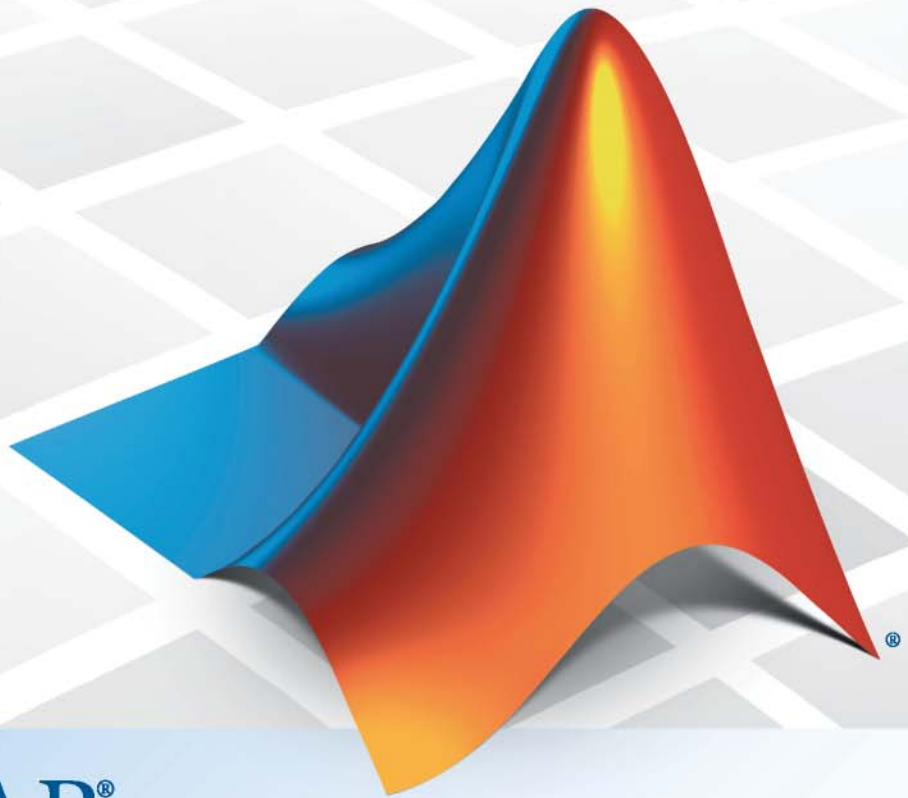


MATLAB® 7

Data Import and Export



MATLAB®

How to Contact MathWorks



www.mathworks.com Web
comp.soft-sys.matlab Newsgroup
www.mathworks.com/contact_TS.html Technical Support



suggest@mathworks.com Product enhancement suggestions
bugs@mathworks.com Bug reports
doc@mathworks.com Documentation error reports
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB® Data Import and Export

© COPYRIGHT 2009–2010 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

September 2009	Online only	New for MATLAB 7.9 (Release 2009b)
March 2010	Online only	Revised for MATLAB 7.10 (Release 2010a)
September 2010	Online only	Revised for MATLAB 7.11 (Release 2010b)

Importing Data

1

Recommended Methods for Importing Data	1-2
Importing Specific File Formats	1-3
Importing Data in Other Formats	1-3
Finding Files	1-4
Processing a Sequence of Files	1-4
Tips for Using the Import Wizard	1-5
Importing MAT-Files	1-14
Viewing the Contents of a MAT-File	1-14
Loading a MAT-File	1-14
Importing Text Data Files	1-18
Requirements for the Import Wizard or importdata	1-18
Importing Numeric ASCII Data	1-19
Importing Numeric ASCII Data with Headers	1-20
Importing Nonnumeric ASCII Data	1-23
Importing Nonrectangular ASCII Data	1-24
Importing Large ASCII Data Sets	1-25
Importing Text Data Files with Low-Level I/O	1-26
Importing XML Documents	1-34
What Is an XML Document Object Model (DOM)?	1-34
Example — Finding Text in an XML File	1-35
Importing Excel Spreadsheets	1-38
Pasting Excel Data from the Clipboard	1-38
Selecting the Spreadsheet and Variables Interactively ...	1-38
Reading a Specific Worksheet and Range of Data	1-39
Reading All Worksheets into a Structure	1-40
Excel Version Support	1-41
Converting Dates	1-42
Importing Scientific Data Files	1-44

Importing Common Data File Format (CDF) Files	1-44
Importing Network Common Data Form (netCDF) Files ..	1-51
Importing Flexible Image Transport System (FITS) Files	1-56
Importing Hierarchical Data Format (HDF5) Files	1-58
Importing Hierarchical Data Format (HDF4) Files	1-73
Importing Images	1-103
Getting Information about Image Files	1-103
Reading Image Data and Metadata from TIFF Files	1-104
Importing Audio and Video	1-107
Getting Information about Audio or Video Files	1-107
Reading Audio Files	1-108
Recording and Playing Audio	1-108
Reading Video Files	1-111
Importing Binary Data with Low-Level I/O	1-116
Reading Binary Data in a File	1-116
Saving Memory	1-118
Reading Portions of a File	1-119
Reading Files Created on Other Systems	1-122
Opening Files with Different Character Encodings	1-123

Exporting Data

2

Exporting to MAT-Files	2-2
What Are MAT-Files?	2-2
Saving Variables from the Workspace	2-2
Updating an Existing MAT-File	2-4
MAT-File Versions	2-5
Exporting to Text Data Files	2-8
Writing to Delimited Data Files	2-8
Writing to a Diary File	2-11
Writing to Text Data Files with Low-Level I/O	2-12

Exporting to XML Documents	2-19
Creating an XML File	2-19
Updating an Existing XML File	2-21
Exporting to Excel Spreadsheets	2-23
Example — Writing to an XLS File	2-23
Adding a New Worksheet	2-24
Converting Dates	2-24
Formatting Cells in Excel Files	2-25
Exporting to Scientific Data Files	2-26
Exporting to Common Data File Format (CDF) Files	2-26
Exporting to Network Common Data Form (netCDF) Files	2-29
Exporting to Hierarchical Data Format (HDF5) Files	2-32
Exporting to Hierarchical Data Format (HDF4) Files	2-43
Exporting to Images	2-54
Exporting Image Data and Metadata to TIFF Files	2-54
Exporting to Audio and Video	2-70
Exporting to Audio Files	2-70
Exporting Video to AVI Files	2-70
Exporting Binary Data with Low-Level I/O	2-72
Writing Binary Data to a File	2-72
Overwriting or Appending to an Existing File	2-73
Creating a File for Use on a Different System	2-75
Opening Files with Different Character Encodings	2-76
Writing and Reading Complex Numbers	2-77
Creating Temporary Files	2-79

Memory-Mapping Data Files

3

Overview of Memory-Mapping	3-2
Benefits of Memory-Mapping	3-2

Limitations of MATLAB Memory-Mapping	3-4
Byte Ordering	3-5
When to Use Memory-Mapping	3-5
The memmapfile Class	3-7
Properties of the memmapfile Class	3-7
Constructing a memmapfile Object	3-9
Constructing the Object with Default Property Values ...	3-10
Changing Property Values	3-10
Selecting the File to Map	3-12
Setting the Start of the Mapped Region	3-13
Identifying the Contents of the Mapped Region	3-14
Mapping of the Example File	3-18
Repeating a Format Scheme	3-20
Setting the Type of Access	3-21
Reading a Mapped File	3-23
Improving Performance	3-23
Example 1 — Reading a Single Data Type	3-24
Example 2 — Formatting File Data as a Matrix	3-25
Example 3 — Reading Multiple Data Types	3-26
Example 4 — Modifying Map Parameters	3-27
Writing to a Mapped File	3-29
Dimensions of the Data Field	3-30
Writing Matrices to a Mapped File	3-31
Selecting Appropriate Data Types	3-34
Working with Copies of the Mapped Data	3-35
Methods of the memmapfile Class	3-37
Using the disp Method	3-37
Using the get Method	3-38
Deleting a Memory Map	3-39
The Effect of Shared Data Copies On Performance	3-39
Memory-Mapping Demo	3-40
The send Function	3-40
The answer Function	3-41
Running the Demo	3-43

4

Downloading Web Content and Files	4-2
Example — Using the <code>urlread</code> Function	4-2
Example — Using the <code>urlwrite</code> Function	4-3
Creating and Decompressing Zip Archives	4-4
Example — Using the <code>zip</code> Function	4-4
Sending E-Mail	4-5
Example — Using the <code>sendmail</code> Function	4-6
Performing FTP File Operations	4-8
Example — Retrieving a File from an FTP Server	4-8

Index

Importing Data


The MATLAB® application can read data in various file formats, discussed in the following sections:

- “Recommended Methods for Importing Data” on page 1-2
- “Importing MAT-Files” on page 1-14
- “Importing Text Data Files” on page 1-18
- “Importing XML Documents” on page 1-34
- “Importing Excel Spreadsheets” on page 1-38
- “Importing Scientific Data Files” on page 1-44
- “Importing Images” on page 1-103
- “Importing Audio and Video” on page 1-107
- “Importing Binary Data with Low-Level I/O” on page 1-116

Recommended Methods for Importing Data

The easiest way to import data into MATLAB from a disk file or the system clipboard is to use the Import Wizard, a graphical user interface. The Import Wizard helps you find a file and define the variables to use in the MATLAB workspace.

To import data from a file, start the Import Wizard, using one of these methods:

- Select **File > Import Data**.
- Double-click a file name in the Current Folder browser.
- Click the Import Data button  in the Workspace browser.
- Call `uiimport`.

To import data from the clipboard, start the Import Wizard, using one of these methods:

- Select **Edit > Paste to Workspace**.
- Call `uiimport`.

To import without invoking a graphical user interface, the easiest option is to use the `importdata` function.

The Import Wizard reads all supported file formats, except netCDF, HDF5, Motion JPEG 2000, and platform-specific video. The `importdata` function reads all formats supported by the Import Wizard, except HDF4.

For more information, see “Tips for Using the Import Wizard” on page 1-5.

Caution When you import data into the MATLAB workspace, the new variables you create overwrite any existing variables in the workspace that have the same name.

Importing Specific File Formats

MATLAB includes functions tailored to import these specific file formats:

- MAT-files
- Text (ASCII) data
- Spreadsheets
- Scientific data
- Images
- Audio and video
- Data in extended markup language (XML)
- Data created using the Data Acquisition Toolbox™ (see `daqread`)

Consider using format-specific functions instead of the Import Wizard or `importdata` when:

- The Import Wizard or `importdata` produces unexpected results.
The Import Wizard makes assumptions about which functions to call, and which input parameters to use. Select and call the format-specific functions directly for more control over the inputs. For example, use `textscan` to import data from a text data file that includes character data (see “Importing Nonnumeric ASCII Data” on page 1-23).
- You want to import only a portion of a file.
Many of the format-specific functions provide options for selecting ranges or portions of data. Alternatively, for binary data files, consider memory-mapping.

For a complete list of the format-specific functions, see the `file formats` table.

Importing Data in Other Formats

If the Import Wizard, `importdata`, and format-specific functions cannot read your data, use *low-level I/O functions* such as `fscanf` or `fread`. Low-level functions allow the most control over reading from a file, but require detailed knowledge of the structure of your data. For more information, see:

- “Importing Text Data Files with Low-Level I/O” on page 1-26
- “Importing Binary Data with Low-Level I/O” on page 1-116

Alternatively, MATLAB toolboxes perform specialized import operations. For example, use Database Toolbox™ software for importing data from relational databases. Refer to the documentation on specific toolboxes to see the available import features.

Finding Files

To find a specific file on the MATLAB search path, use the `which` function. If the file is not in the current folder, include the full or partial path with the file name in calls to import functions.

For example, to locate and load `myfile.mat`:

```
fullname = which('myfile.mat');  
load(fullname);
```

For more information, see:

- “Finding Files and Folders”
- “Path Names in MATLAB”
- “Using the MATLAB Search Path”

Processing a Sequence of Files

To import or export multiple files, create a control loop to process one file at a time. When constructing the loop:

- To build sequential file names, use `sprintf`.
- To find files that match a pattern, use `dir`.
- Use *function syntax* to pass the name of the file to the import or export function. (For more information, see “Command vs. Function Syntax” in the Programming Fundamentals documentation, or the `syntax` reference page.)

For example, to read files named `file1.txt` through `file20.txt` with `importdata`:

```
numfiles = 20;
mydata = cell(1, numfiles);

for k = 1:numfiles
    myfilename = sprintf('file%d.txt', k);
    mydata{k} = importdata(myfilename);
end
```

To read all files that match `*.jpg` with `imread`:

```
jpegFiles = dir('*.jpg');
numfiles = length(jpegFiles);
mydata = cell(1, numfiles);

for k = 1:numfiles
    mydata{k} = imread(jpegFiles(k).name);
end
```

Tips for Using the Import Wizard

Start the Import Wizard by selecting **File > Import Data** or calling `uiimport`.

The Import Wizard provides the following options for reading text files, spreadsheets, images, audio, or video data:

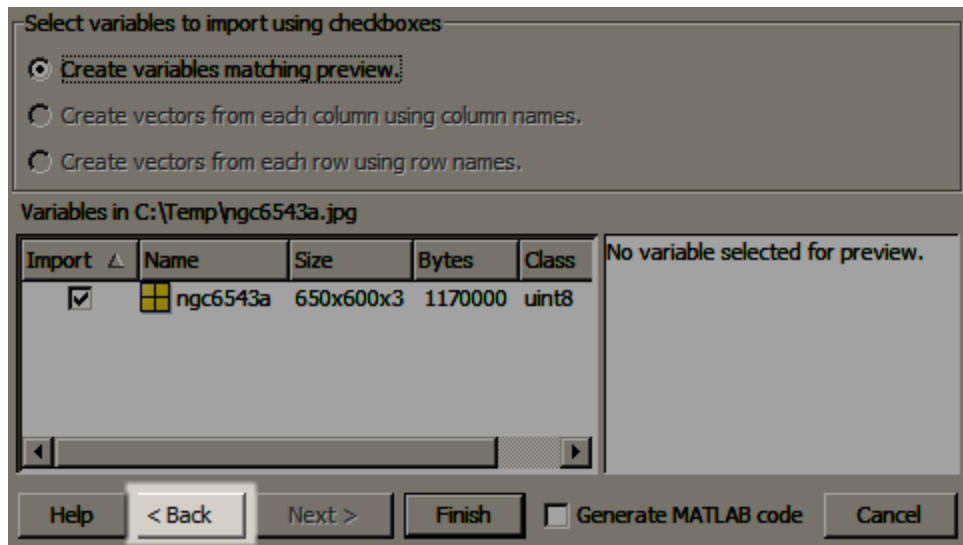
- “Viewing the Contents of a File” on page 1-6
- “Specifying Variables” on page 1-7
- “Generating Reusable MATLAB Code” on page 1-11

Note For information about importing HDF4 files, see “Using the HDF Import Tool” on page 1-74.

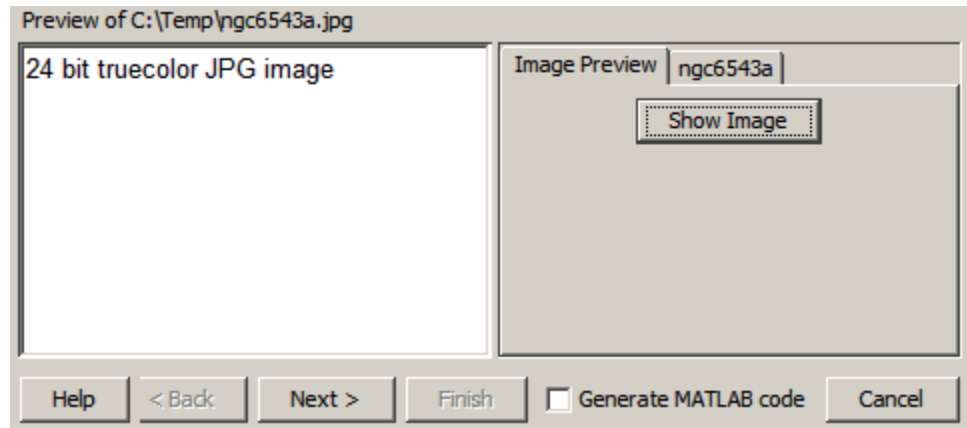
Viewing the Contents of a File

For text files and spreadsheets, the Import Wizard automatically displays a preview of the data in the file.

To view images or video, or to listen to audio, click the **Back** button on the first window that the Import Wizard displays.



The right pane of the new window includes a preview tab. Click the button in the preview tab to show an image or to play audio or video.



Specifying Variables

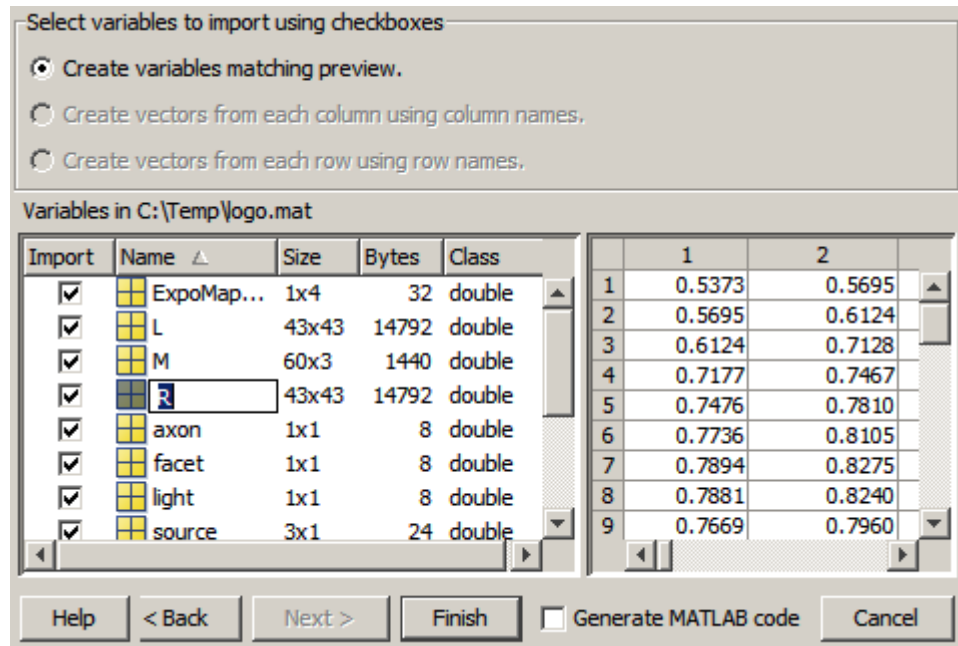
The Import Wizard generates default variable names based on the format and content of your data. You can change the variables in any of the following ways:

- “Renaming or Deselecting Variables” on page 1-7
- “Importing to a Structure Array” on page 1-8
- “Creating Column or Row Vectors from Text Files or Spreadsheets” on page 1-9

The default variable name for data imported from the system clipboard is `A_pastespecial`.

If the Import Wizard detects a single variable in a file, the default variable name is the file name. Otherwise, the Import Wizard uses default variable names that correspond to the output fields of the `importdata` function. For more information on the output fields, see the `importdata` function reference page.

Renaming or Deselecting Variables. To override the default variable, right-click the name and type a new one.



To avoid importing a particular variable, clear the check box in the **Import** column.

Importing to a Structure Array. To import data into fields of a structure array rather than as individual variables, start the Import Wizard by calling `uiimport` with an output argument. For example, the demo file `durer.mat` contains three variables: `X`, `caption`, and `map`. If you issue the command

```
durerStruct = uiimport('durer.mat')
```

and click the **Finish** button, the Import Wizard returns a scalar structure with three fields:

```
durerStruct =
    X: [648x509 double]
   map: [128x3 double]
  caption: [2x28 char]
```


To access a particular field, use dot notation. For example, view the caption field:

```
disp(durerStruct.caption)
```

MATLAB returns:

```
Albrecht Durer's Melancholia.  
Can you find the matrix?
```

For more information, see “Structures” in the MATLAB Programming Fundamentals documentation.

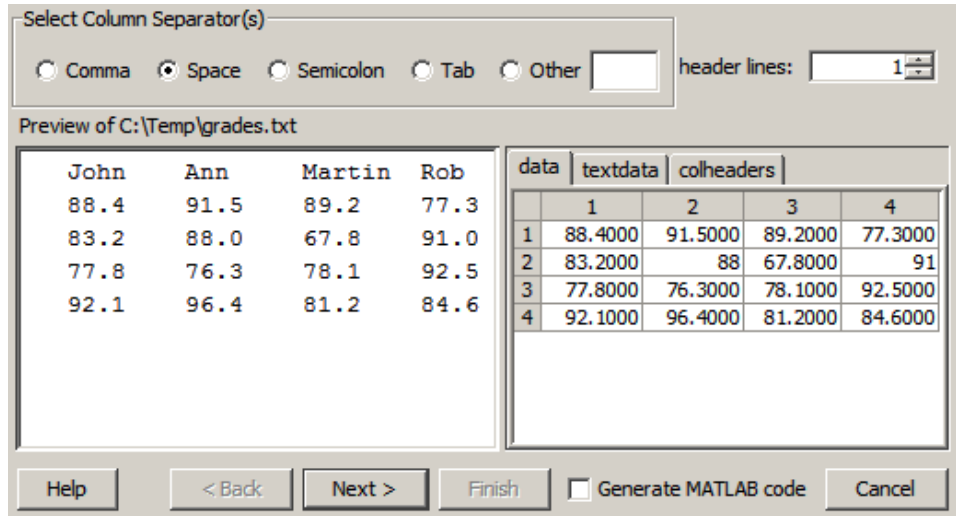
Creating Column or Row Vectors from Text Files or Spreadsheets. If your ASCII file or Excel® worksheet includes either column or row headers, but not both, you can create vectors from the data. The following additional limitations apply:

- If the column headers include more than one line of text, the Import Wizard regards only the lowest line as the column header. However, the default `textdata` variable includes all text from the file.
- If the file contains multiple row headers, or if the file contains text headers, the Import Wizard cannot detect the row headers.
- For text files, the **Number of text header lines** that the Import Wizard lists in the preview window includes column header lines. You cannot specify the number of row headers.

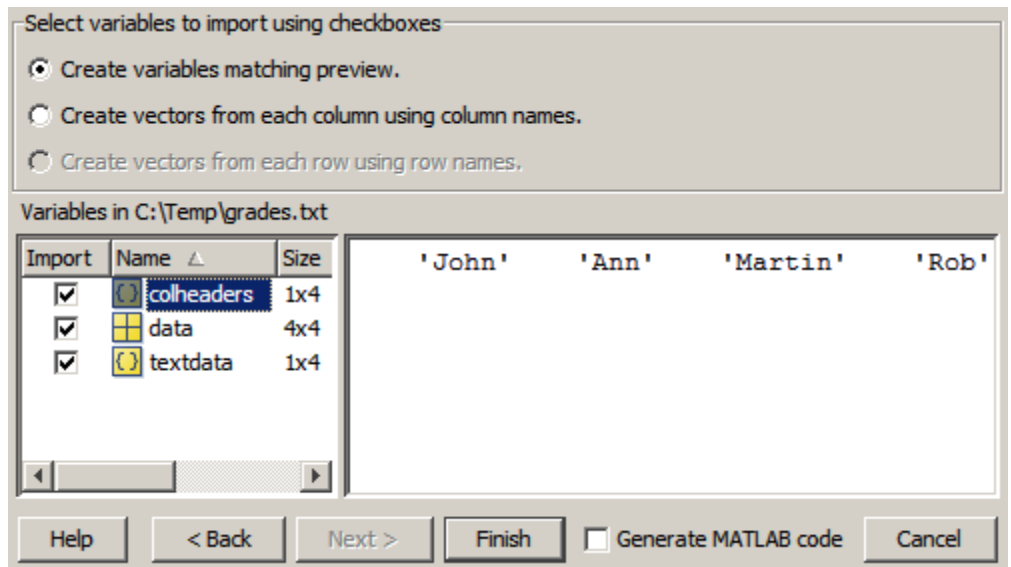
For example, consider a hypothetical text file called `grades.txt` that contains the following data (to create the file, use any text editor, and copy and paste):

John	Ann	Martin	Rob
88.4	91.5	89.2	77.3
83.2	88.0	67.8	91.0
77.8	76.3	78.1	92.5
92.1	96.4	81.2	84.6

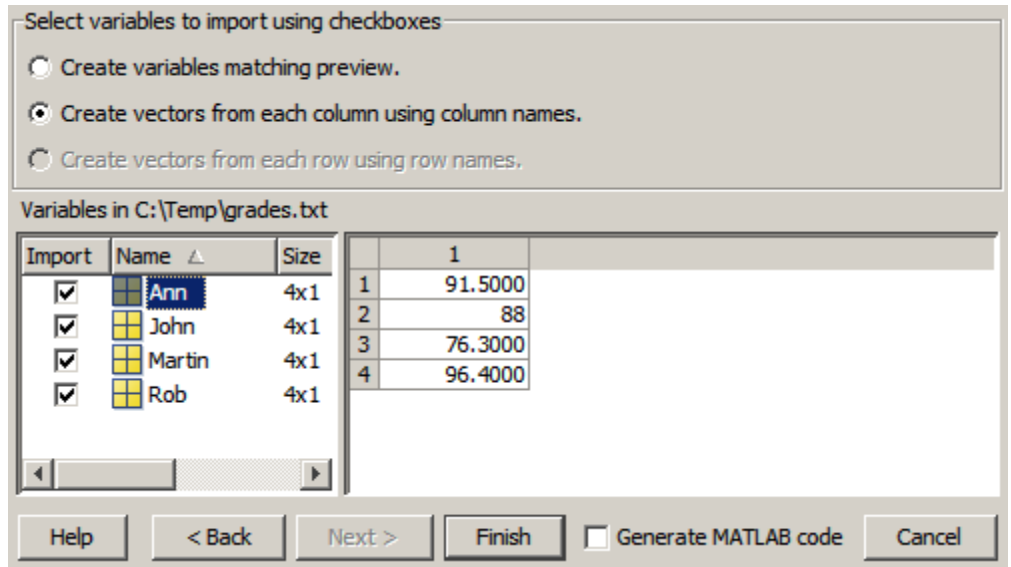
Start the Import Wizard by selecting **File > Import Data**. Select the file and click **Open**. The Import Wizard displays the preview window.



By default, the Import Wizard separates the text and numeric data, and creates three variables: colheaders, data, and textdata. Click the **Next** button.



Select **Create vectors from each column using column names**. The Import Wizard organizes the data into four column vectors named John, Ann, Martin, and Rob.



Generating Reusable MATLAB Code

To create a function that reads similar files without restarting the Import Wizard, select the **Generate MATLAB code** check box. When you click **Finish** to complete the initial import operation, MATLAB opens an Editor window that contains an unsaved function. The default function name is `importfile.m` or `importfileN.m`, where N is an integer.

The function in the generated code includes the following features:

- For text or spreadsheet files, if you request vectors from rows or columns, the generated code also returns vectors.
- When importing from files, the function includes an input argument for the name of the file to import, `fileToRead1`.
- When importing into a structure array, the function includes an output argument for the name of the structure, `newData1`.

However, the generated code has the following limitations:

- If you rename or deselect any variables in the Import Wizard, the generated code does not reflect those changes.
- If you do not import into a structure array, the generated function creates variables in the base workspace. If you plan to call the generated function from within your own function, your function cannot access these variables. To allow your function to access the data, start the Import Wizard by calling `uiimport` with an output argument. Call the generated function with an output argument to create a structure array in the workspace of your function.
- For text files or spreadsheets, the generated code can include specified values for the delimiter, number of header lines, or worksheet name. If other files require different values for these parameters, modify the generated code. For example, consider adding a `sheetName` input argument to the function for importing spreadsheets.

MATLAB does not automatically save the function. To save the file, select **File > Save**. For best results, use the function name with a `.m` extension for the file name.

Example – Generating Code to Import a Text File. Consider the file `grades.txt` discussed in “Creating Column or Row Vectors from Text Files or Spreadsheets” on page 1-9. The file contains the following data:

John	Ann	Martin	Rob
88.4	91.5	89.2	77.3
83.2	88.0	67.8	91.0
77.8	76.3	78.1	92.5
92.1	96.4	81.2	84.6

Use the Import Wizard to create column vectors, select the **Generate MATLAB code** check box, and click **Finish**. MATLAB creates a function in an unsaved document in the Editor:

```
function importfile1(fileToRead1)
%IMPORTFILE1(FILETOREAD1)
% Imports data from the specified file
% FILETOREAD1: file to read
```

```

% Auto-generated by MATLAB on 22-Sep-2009 14:01:02

% Import the file
newData1 = importdata(fileToRead1);

% Break the data up into a new structure with one field per column.
colheaders = genvarname(newData1.textdata);
for i = 1:length(colheaders)
    dataByColumn1.(colheaders{i}) = newData1.data(:, i);
end

% Create new variables in the base workspace from those fields.
vars = fieldnames(dataByColumn1);
for i = 1:length(vars)
    assignin('base', vars{i}, dataByColumn1.(vars{i}));
end

```

Save the function. Create a file `new_grades.txt` that contains the following data:

Michael	Juan	Nita	Steve	Lynn
76.3	89.0	93.1	72.4	81.7
81.9	93.4	90.5	81.8	76.7
80.3	97.8	100.0	89.2	79.6

Import the data using the generated function:

```
importfile1('new_grades.txt')
```

The workspace includes the following variables:

Name	Size	Bytes	Class	Attributes
Juan	3x1	24	double	
Lynn	3x1	24	double	
Michael	3x1	24	double	
Nita	3x1	24	double	
Steve	3x1	24	double	

Importing MAT-Files

In this section...

“Viewing the Contents of a MAT-File” on page 1-14

“Loading a MAT-File” on page 1-14

Viewing the Contents of a MAT-File

MAT-files are binary MATLAB format files that store workspace variables.

To see the variables in a MAT-file before loading the file into your workspace, click the file name in the Current Folder browser. Information about the variables appears in the Details Panel.

Alternatively, use the command `whos -file filename`. This function returns the name, dimensions, size, and class of all variables in the specified MAT-file.

For example, view the contents of the demo file `durer.mat`:

```
whos -file durer.mat
```

MATLAB returns:

Name	Size	Bytes	Class	Attributes
X	648x509	2638656	double	
caption	2x28	112	char	
map	128x3	3072	double	

The byte counts represent the number of bytes that the data occupies when loaded into the MATLAB workspace. Compressed data uses fewer bytes in a file than in the workspace. In Version 7 or higher MAT-files, MATLAB compresses data. For more information, see “MAT-File Versions” on page 2-5.

Loading a MAT-File

To import all variables from a MAT-file, use one of the following options:

- In the Current Folder browser, double-click the name of the file, or right-click the name of the file and select **Open** or **Load**.
- Call the `load` function. For example, the following command loads all variables from the demo file `durer.mat`:

```
load('durer.mat')
```


To load variables into a structure array, specify an output variable for the `load` function:

```
durerStruct = load('durer.mat')
```

Caution When you import data into the MATLAB workspace, the new variables you create overwrite any existing variables in the workspace that have the same name.

Loading Selected Variables

To select and load MAT-file variables interactively, use any of the following options:

- Select **File > Import Data**.
- Click the Import data button  in the Workspace browser toolbar.
- Drag variables from the Details Panel of the Current Folder browser to the Workspace browser.

Alternatively, use the `load` function. For example, load variables `X` and `map` from `durer.mat`:

```
load('durer.mat', 'X', 'map')
```

To load variables with names that match a specific pattern, use one of the following options:

- Include the wildcard character (`*`). For example, load all variables that start with `str` from a file named `strinfo.mat`:

```
load('strinfo.mat', 'str*')
```

- Apply the `-regexp` option, which matches variables to regular expressions. For example, load all variables that start with `Mon`, `Tue`, or `Wed` from a hypothetical file named `myfile.mat`:

```
load('myfile.mat', '-regexp', '^Mon|^Tue|^Wed')
```

For more information, see “Regular Expressions” in the MATLAB Programming Fundamentals documentation.

Troubleshooting: Loading Variables within a Function

If you define a function that loads data from a MAT-file, and find that MATLAB does not return the expected results, check whether any variables in the MAT-file share the same name as a MATLAB function. Common variable names that conflict with function names include `i`, `j`, `mode`, `char`, `size`, or `path`. To determine whether a particular name is associated with a MATLAB function, use the `which` function.

For example, consider a MAT-file with variables `height`, `width`, and `length`. If you load these variables using a function such as `findVolume`,

```
function vol = findVolume(myfile)
    load(myfile);
    vol = height * width * length;
```

MATLAB interprets the reference to `length` as a call to the `length` function, and returns an error:

```
??? Error using ==> length
Not enough input arguments.
```

To avoid confusion, when defining your function, choose one (or more) of the following approaches:

- Load into a structure array. For example, define the `findVolume` function as follows:

```
function vol = findVolume(myfile)
    dims = load(myfile);
    vol = dims.height * dims.width * dims.length;
```


- Explicitly include the names of variables in the call to `load`, as described in “Loading Selected Variables” on page 1-15.
- Initialize variables within the function before calling `load`.

Any of these methods allow you to override the function name with the variable name within your function. For more information, see “Potential Conflict with Function Names” in the MATLAB Programming Fundamentals documentation.

Importing Text Data Files

The easiest way to import data from an ASCII file is to use the Import Wizard, a graphical user interface. To start the Import Wizard, select **File > Import Data**.

To import without invoking a graphical interface, use `importdata`.

For most files, the Import Wizard and `importdata` automatically detect:

- Row and column headers.
- Field delimiters (characters between data items, such as commas, spaces, tabs, or semicolons).
- MATLAB comments (lines that begin with a percent sign, '%').

For example, you can easily read ASCII data in the following form (see “Importing Numeric ASCII Data with Headers” on page 1-20):

Text header line	_____				
	Class Grades for Spring Term				
Column headers	_____	Grade1	Grade2	Grade3	
	John	85	90	95	
Row headers	_____	Ann	90	92	98
	Martin	100	95	97	
	Rob	77	86	93	
Tab-delimited data	_____				

Requirements for the Import Wizard or `importdata`

The data in your file must be:

- Rectangular, like a matrix, with the same number of data fields in each row.
- Numeric. Formatted dates and times (such as '01/01/01' or '12:30:45') are *not* numeric. However, you can import formatted dates and times as headers.

If your data file does not meet these requirements, consider using `textscan`. For more information, see:

- “Importing Nonnumeric ASCII Data” on page 1-23.
- “Importing Nonrectangular ASCII Data” on page 1-24.

To import files with more complex formats, see “Importing Text Data Files with Low-Level I/O” on page 1-26.

Importing a Subset of Your Data

The Import Wizard and `importdata` import all rows and columns of your data file. To import only part of your data, use `dlmread` or `textscan`, where:

- `dlmread` requires rectangular, numeric data, but is easy to use. For more information, see “Selecting a Range of Numeric Data” on page 1-20.
- `textscan` imports a wider variety of file formats, and tracks your position in the file. For more information, see “Importing Large ASCII Data Sets” on page 1-25.

Importing Numeric ASCII Data

You can import any ASCII data file with numeric fields easily using the Import Wizard or `importdata`. For example, consider a comma-delimited ASCII data file named `ph.dat`:

```
7.2, 8.5, 6.2, 6.6
5.4, 9.2, 8.1, 7.2
```

Use `importdata` to import the data. Call `whos` to learn the class of the data returned, and type the name of the output variable (in this case, `'ph'`) to see its contents:

```
ph = importdata('ph.dat');
```

```
whos ph
```

Name	Size	Bytes	Class	Attributes
ph	2x4	64	double	

```
ph
```

```
ph =
```

```
7.2000    8.5000    6.2000    6.6000
5.4000    9.2000    8.1000    7.2000
```

Note As an alternative to `importdata`, you can import data like `ph.dat` with `load`, `dlmread`, or the Import Wizard. All four approaches return identical 2-by-4 double arrays for `ph`.

Selecting a Range of Numeric Data

To select specific rows and columns to import, use `dlmread`. For example, to read the first two columns from `ph.dat`:

```
ph_partial = dlmread('ph.dat', ',', 'A1..B2')

ph_partial =
    7.2000    8.5000
    5.4000    9.2000
```

Importing Formatted Dates and Times

Formatted dates and times (such as `'01/01/01'` or `'12:30:45'`) are not numeric fields. How you import them depends on their location in the file. If the dates and times are:

- In the initial columns, like row headers, use `importdata` or the Import Wizard. For more information, see “Importing Numeric ASCII Data with Headers” on page 1-20.
- In other columns, use `textscan`. For more information, see “Importing Nonnumeric ASCII Data” on page 1-23.

Importing Numeric ASCII Data with Headers

You can import any ASCII data file with numeric fields and text headers easily using the Import Wizard or `importdata`.

For example, consider the file `grades.dat`:

```
Class Grades for Spring Term
    Grade1 Grade2 Grade3
```

John	85	90	95
Ann	90	92	98
Martin	100	95	97
Rob	77	86	93

A call to `importdata` of the form

```
grades_imp = importdata('grades.dat');
```

Returns the same results as a call to the Import Wizard:

```
grades_imp = uiimport('grades.dat');
```

You can also start the Import Wizard by selecting **File > Import Data**.

Because the data includes both row and column headers, `importdata` or the Import Wizard returns the structure `grades_imp` as follows:

```
grades_imp =
    data: [4x3 double]
    textdata: {6x1 cell}

grades_imp.data =
    85    90    95
    90    92    98
   100    95    97
    77    86    93

grades_imp.textdata =
    'Class Grades for Spring Term'
    '      Grade1 Grade2 Grade3'
    'John'
    'Ann'
    'Martin'
    'Rob'
```

Additional Variables and Fields

If your data file includes either column headers or a single column of row headers, but not both:

- You can create vectors based on the rows or columns in your file with the Import Wizard. For more information, see “Creating Column or Row Vectors from Text Files or Spreadsheets” on page 1-9.
- `importdata` and the Import Wizard store the row or column headers in `rowheaders` or `colheaders` fields of the output structure. For example, if `grades_col.dat` includes only column headers:

```
Grade1 Grade2 Grade3
85      90      95
90      92      98
100     95      97
77      86      93
```

A call to `importdata` of the form

```
grades_col = importdata('grades_col.dat');
```

Or a call to the Import Wizard, using the default settings:

```
grades_col = uiimport('grades_col.dat');
```

returns

```
grades_col =
    data: [4x3 double]
    textdata: {'Grade1' 'Grade2' 'Grade3'}
    colheaders: {'Grade1' 'Grade2' 'Grade3'}
```

Restrictions

If your file includes:

- Multiple column headers, `colheaders` contains only the lowest row of header text. However, `textdata` contains all text.
- Nonnumeric characters that are not part of row or column headers, including formatted dates or times, use `textscan` to import the file. For more information, see “Importing Nonnumeric ASCII Data” on page 1-23.

Importing Nonnumeric ASCII Data

To import an ASCII data file with fields that contain nonnumeric characters, use `textscan`.

For example, you can use `textscan` to import a file called `mydata.dat`:

```
Sally    09/12/2005 12.34 45 Yes
Larry    10/12/2005 34.56 54 Yes
Tommy    11/12/2005 67.89 23 No
```

Open the File

Preface any calls to `textscan` with a call to `fopen` to open the file for reading, and, when finished, close the file with `fclose`.

Describe Your Data

The `textscan` function is flexible, but requires that you specify more information about your file. Describe each field using format specifiers, such as `'%s'` for a string, `'%d'` for an integer, or `'%f'` for a floating-point number. (For a complete list of format specifiers, see the `textscan` reference page.)

Import into a Cell Array

Send `textscan` the file identifier and the format specifiers to describe the five fields in each row of `mydata.dat`. `textscan` returns a cell array with five cells:

```
fid = fopen('mydata.dat');
mydata = textscan(fid, '%s %s %f %d %s');
fclose(fid);

whos mydata
  Name          Size          Bytes  Class      Attributes

  mydata        1x5              952   cell

mydata =
  {3x1 cell} {3x1 cell} [3x1 double] [3x1 int32] {3x1 cell}
```

where

```
mydata{1} = {'Sally'; 'Larry'; 'Tommy'}
mydata{2} = {'09/12/2005'; '10/12/2005'; '11/12/2005'}
mydata{3} = [12.3400; 34.5600; 67.8900]
mydata{4} = [45; 54; 23]
mydata{5} = {'Yes'; 'Yes'; 'No'}
```

Importing Nonrectangular ASCII Data

Most of the ASCII data import functions require that your data is *rectangular*, that is, in a regular pattern of columns and rows. The `textscan` function relaxes this restriction, although it requires that your data is in a repeated pattern.

For example, you can use `textscan` to import a file called `nonrect.dat`:

```
begin
v1=12.67
v2=3.14
v3=6.778
end
begin
v1=21.78
v2=5.24
v3=9.838
end
```

Describe Your Data

To use `textscan`, describe the pattern of the data using format specifiers and delimiter parameters. Typical format specifiers include `'%s'` for a string, `'%d'` for an integer, or `'%f'` for a floating-point number. (For a complete list of format specifiers and parameters, see the `textscan` reference page.)

To import `nonrect.dat`, use the format specifier `'%*s'` to tell `textscan` to skip the strings `'begin'` and `'end'`. Include the literals `'v1='`, `'v2='`, and `'v3='` as part of the format specifiers, so that `textscan` ignores those strings as well.

Since each field is on a new line, the `delimiter` is a newline character (`'\n'`). To combine all the floating-point data into a single array, set the `CollectOutput` parameter to `true`. The final call to `textscan` is:


```

fid = fopen('nonrect.dat');

c = textscan(fid, ...
            '%*s v1=%f v2=%f v3=%f %*s', ...
            'Delimiter', '\n', ...
            'CollectOutput', true);

fclose(fid);

whos c
Name      Size      Bytes  Class  Attributes

   c      1x1      108   cell

c{1} =
    12.6700    3.1400    6.7780
    21.7800    5.2400    9.8380

```

Importing Large ASCII Data Sets

To import large data files, consider using `textscan` to read the file in segments, which reduces the amount of memory required.

For example, suppose you want to process the file `largefile.dat` with the user-defined `process_data` function. This example assumes that the `process_data` function processes any number of lines of data, including zero.

```

clear segarray;
block_size = 10000;

% describe the format of the data
% for more information, see the textscan reference page
format = '%s %n %s %8.2f %8.2f %8.2f %8.2f %u8';

file_id = fopen('largefile.dat');

while ~feof(file_id)
    segarray = textscan(file_id, format, block_size);
    process_data(segarray);
end

```

```
fclose(file_id);
```

The `fopen` function positions a pointer at the beginning of the file, and each read operation adjusts the location of that pointer. You can also use low-level file I/O functions such as `fseek` and `frewind` to reposition the pointer within the file. For more information, see “Moving within a File” on page 1-120.

Importing Text Data Files with Low-Level I/O

Low-level file I/O functions allow the most control over reading or writing data to a file. However, these functions require that you specify more detailed information about your file than the easier-to-use *high-level functions*, such as `importdata`. For more information on the high-level functions that read text files, see “Importing Text Data Files” on page 1-18.

If the high-level functions cannot import your data, use one of the following:

- `fscanf`, which reads formatted data in a text or ASCII file; that is, a file you can view in a text editor. For more information, see “Reading Data in a Formatted Pattern” on page 1-27.
- `fgetl` and `fgets`, which read one line of a file at a time, where a newline character separates each line. For more information, see “Reading Data Line-by-Line” on page 1-29.
- `fread`, which reads a stream of data at the byte or bit level. For more information, see “Importing Binary Data with Low-Level I/O” on page 1-116.

For additional information, see:

- “Testing for End of File (EOF)” on page 1-30
- “Opening Files with Different Character Encodings” on page 1-33

Note The low-level file I/O functions are based on functions in the ANSI® Standard C Library. However, MATLAB includes *vectorized* versions of the functions, to read and write data in an array with minimal control loops.

Reading Data in a Formatted Pattern

To import text files that `importdata` and `textscan` cannot read, consider using `fscanf`. The `fscanf` function requires that you describe the format of your file, but includes many options for this format description.

For example, create a text file `mymeas.dat` as shown. The data in `mymeas.dat` includes repeated sets of times, dates, and measurements. The header text includes the number of sets of measurements, `N`:

```
Measurement Data
N=3

12:00:00
01-Jan-1977
4.21 6.55 6.78 6.55
9.15 0.35 7.57 NaN
7.92 8.49 7.43 7.06
9.59 9.33 3.92 0.31
09:10:02
23-Aug-1990
2.76 6.94 4.38 1.86
0.46 3.17 NaN 4.89
0.97 9.50 7.65 4.45
8.23 0.34 7.95 6.46
15:03:40
15-Apr-2003
7.09 6.55 9.59 7.51
7.54 1.62 3.40 2.55
NaN 1.19 5.85 5.05
6.79 4.98 2.23 6.99
```

Opening the File. As with any of the low-level I/O functions, before reading, open the file with `fopen`, and obtain a file identifier. By default, `fopen` opens files for read access, with a permission of `'r'`.

When you finish processing the file, close it with `fclose(fid)`.

Describing the Data. Describe the data in the file with format specifiers, such as `'%s'` for a string, `'%d'` for an integer, or `'%f'` for a floating-point number. (For a complete list of specifiers, see the `fscanf` reference page.)

To skip literal characters in the file, include them in the format description. To skip a data field, use an asterisk ('*') in the specifier.

For example, consider the header lines of `mymeas.dat`:

```
Measurement Data  % skip 2 strings, go to next line:  %*s %*s\n
N=3                % ignore 'N=', read integer:  N=%d\n
                  % go to next line:  \n
12:00:00
01-Jan-1977
4.21  6.55  6.78  6.55
...
```

To read the headers and return the single value for `N`:

```
N = fscanf(fid, '%*s %*s\nN=%d\n\n', 1);
```

Specifying the Number of Values to Read. By default, `fscanf` reapplies your format description until it cannot match the description to the data, or it reaches the end of the file.

Optionally, specify the number of values to read, so that `fscanf` does not attempt to read the entire file. For example, in `mymeas.dat`, each set of measurements includes a fixed number of rows and columns:

```
measrows = 4;
meascols = 4;
meas = fscanf(fid, '%f', [measrows, meascols]);
```

Creating Variables in the Workspace. There are several ways to store `mymeas.dat` in the MATLAB workspace. In this case, read the values into a structure. Each element of the structure has three fields: `mtime`, `mdate`, and `meas`.

Note `fscanf` fills arrays with numeric values in column order. To make the output array match the orientation of numeric data in a file, transpose the array.

```
filename = 'mymeas.dat';
```

```

measrows = 4;
meascols = 4;

% open the file
fid = fopen(filename);

% read the file headers, find N (one value)
N = fscanf(fid, '%*s %*s\nN=%d\n\n', 1);

% read each set of measurements
for n = 1:N
    mystruct(n).mtime = fscanf(fid, '%s', 1);
    mystruct(n).mdate = fscanf(fid, '%s', 1);

    % fscanf fills the array in column order,
    % so transpose the results
    mystruct(n).meas = ...
        fscanf(fid, '%f', [measrows, meascols]);
end

% close the file
fclose(fid);

```

Reading Data Line-by-Line

MATLAB provides two functions that read lines from files and store them in string vectors: `fgetl` and `fgets`. The `fgets` function copies the newline character to the output string, but `fgetl` does not.

The following example uses `fgetl` to read an entire file one line at a time. The function `litcount` determines whether an input literal string (`literal`) appears in each line. If it does, the function prints the entire line preceded by the number of times the literal string appears on the line.

```

function y = litcount(filename, literal)
% Search for number of string matches per line.

fid = fopen(filename);
y = 0;
tline = fgetl(fid);

```

```
while ischar(tline)
    matches = strfind(tline, literal);
    num = length(matches);
    if num > 0
        y = y + num;
        fprintf(1, '%d:%s\n', num, tline);
    end
    tline = fgetl(fid);
end
fclose(fid);
```

Create an input data file called `badpoem`:

```
Oranges and lemons,
Pineapples and tea.
Orangutans and monkeys,
Dragonflys or fleas.
```

To find out how many times the string `'an'` appears in this file, call `litcount`:

```
litcount('badpoem', 'an')
```

This returns:

```
2: Oranges and lemons,
1: Pineapples and tea.
3: Orangutans and monkeys,
ans =
     6
```

Testing for End of File (EOF)

When you read a portion of your data at a time, you can use `feof` to check whether you have reached the end of the file. `feof` returns a value of 1 when the file pointer is at the end of the file. Otherwise, it returns 0.

Note Opening an empty file does *not* move the file position indicator to the end of the file. Read operations, and the `fseek` and `frewind` functions, move the file position indicator.

Testing for EOF with feof. When you use `textscan`, `fscanf`, or `fread` to read portions of data at a time, use `feof` to check whether you have reached the end of the file.

For example, suppose that the hypothetical file `mymeas.dat` has the following form, with no information about the number of measurement sets. Read the data into a structure with fields for `mtime`, `mdate`, and `meas`:

```
12:00:00
01-Jan-1977
4.21 6.55 6.78 6.55
9.15 0.35 7.57 NaN
7.92 8.49 7.43 7.06
9.59 9.33 3.92 0.31
09:10:02
23-Aug-1990
2.76 6.94 4.38 1.86
0.46 3.17 NaN 4.89
0.97 9.50 7.65 4.45
8.23 0.34 7.95 6.46
```

To read the file:

```
filename = 'mymeas.dat';
measrows = 4;
meascols = 4;

% open the file
fid = fopen(filename);

% make sure the file is not empty
finfo = dir(filename);
fsize = finfo.bytes;

if fsize > 0

    % read the file
    block = 1;
    while ~feof(fid)
        mystruct(block).mtime = fscanf(fid, '%s', 1);
```

```
        mystruct(block).mdate = fscanf(fid, '%s', 1);

        % fscanf fills the array in column order,
        % so transpose the results
        mystruct(block).meas = ...
            fscanf(fid, '%f', [measrows, meascols]);

        block = block + 1;
    end

end

% close the file
fclose(fid);
```

Testing for EOF with `fgetl` and `fgets`. If you use `fgetl` or `fgets` in a control loop, `feof` is not always the best way to test for end of file. As an alternative, consider checking whether the value that `fgetl` or `fgets` returns is a character string.

For example, the function `litcount` described in “Reading Data Line-by-Line” on page 1-29 includes the following while loop and `fgetl` calls :

```
y = 0;
tline = fgetl(fid);
while ischar(tline)
    matches = strfind(tline, literal);
    num = length(matches);
    if num > 0
        y = y + num;
        fprintf(1, '%d:%s\n', num, tline);
    end
    tline = fgetl(fid);
end
```

This approach is more robust than testing `~feof(fid)` for two reasons:

- If `fgetl` or `fgets` find data, they return a string. Otherwise, they return a number (-1).

- After each read operation, `fgetl` and `fgets` check the next character in the file for the end-of-file marker. Therefore, these functions sometimes set the end-of-file indicator *before* they return a value of `-1`. For example, consider the following three-line text file. Each of the first two lines ends with a newline character, and the third line contains only the end-of-file marker:

```
123
456
```

Three sequential calls to `fgetl` yield the following results:

```
t1 = fgetl(fid);    % t1 = '123', feof(fid) = false
t2 = fgetl(fid);    % t2 = '456', feof(fid) = true
t3 = fgetl(fid);    % t3 = -1,    feof(fid) = true
```

This behavior does not conform to the ANSI specifications for the related C language functions.

Opening Files with Different Character Encodings

Encoding schemes support the characters required for particular alphabets, such as those for Japanese or European languages. Common encoding schemes include US-ASCII or UTF-8.

If you do not specify an encoding scheme, `fopen` opens files for processing using the default encoding for your system. To determine the default, open a file, and call `fopen` again with the syntax:

```
[filename, permission, machineformat, encoding] = fopen(fid);
```

If you specify an encoding scheme when you open a file, the following functions apply that scheme: `fscanf`, `fprintf`, `fgetl`, `fgets`, `fread`, and `fwrite`.

For a complete list of supported encoding schemes, and the syntax for specifying the encoding, see the `fopen` reference page.

Importing XML Documents

To read an XML file from your local disk or from a URL, use the `xmlread` function. `xmlread` returns the contents of the file in a Document Object Model (DOM) node.

What Is an XML Document Object Model (DOM)?

In a Document Object Model, every item in an XML file corresponds to a node. The properties and methods for DOM nodes (that is, the way you create and access nodes) follow standards set by the World Wide Web consortium.

For example, consider this sample XML file:

```
<productinfo
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://www.mathworks.com/namespace/info/v1/info.xsd">

  <!-- This is a sample info.xml file. -->

  <list>

    <listitem>
      <label>Import Wizard</label>
      <callback>uiimport</callback>
      <icon>ApplicationIcon.GENERIC_GUI</icon>
    </listitem>

    <listitem>
      <label>Profiler</label>
      <callback>profile viewer</callback>
      <icon>ApplicationIcon.PROFILER</icon>
    </listitem>

  </list>
</productinfo>
```

The information in the file maps to the following types of nodes in a DOM:

- *Element nodes* — Corresponds to tag names. In the sample `info.xml` file, these tags correspond to element nodes:
 - `productinfo`
 - `list`
 - `listitem`
 - `label`
 - `callback`
 - `icon`

In this case, the `list` element is the *parent* of `listitem` element *child* nodes. The `productinfo` element is the *root* element node.

- *Text nodes* — Contains values associated with element nodes. Every text node is the child of an element node. For example, the Import Wizard text node is the child of the first `label` element node.
- *Attribute nodes* — Contains name and value pairs associated with an element node. For example, `xmlns:xsi` is the name of an attribute and `http://www.w3.org/2001/XMLSchema-instance` is its value. Attribute nodes are not parents or children of any nodes.
- *Comment nodes* — Includes additional text in the file, in the form `<!--Sample comment-->`.
- *Document nodes* — Corresponds to the entire file. Use methods on the document node to create new element, text, attribute, or comment nodes.

For a complete list of the methods and properties of DOM nodes, select the `org.w3c.dom` package at:
<https://jaxp-sources.dev.java.net/nonav/docs/api/>.

Example — Finding Text in an XML File

The full `matlabroot/toolbox/matlab/general/info.xml` file contains several `listitem` elements, such as:

```
<listitem>
<label>Import Wizard</label>
<callback>uiimport</callback>
<icon>ApplicationIcon.GENERIC_GUI</icon>
```

```
</listitem>
```

One of the `label` elements has the child text `Plot Tools`. Suppose that you want to find the text for the `callback` element in the same `listitem`. Follow these steps:

- 1 Initialize your variables, and call `xmlread` to obtain the document node:

```
findLabel = 'Plot Tools';  
findCbk = '';  
  
xDoc = xmlread(fullfile(matlabroot, ...  
                        'toolbox','matlab','general','info.xml'));
```

- 2 Find all the `listitem` elements. The `getElementsByTagName` method returns a deep list that contains information about the child nodes:

```
allListItems = xDoc.getElementsByTagName('listitem');
```

Note Lists returned by DOM methods use zero-based indexing.

- 3 For each `listitem`, compare the text for the `label` element to the text you want to find. When you locate the correct label, get the `callback` text:

```
for k = 0:allListItems.getLength-1  
    thisListitem = allListItems.item(k);  
  
    % Get the label element. In this file, each  
    % listitem contains only one label.  
    thisList = thisListitem.getElementsByTagName('label');  
    thisElement = thisList.item(0);  
  
    % Check whether this is the label you want.  
    % The text is in the first child node.  
    if strcmp(thisElement.getFirstChild.getData, findLabel)  
        thisList = thisListitem.getElementsByTagName('callback');  
        thisElement = thisList.item(0);  
        findCbk = char(thisElement.getFirstChild.getData);  
        break;
```

```
end
```

```
end
```

4 Display the final results:

```
if ~isempty(findCbk)
    msg = sprintf('Item "%s" has a callback of "%s."',...
                 findLabel, findCbk);
else
    msg = sprintf('Did not find the "%s" item.', findLabel);
end
disp(msg);
```

For an additional example that creates a structure array to store data from an XML file, see the `xmlread` function reference page.

Importing Excel Spreadsheets

There are several ways to read data from an Excel spreadsheet file into the MATLAB workspace:

- “Pasting Excel Data from the Clipboard” on page 1-38.
- “Selecting the Spreadsheet and Variables Interactively” on page 1-38.
- “Reading a Specific Worksheet and Range of Data” on page 1-39.
- “Reading All Worksheets into a Structure” on page 1-40.

Some import options require that your system includes Excel for Windows®. For more information, see “Excel Version Support” on page 1-41.

Note Excel and MATLAB can store dates as strings (such as '10/31/96') or numbers (such as 35369 or 729329). If your file includes *numeric* dates, see “Converting Dates” on page 1-42.

Pasting Excel Data from the Clipboard

Paste Excel data from the clipboard into MATLAB using one of the following methods:

- Select **Edit > Paste to Workspace**.
- Call `uiimport -pastespecial`.
- Open an existing variable in the MATLAB Variable Editor, and select **Edit > Paste Excel Data**.

Selecting the Spreadsheet and Variables Interactively

The Import Wizard is a graphical user interface that helps you find a file, specify a worksheet, and define the variables to use in the workspace. For worksheets with row or column headers, the Import Wizard allows you to create variables based on the rows or columns. However, you cannot specify a range of data to import.

To start the Import Wizard, use one of the following methods:

- Select **File > Import Data**.
- Double-click a file name in the Current Folder browser.
- Call `uiimport`.

Reading a Specific Worksheet and Range of Data

Consider the file `climate.xls` created with `xlswrite` as follows:

```
d = {'Time', 'Temp';
    12 98;
    13 99;
    14 97}

xlswrite('climate.xls', d, 'Temperatures');
```

To import the numeric data into a matrix, use `xlsread` with a single return argument. `xlsread` ignores any leading row or column of text in the numeric result:

```
ndata = xlsread('climate.xls', 'Temperatures')

ndata =
    12    98
    13    99
    14    97
```

To import both numeric data and text data, specify two return values for `xlsread`:

```
[ndata, headertext] = xlsread('climate.xls', 'Temperatures')

ndata =
    12    98
    13    99
    14    97

headertext =
    'Time'    'Temp'
```

To read only the first row of data, specify the range:

```
firstrow = xlsread('climate.xls', 'Temperatures', 'A2:B2')

firstrow =
    12    98
```

Getting Information about a Spreadsheet

To determine whether a file contains a readable Excel spreadsheet, use the `xlsfinfo` function. For readable files, `xlsfinfo` returns the string 'Microsoft Excel Spreadsheet'. Otherwise, it returns an empty string ('').

You also can use `xlsfinfo` to identify the names of the worksheets in the file, and to obtain the file format reported by Excel. For example, retrieve information on the spreadsheet `climate.xls`:

```
[type, sheets] = xlsfinfo('climate.xls')

type =
Microsoft Excel Spreadsheet
sheets =
    'Sheet1'    'Sheet2'    'Sheet3'    'Temperatures'
```

Reading All Worksheets into a Structure

The `importdata` function reads data from an Excel file into a structure. Continuing the example in “Reading a Specific Worksheet and Range of Data” on page 1-39, where the data includes column headers, a call of the form

```
climate = importdata('climate.xls')    % with column headers
```

returns the nested structure array

```
climate =
    data: [1x1 struct]
    textdata: [1x1 struct]
    colheaders: [1x1 struct]
```

Structures created from Excel files with row headers include the field `rowheaders`, which also contains a 1-by-1 structure.

The structure named `data` contains one field for each worksheet with numeric data. The other structures contain one field for each worksheet with text cells or headers. In this case:

```
climate.data =
    Temperatures: [3x2 double]

climate.textdata =
    Temperatures: {'Time' 'Temp'}

climate.colheaders =
    Temperatures: {'Time' 'Temp'}
```

If the Excel file contains only numeric data (no row or column headers, and no inner cells with text), the output structure is simpler. `importdata` returns a 1-by-1 structure, with one field for each worksheet with data.

For example, if the `Temperatures` worksheet in `climate_nums.xls` does not include column headers, the call

```
ndata = importdata('climate_nums.xls')    % only numeric data
```

returns

```
ndata =
    Temperatures: [3x2 double]
```

Excel Version Support

If your system has Excel for Windows installed, including the COM server (part of the typical installation of Excel):

- All MATLAB import options support XLS, XLSX, XLSB, and XLSM formats. `xlsread` also imports HTML-based formats.

If you have Excel 2003 installed, but want to read a 2007 format (such as XLSX, XLSB, or XLSM), install the Office 2007 Compatibility Pack.

- `xlsread` includes an option to open Excel and select the range of data interactively. To use this option, call `xlsread` with the following syntax:

```
mydata = xlsread(filename, -1)
```

Restrictions

If your system does not have Excel for Windows installed, or the COM server is not available:

- The Import Wizard, `importdata`, and `xlsread` only read XLS files compatible with Excel 97-2003.
- You can specify a worksheet to read in the Excel file with the `xlsread` function, but you cannot specify a range of data. See the `xlsread` reference page for additional information.

Note Large files in XLSX format sometimes load slowly. For better import and export performance with Excel 2007 files, Microsoft recommends that you use the XLSB format.

Converting Dates

In both MATLAB and Excel applications, dates can be represented as character strings or numeric values. For example, May 31, 2009, can be represented as the character string `'05/31/09'` or as the numeric value 733924. Within MATLAB, The `datestr` and `datenum` functions allow you to convert easily between string and numeric representations.

If you import a spreadsheet with dates stored as strings, you do not need to convert the dates before processing in MATLAB.

However, if you import a spreadsheet with dates stored as numbers, you must convert the dates. Both Excel and MATLAB represent numeric dates as a number of serial days elapsed from a specific reference date, but the applications use different reference dates.

The following table lists the reference dates for MATLAB and Excel. For more information on the 1900 and 1904 date systems, see the Excel help.

Application	Reference Date
MATLAB	January 0, 0000
Excel for Windows	January 1, 1900
Excel for the Macintosh®	January 2, 1904

Example – Importing an Excel File with Numeric Dates

Consider the hypothetical file `weight_log.xls` with

Date	Weight
10/31/96	174.8
11/29/96	179.3
12/30/96	190.4
01/31/97	185.7

To import this file, first convert the dates within Excel to a numeric format. In Windows, the file now appears as

Date	Weight
35369	174.8
35398	175.3
35429	190.4
35461	185.7

Import the file:

```
wt = xlsread('weight_log.xls');
```

Convert the dates to the MATLAB reference date. If the file uses the 1900 date system (the default in Excel for Windows):

```
datecol = 1;
wt(:,datecol) = wt(:,datecol) + datenum('30-Dec-1899');
```

If the file uses the 1904 date system (the default in Excel for the Macintosh):

```
datecol = 1;
wt(:,datecol) = wt(:,datecol) + datenum('01-Jan-1904');
```

Importing Scientific Data Files

In this section...

- “Importing Common Data File Format (CDF) Files” on page 1-44
- “Importing Network Common Data Form (netCDF) Files” on page 1-51
- “Importing Flexible Image Transport System (FITS) Files” on page 1-56
- “Importing Hierarchical Data Format (HDF5) Files” on page 1-58
- “Importing Hierarchical Data Format (HDF4) Files” on page 1-73

Importing Common Data File Format (CDF) Files

CDF was created by the National Space Science Data Center (NSSDC) to provide a self-describing data storage and manipulation format that matches the structure of scientific data and applications (i.e., statistical and numerical methods, visualization, and management). For more information about this format, see the CDF Web site.

MATLAB provides two ways to access CDF files: a set of high-level functions and a package of low-level functions that provide direct access to the routines in the CDF C API library. The high level functions provide a simpler interface to accessing CDF files. However, if you require more control over the import operation, such as data subsetting for large data sets, use the low-level functions. The following sections provide more information.

- “High-Level CDF Import Functions” on page 1-44
- “Using the CDF Library Low-Level Functions to Import Data” on page 1-48

High-Level CDF Import Functions

MATLAB includes high-level functions that you can use to get information about the contents of a Common Data Format (CDF) file and then read data from the file. The following sections provide more information.

- “Getting Information about the Contents of CDF File” on page 1-45
- “Reading Data from a CDF File ” on page 1-46
- “Speeding Up Read Operations” on page 1-46

- “Representing CDF Time Values” on page 1-48

Getting Information about the Contents of CDF File. To get information about the contents of a CDF file, such as the names of variables in the CDF file, use the `cdfinfo` function. The `cdfinfo` function returns a structure containing general information about the file and detailed information about the variables and attributes in the file.

In this example, the `Variables` field indicates the number of variables in the file. Taking a closer look at the contents of this field, you can see that the first variable, `Time`, is made up of 24 records containing CDF epoch data. The next two variables, `Longitude` and `Latitude`, have only one associated record containing `int8` data. For details about how to interpret the data returned in the `Variables` field, see `cdfinfo`.

Note Because `cdfinfo` creates temporary files, make sure that your current working directory is writable before attempting to use the function.

```
info = cdfinfo('example.cdf')

info =

    Filename: 'example.cdf'
  FileModDate: '19-May-2010 12:03:11'
    FileSize: 1310
      Format: 'CDF'
  FormatVersion: '2.7.0'
  FileSettings: [1x1 struct]
    Subfiles: {}
    Variables: {6x6 cell}
  GlobalAttributes: [1x1 struct]
  VariableAttributes: [1x1 struct]

vars = info.Variables

vars =

    'Time'                [1x2 double]    [24]    'epoch'    'T/'    'Full'
```

```

'Longitude'      [1x2 double] [ 1] 'int8'   'F/FT'   'Full'
'Latitude'      [1x2 double] [ 1] 'int8'   'F/TF'   'Full'
'Data'          [1x3 double] [ 1] 'double' 'T/TTT'  'Full'
'multidimensional' [1x4 double] [ 1] 'uint8'  'T/TTTT' 'Full'
'Temperature'   [1x2 double] [10] 'int16'  'T/TT'   'Full'

```

Reading Data from a CDF File . To read all of the data in the CDF file, use the `cdfread` function. The function returns the data in a cell array. The columns of data correspond to the variables; the rows correspond to the records associated with a variable.

```
data = cdfread('example.cdf');
```

```
whos data
Name      Size      Bytes Class  Attributes
data      24x6      16512 cell
```

To read the data associated with one or more particular variables, use the `'Variable'` parameter. Specify the names of the variables as text strings in a cell array. Variable names are case sensitive. The following example reads the `Longitude` and `Latitude` variables from the file.

```
var_long_lat = cdfread('example.cdf','Variable',{'Longitude','Latitude'});
```

```
whos var_long_lat
Name      Size      Bytes Class  Attributes
var_long_lat 1x2      128   cell
```

Speeding Up Read Operations. The `cdfread` function offers two ways to speed up read operations when working with large data sets:

- Reducing the number of elements in the returned cell array
- Returning CDF epoch values as MATLAB serial date numbers rather than as MATLAB `cdfepoch` objects

To reduce the number of elements in the returned cell array, specify the `'CombineRecords'` parameter. By default, `cdfread` creates a cell array with a separate element for every variable and every record in each variable,

padding the records dimension to create a rectangular cell array. For example, reading all the data from the example file produces an output cell array, 24-by-6, where the columns represent variables and the rows represent the records for each variable. When you set the 'CombineRecords' parameter to true, `cdfread` creates a separate element for each variable but saves time by putting all the records associated with a variable in a single cell array element. Thus, reading the data from the example file with 'CombineRecords' set to true produces a 1-by-5 cell array, as shown below.

```
data_combined = cdfread('example.cdf','CombineRecords',true);
```

```
whos
  Name              Size              Bytes  Class  Attributes

  data              24x6              16512  cell

  data_combined     1x6              2544   cell
```

When combining records, note that the dimensions of the data in the cell change. For example, if a variable has 20 records, each of which is a scalar value, the data in the cell array for the combined element contains a 20-by-1 vector of values. If each record is a 3-by-4 array, the cell array element contains a 20-by-3-by-4 array. For combined data, `cdfread` adds a dimension to the data, the first dimension, that is the index into the records.

Another way to speed up read operations is to read CDF epoch values as MATLAB serial date numbers. By default, `cdfread` creates a MATLAB `cdfepoch` object for each CDF epoch value in the file. If you specify the 'ConvertEpochToDatenum' parameter, setting it to true, `cdfread` returns CDF epoch values as MATLAB serial date numbers. For more information about working with MATLAB `cdfepoch` objects, see “Representing CDF Time Values” on page 1-48.

```
data_datenums = cdfread('example.cdf','ConvertEpochToDatenum',true);
```

```
whos
  Name              Size              Bytes  Class  Attributes

  data              24x6              16512  cell

  data_combined     1x6              2544   cell

  data_datenums     24x6              13536  cell
```

Representing CDF Time Values. CDF represents time differently than MATLAB. CDF represents date and time as the number of milliseconds since 1-Jan-0000. This is called an *epoch* in CDF terminology. MATLAB represents date and time as a serial date number, which is the number of days since 0-Jan-0000. To represent CDF dates, MATLAB uses an object called a CDF epoch object. To access the time information in a CDF object, use the object's `todatenum` method.

For example, this code extracts the date information from a CDF epoch object:

- 1 Extract the date information from the CDF epoch object returned in the cell array `data` (see “Importing Common Data File Format (CDF) Files” on page 1-44). Use the `todatenum` method of the CDF epoch object to get the date information, which is returned as a MATLAB serial date number.

```
m_date = todatenum(data{1});
```

- 2 View the MATLAB serial date number as a string.

```
datestr(m_date)
ans =
01-Jan-2001
```

Using the CDF Library Low-Level Functions to Import Data

To import (read) data from a Common Data Format (CDF) file, you can use the MATLAB low-level CDF functions. The MATLAB functions correspond to dozens of routines in the CDF C API library. For a complete list of all the MATLAB low-level CDF functions, see `cdflib`.

This section does not attempt to describe all features of the CDF library or explain basic CDF programming concepts. To use the MATLAB CDF low-level functions effectively, you must be familiar with the CDF C interface. Documentation about CDF, version 3.3.0, is available at the [CDF Web site](#).

The following example shows how to use low-level functions to read data from a CDF file.

- 1 Open the sample CDF file. For information about creating a new CDF file, see “Using the Low-level CDF Functions to Export Data” on page 2-27.


```
cdfid = cdflib.open('example.cdf');
```

- 2** Get some information about the contents of the file, such as the number of variables in the file, the number of global attributes, and the number of attributes with variable scope.

```
info = cdflib.inquire(cdfid)
```

```
info =
```

```
    encoding: 'IBMPC_ENCODING'  
    majority: 'ROW_MAJOR'  
      maxRec: 23  
      numVars: 6  
    numVAttrs: 1  
    numGAttrs: 3
```

- 3** Get information about the individual variables in the file. Variable ID numbers start at zero.

```
info = cdflib.inquireVar(cdfid,0)
```

```
info =
```

```
      name: 'Time'  
    datatype: 'cdf_epoch'  
numElements: 1  
      dims: []  
    recVariance: 1  
    dimVariance: []
```

```
info = cdflib.inquireVar(cdfid,1)
```

```
info =
```

```
      name: 'Longitude'  
    datatype: 'cdf_int1'  
numElements: 1  
      dims: [2 2]  
    recVariance: 0
```

```
dimVariance: [1 0]
```

- 4** Read the data in a variable into the workspace. The first variable contains CDF Epoch time values. The low-level interface returns these as double values.

```
data_time = cdflib.getVarRecordData(cdfid,0,0)
```

```
data_time =
```

```
6.3146e+013
```

```
% convert the time value to a time vector  
timeVec = cdflib.epochBreakdown(data_time)
```

```
timeVec =
```

```
2001
```

```
1
```

```
1
```

```
0
```

```
0
```

```
0
```

```
0
```

- 5** Read a global attribute from the file.

```
% Determine which attributes are global.
```

```
info = cdflib.inquireAttr(cdfid,0)
```

```
info =
```

```
name: 'SampleAttribute'
```

```
scope: 'GLOBAL_SCOPE'
```

```
maxgEntry: 4
```

```
maxEntry: -1
```

```
% Read the value of the attribute. Note you must use the  
% cdflib.getAttrgEntry function for global attributes.
```

```
value = cdflib.getAttrgEntry(cdfid,0,0)
```

```
value =
```

```
This is a sample entry.
```

6 Close the CDF file.

```
cdflib.close(cdfid);
```

Importing Network Common Data Form (netCDF) Files

Note For information about importing to Common Data Format (CDF) files, which have a completely separate, incompatible format, see “Importing Common Data File Format (CDF) Files” on page 1-44.

MATLAB provides access to the routines in the netCDF C library that you can use to read data from netCDF files and write data to netCDF files. MATLAB provides this access through a set of MATLAB functions that correspond to the functions in the netCDF C library. MATLAB groups the functions into a package, called `netcdf`. To call one of the functions in the package, you must specify the package name. For a complete list of all the functions, see `netcdf`.

Note The MATLAB netCDF functions support netCDF Version 3.6.2.

This section does not attempt to describe all features of the netCDF library or explain basic netCDF programming concepts. To use the MATLAB netCDF functions effectively, you should be familiar with the information about netCDF contained in the *NetCDF C Interface Guide* for version 3.6.2.

Mapping netCDF API Syntax to MATLAB Function Syntax

For the most part, the MATLAB netCDF functions correspond directly to routines in the netCDF C library. For example, the MATLAB function `netcdf.open` corresponds to the netCDF library routine `nc_open`. In some cases, one MATLAB function corresponds to a group of netCDF library functions. For example, instead of creating MATLAB versions of every netCDF library `nc_put_att_type` function, where *type* represents a data

type, MATLAB uses one function, `netcdf.putAtt`, to handle all supported data types.

The syntax of the MATLAB functions is similar to the netCDF library routines, with some exceptions. For example, the netCDF C library routines use input parameters to return data, while their MATLAB counterparts use one or more return values. For example, the following is the function signature of the `nc_open` routine in the netCDF library. Note how the netCDF file identifier is returned in the `ncidp` argument.

```
int nc_open (const char *path, int omode, int *ncidp); /* C syntax */
```

The following shows the signature of the corresponding MATLAB function, `netcdf.open`. Like its netCDF C library counterpart, the MATLAB netCDF function accepts a character string that specifies the file name and a constant that specifies the access mode. Note, however, that the MATLAB `netcdf.open` function returns the file identifier, `ncid`, as a return value.

```
ncid = netcdf.open(filename, mode)
```

To see a list of all the functions in the MATLAB netCDF package, see the netCDF reference page.

Exploring the Contents of a netCDF File

This example shows how to use the MATLAB netCDF functions to explore the contents of a netCDF file. The section uses the example netCDF file included with MATLAB, `example.nc`, as an illustration. For an example of reading data from a netCDF file, see “Reading Data from a netCDF File” on page 1-55

- 1 Open the netCDF file using the `netcdf.open` function. This function returns an identifier that you use thereafter to refer to the file. The example opens the file for read-only access, but you can specify other access modes. For more information about modes, see `netcdf.open`.

```
ncid = netcdf.open('example.nc', 'NC_NOWRITE');
```

- 2 Explore the contents of the file using the `netcdf.inq` function. This function returns the number of dimensions, variables, and global attributes in the file, and returns the identifier of the unlimited dimension in the file. (An unlimited dimension can grow.)

```

[ndims,nvars,natts,unlimdimID]= netcdf.inq(ncid)
ndims =

    3

nvars =

    3

natts =

    1

unlimdimID =

   -1

```

- 3** Get more information about the dimensions, variables, and global attributes in the file by using netCDF inquiry functions. For example, to get information about the global attribute, first get the name of the attribute, using the `netcdf.inqAttName` function. After you get the name, 'creation_date' in this case, you can use the `netcdf.inqAtt` function to get information about the data type and length of the attribute.

To get the name of an attribute, you must specify the ID of the variable the attribute is associated with and the attribute number. To access a global attribute, which isn't associated with a particular variable, use the constant 'NC_GLOBAL' as the variable ID. The attribute number is a zero-based index that identifies the attribute. For example, the first attribute has the index value 0, and so on.

```

global_att_name = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0)

global_att_name =

creation_date

```

```
[xtype attlen] = netcdf.inqAtt(ncid,netcdf.getConstant('NC_GLOBAL'),global_att_name)

xtype =

    2

attlen =

    11
```

- 4** Get the value of the attribute, using the `netcdf.getAtt` function.

```
global_att_value = netcdf.getAtt(ncid,netcdf.getConstant('NC_GLOBAL'),global_att_name)

global_att_value =

29-Mar-2010
```

- 5** Get information about the dimensions defined in the file through a series of calls to `netcdf.inqDim`. This function returns the name and length of the dimension. The `netcdf.inqDim` function requires the dimension ID, which is a zero-based index that identifies the dimensions. For example, the first dimension has the index value 0, and so on.

```
[dimname, dimlen] = netcdf.inqDim(ncid,0)

dimname =

x

dimlen =

    50
```

- 6** Get information about the variables in the file through a series of calls to `netcdf.inqVar`. This function returns the name, data type, dimension ID, and the number of attributes associated with the variable. The `netcdf.inqVar` function requires the variable ID, which is a zero-based index that identifies the variables. For example, the first variable has the index value 0, and so on.

```
[varname, vartype, dimids, natts] = netcdf.inqVar(ncid,0)

varname =

avagadros_number

vartype =

        6

dimids =

        []

natts =

        1
```

The data type information returned in `vartype` is the numeric value of the netCDF data type constants, such as `NC_INT` and `NC_BYTE`. See the official netCDF documentation for information about these constants.

Reading Data from a netCDF File

After you understand the contents of a netCDF file, by using the inquiry functions, you can retrieve the data from the variables and attributes in the file. To read the data associated with the variable `avagadros_number` in the example file, use the `netcdf.getVar` function. The following example uses the netCDF file identifier returned in the previous section, “Exploring the Contents of a netCDF File” on page 1-52. The variable ID is a zero-based index that identifies the variables. For example, the first variable has the index value 0, and so on. (To learn how to write data to a netCDF file, see “Storing Data in a netCDF File” on page 2-29.)

```
A_number = netcdf.getVar(ncid,0)

A_number =
```

6.0221e+023

The netCDF functions automatically choose the MATLAB class that best matches the netCDF data type, but you can also specify the class of the return data by using an optional argument to `netcdf.getVar`. The following table shows the default mapping. For more information about netCDF data types, see the NetCDF C Interface Guide for version 3.6.2.

netCDF Data Type	MATLAB Class	Notes
NC_BYTE	int8 or uint8	netCDF interprets byte data as either signed or unsigned.
NC_CHAR	char	
NC_SHORT	int16	
NC_INT	int32	
NC_FLOAT	single	
NC_DOUBLE	double	

Importing Flexible Image Transport System (FITS) Files

The FITS file format is the standard data format used in astronomy, endorsed by both NASA and the International Astronomical Union (IAU). For more information about the FITS standard, go to the official FITS Web site, <http://fits.gsfc.nasa.gov/>.

The FITS file format is designed to store scientific data sets consisting of multidimensional arrays (1-D spectra, 2-D images, or 3-D data cubes) and two-dimensional tables containing rows and columns of data. A data file in FITS format can contain multiple components, each marked by an ASCII text header followed by binary data. The first component in a FITS file is known as the *primary*, which can be followed by any number of other components, called *extensions*, in FITS terminology.

To get information about the contents of a Flexible Image Transport System (FITS) file, use the `fitsinfo` function. The `fitsinfo` function returns a structure containing the information about the file and detailed information about the data in the file.

To import data into the MATLAB workspace from a Flexible Image Transport System (FITS) file, use the `fitsread` function. Using this function, you can import the data in the `PrimaryData` section of the file or you can import the data in any of the extensions in the file, such as the `Image` extension. This example illustrates how to use the `fitsread` function to read data from a sample FITS file included with MATLAB:

- 1 Determine which extensions the FITS file contains, using the `fitsinfo` function.

```
info = fitsinfo('tst0012.fits')

info =

    Filename: 'matlabroot\tst0012.fits'
  FileModDate: '12-Mar-2001 18:37:46'
    FileSize: 109440
   Contents: {1x5 cell}
 PrimaryData: [1x1 struct]
 BinaryTable: [1x1 struct]
    Unknown: [1x1 struct]
     Image: [1x1 struct]
  AsciiTable: [1x1 struct]
```

The `info` structure shows that the file contains several extensions including the `BinaryTable`, `AsciiTable`, and `Image` extensions.

- 2 Read data from the file.

To read the `PrimaryData` in the file, specify the filename as the only argument:

```
pdata = fitsread('tst0012.fits');
```

To read any of the extensions in the file, you must specify the name of the extension as an optional parameter. This example reads the `BinaryTable` extension from the FITS file:

```
bindata = fitsread('tst0012.fits','bintable');
```

Note To read the `BinaryTable` extension using `fitsread`, you must specify the parameter `'bintable'`. Similarly, to read the `AsciiTable` extension, you must specify the parameter `'table'`. See the `fitsread` reference page for more information.

Importing Hierarchical Data Format (HDF5) Files

Hierarchical Data Format, Version 5, (HDF5) is a general-purpose, machine-independent standard for storing scientific data in files, developed by the National Center for Supercomputing Applications (NCSA). HDF5 is used by a wide range of engineering and scientific fields that want a standard way to store data so that it can be shared. For more information about the HDF5 file format, read the HDF5 documentation available at the HDF Web site (<http://www.hdfgroup.org>).

The MATLAB *high-level* HDF5 function `hdf5read` provides an easy way to import data from an HDF5 file. In addition, you can use `hdf5info` to get information about an HDF5 file. These functions are discussed in the following sections:

- “Determining the Contents of an HDF5 File” on page 1-59
- “Importing Data from an HDF5 File” on page 1-62
- “Mapping HDF5 Data Types to MATLAB Data Types” on page 1-63

MATLAB also provides direct access to the over 200 functions in the HDF5 library with *low-level* functions that correspond to the functions in the HDF5 library. In this way, you can access the features of the HDF5 library from MATLAB, such as reading and writing complex data types and using the HDF5 subsetting capabilities. For more information, see “Using the MATLAB Low-Level HDF5 Functions” on page 2-34.

Note For information about importing to HDF4 files, which have a completely separate, incompatible format, see “Importing Hierarchical Data Format (HDF4) Files” on page 1-73.

Determining the Contents of an HDF5 File

HDF5 files can contain data and metadata, called *attributes*. HDF5 files organize the data and metadata in a hierarchical structure similar to the hierarchical structure of a UNIX® file system.

In an HDF5 file, the directories in the hierarchy are called *groups*. A group can contain other groups, data sets, attributes, links, and data types. A data set is a collection of data, such as a multidimensional numeric array or string. An attribute is any data that is associated with another entity, such as a data set. A link is similar to a UNIX file system symbolic link. Links are a way to reference data without having to make a copy of the data.

Data types are a description of the data in the data set or attribute. Data types tell how to interpret the data in the data set. For example, a file might contain a data type called “Reading” that is comprised of three elements: a longitude value, a latitude value, and a temperature value.

To explore the hierarchical organization of an HDF5 file, use the `hdf5info` function. For example, to find out what the sample HDF5 file, `example.h5`, contains, use this syntax:

```
fileinfo = hdf5info('example.h5');
```

`hdf5info` returns a structure that contains various information about the HDF5 file, including the name of the file and the version of the HDF5 library that MATLAB is using:

```
fileinfo =
    Filename: 'example.h5'
  LibVersion: '1.8.3'
      Offset: 0
    FileSize: 35072
 GroupHierarchy: [1x1 struct]
```

In the information returned by `hdf5info`, look at the `GroupHierarchy` field. This field is a structure that describes the top-level group in the file, called the *root* group. Using the UNIX convention, HDF5 names this top-level group / (forward slash), as shown in the `Name` field of the `GroupHierarchy` structure.

```
toplevel = fileinfo.GroupHierarchy
```

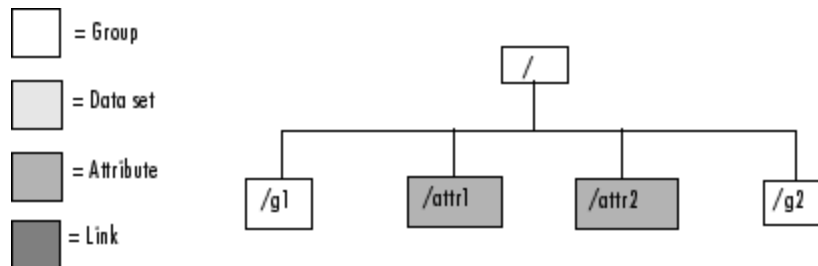
```

toplevel =

    Filename: 'matlabroot\example.h5'
      Name: '/'
     Groups: [1x4 struct]
   Datasets: []
  Datatypes: []
     Links: []
  Attributes: [1x2 struct]
    
```

By looking at the `Groups` and `Attributes` fields, you can see that the file contains two groups and two attributes. The `Datasets`, `Datatypes`, and `Links` fields are all empty, indicating that the root group does not contain any data sets, data types, or links.

The following figure illustrates the organization of the root group in the sample HDF5 file `example.h5`.



Organization of the Root Group of the Sample HDF5 File

To explore the contents of the sample HDF5 file further, examine one of the two structures in the `Groups` field of the `GroupHierarchy` structure. Each structure in this field represents a group contained in the root group. The following example shows the contents of the second structure in this field.

```

level2 = topLevel.Groups(2)

level2 =

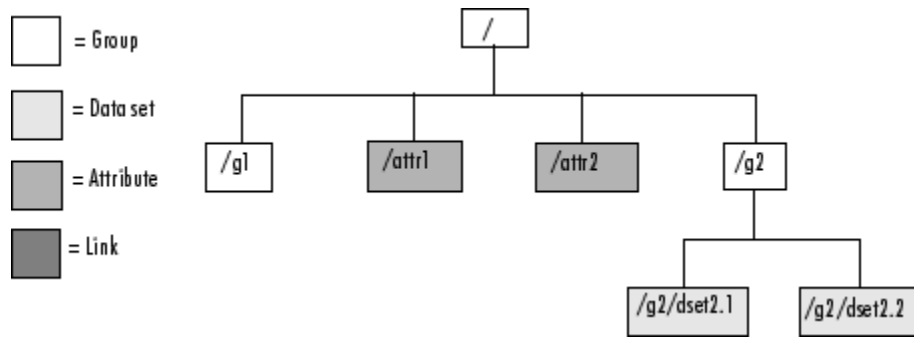
    Filename: 'matlabroot\example.h5'
      Name: '/g2'
    
```

```

Groups: []
Datasets: [1x2 struct]
Datatypes: []
Links: []
Attributes: []

```

In the sample file, the group named `/g2` contains two data sets. The following figure illustrates this part of the sample HDF5 file organization.



Organization of the Data Set `/g2` in the Sample HDF5 File

To get information about a data set, look at either of the structures returned in the `Datasets` field. These structures provide information about the data set, such as its name, dimensions, and data type.

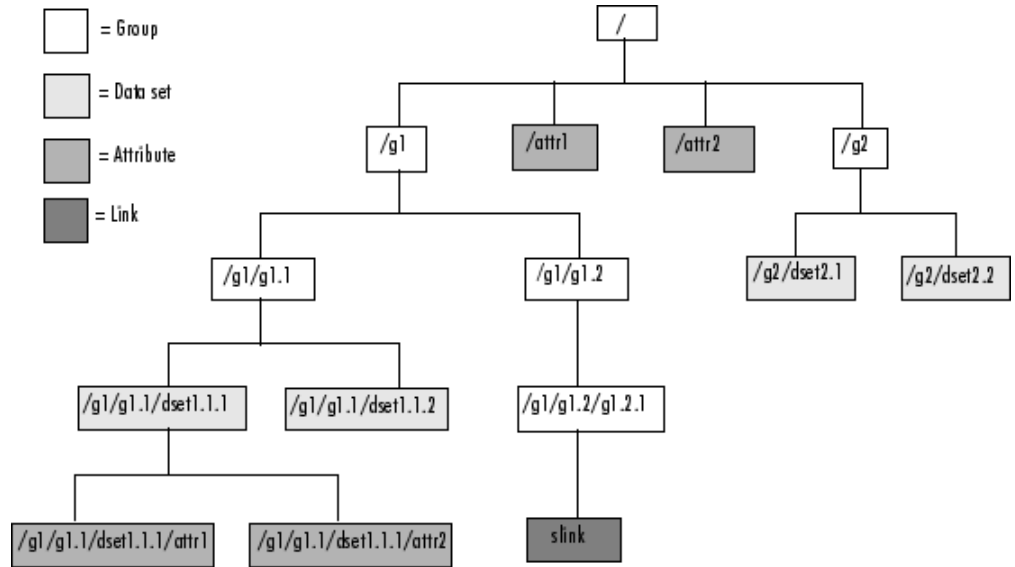
```

dataset1 = level2.Datasets(1)

dataset1 =
    Filename: 'matlabroot\example.h5'
    Name: '/g2/dset2.1'
    Rank: 1
    Datatype: [1x1 struct]
    Dims: 10
    MaxDims: 10
    Layout: 'contiguous'
    Attributes: []
    Links: []
    Chunksize: []
    Fillvalue: []

```

By examining the structures at each level of the hierarchy, you can traverse the entire file. The following figure describes the complete hierarchical organization of the sample file `example.h5`.



Hierarchical Structure of `example.h5` HDF5 File

Importing Data from an HDF5 File

To read data or metadata from an HDF5 file, use the `hdf5read` function. As arguments, you must specify the name of the HDF5 file and the name of the data set or attribute. Alternatively, you can specify just the field in the structure returned by `hdf5info` that contains the name of the data set or attribute; `hdf5read` can determine the file name from the `Filename` field in the structure. For more information about finding the name of a data set or attribute in an HDF5 file, see “Determining the Contents of an HDF5 File” on page 1-59.

To illustrate, this example reads the data set, `/g2/dset2.1` from the HDF5 sample file `example.h5`.

```
data = hdf5read('example.h5', '/g2/dset2.1');
```

The return value contains the values in the data set, in this case a 1-by-10 vector of single-precision values:

```
data =  
  
    1.0000  
    1.1000  
    1.2000  
    1.3000  
    1.4000  
    1.5000  
    1.6000  
    1.7000  
    1.8000  
    1.9000
```

The `hdf5read` function maps HDF5 data types to appropriate MATLAB data types, whenever possible. If the HDF5 file contains data types that cannot be represented in MATLAB, `hdf5read` uses one of the predefined MATLAB HDF5 data type objects (`hdf5.h5array`) to represent the data. For more information about the MATLAB HDF5 data type objects, see “Mapping HDF5 Data Types to MATLAB Data Types” on page 1-63.

Mapping HDF5 Data Types to MATLAB Data Types

When the `hdf5read` function reads data from an HDF5 file into the MATLAB workspace, it maps HDF5 data types to MATLAB data types, depending on whether the data in the data set is in an *atomic* data type or a nonatomic *composite* data type.

Atomic data types describe commonly used binary formats for numbers (integers and floating point) and characters (ASCII). Since MATLAB and HDF5 support similar data types, mapping atomic data types is typically straightforward.

Composite data types are aggregations of one or more atomic data types. Composite data types include structures, multidimensional arrays, and variable-length data types (one-dimensional arrays). The mapping is sometimes ambiguous between MATLAB classes and HDF5 data types. For example, in HDF5, a 5-by-5 data set containing a single `uint8` value in each element is distinct from a 1-by-1 data set containing a 5-by-5 array of `uint8`

values. In the first case, the data set contains 25 observations of a single value; in the second case, the data set contains a single observation with 25 values. In MATLAB both of these data sets are represented by a 5-by-5 matrix.

Mapping Atomic Data Types. HDF5 and MATLAB support similar atomic data types, mapped by `hdf5read` as shown in the table below.

Mapping Between HDF5 Atomic Data Types and MATLAB Data Types

HDF5 Atomic Data Type	MATLAB Data Type
Bit-field	Array of packed 8-bit integers
Float	MATLAB single and double types, provided that they occupy 64 bits or fewer
Integer types, signed and unsigned	Equivalent MATLAB integer types, signed and unsigned
Opaque	Array of <code>uint8</code> values
Reference	Array of <code>uint8</code> values
String	MATLAB character arrays

To find information about the data types in HDF5 files, use the `hdf5info` function. Because different computing architectures and programming languages support different number and character representations, the HDF5 library provides platform-independent data types, which it then maps to an appropriate data type for each platform.

For example, the data set `/g2/dset2.2` in the sample file `example.h5` includes atomic data. The data type information is in a `Datatype` field:

```
fileinfo = hdf5info('example.h5');
dataset1 = fileinfo.GroupHierarchy.Groups(1,2).Datasets(1,2);

dtype = dataset1.Datatype
dtype =
```

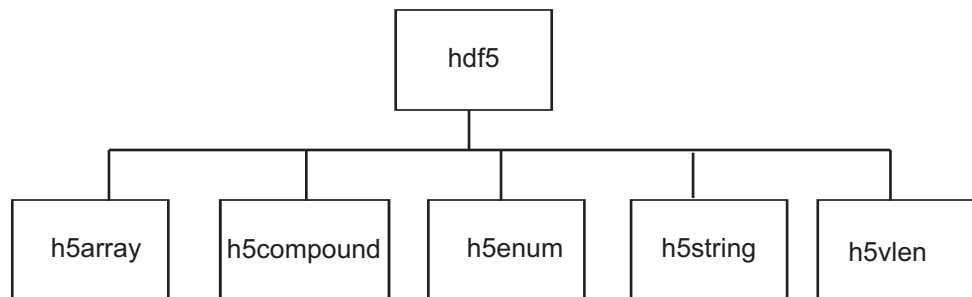


```
Name: []
Class: 'H5T_IEEE_F32BE'
Elements: []
```

The `H5T_IEEE_F32BE` class name indicates the data is a 4-byte, big endian, IEEE® floating-point data type. When `hdf5read` reads this data, MATLAB maps it to class `single`.

Mapping Composite Data Types. A composite data type is an aggregation of one or more atomic data types. Composite data types include structures, multidimensional arrays, and variable-length data types (one-dimensional arrays).

To support reading HDF5 composite data types, or writing data to an HDF5 file, MATLAB includes a set of classes to represent HDF5 data types. If the data in the data set is stored in one of the HDF5 nonatomic data types and the data cannot be represented in the workspace using a native MATLAB data type, `hdf5read` uses one of a set of classes MATLAB defines to represent HDF5 data types. The following figure illustrates the `hdf5` class and its subclasses.



To access the data in the data set in the MATLAB workspace, you must access the `Data` field in the object.

For example, if an HDF5 file contains a data set made up of an enumerated data type which cannot be represented in MATLAB, `hdf5read` uses the HDF5 `h5enum` class to represent the data. An `h5enum` object has data members that store the enumerations (text strings), their corresponding values, and the enumerated data.

This example converts a simple MATLAB vector into an `h5array` object and then displays the fields in the object:

```
vec = [ 1 2 3];  
  
hhh = hdf5.h5array(vec);  
  
hhh:  
  
    Name: ''  
    Data: [1 2 3]  
  
hhh.Data  
  
ans =  
  
     1     2     3
```

For more information about a specific MATLAB HDF5 data class, see the sections that follow:

- `h5array`
- `h5compound`
- `h5enum`
- `h5string`
- `h5vlen`

To learn more about the HDF5 data types in general, see the HDF Web page at <http://www.hdfgroup.org>.

MATLAB HDF5 `h5array` Data Class. The `h5array` data class associates a name with an array. The following tables list the class constructors, data members, and methods.

Constructors	Description
<code>arr = hdf5.h5array</code>	Creates an h5array object.
<code>arr = hdf5.h5array(data)</code>	Creates an h5array object, where <code>data</code> specifies the value of the Data member. <code>data</code> can be numeric, a cell array, or an HDF5 data type.

Data Members	Description
Data	Multidimensional array
Name	Text string specifying name of the object

Methods	Description
<code>setData(arr, data)</code>	Sets the value of the Data member, where <code>arr</code> is an h5array object and <code>data</code> can be numeric, a cell array, or an HDF5 data type.
<code>setName(arr, name)</code>	Sets the value of the Name member, where <code>arr</code> is an h5array object and <code>name</code> is a string or cell array.

MATLAB HDF5 h5compound Data Class. The h5compound data class associates a name with a structure. You can define the field names in the structure and their values. The following tables list the class constructors, data members, and methods.

Constructors	Description
<code>C = hdf5.h5compound</code>	Creates an h5compound object.
<code>C = hdf5.h5compound(n1,n2,...)</code>	Creates an h5compound object, where <code>n1</code> , <code>n2</code> and so on are text strings that specify field names. The constructor creates a corresponding data field for every member name.

Data Members	Description
Data	Multidimensional array
Name	Text string specifying name of the object
MemberNames	Text strings specifying name of the object

Methods	Description
addMember(C, mName)	Creates a new field in the object C. mName specifies the name of the field.
setMember(C, mName, mData)	Sets the value of the Data element associated with the field specified by mName, where C is an h5compound object and mData can be numeric or an HDF5 data type.
setMemberNames(C, n1, n2, ...)	Specifies the names of fields in the structure, where C is an h5compound object and n1, n2, and so on are text strings that specify field names. The method creates a corresponding data field for every name specified.
setName(C, name)	Sets the value of the Name member, where C is an h5compound object and name is a string or cell array.

MATLAB HDF5 h5enum Data Class. The h5enum data class defines an enumerated type. You can specify the enumerations (text strings) and the values they represent. The following tables list the class constructors, data members, and methods.

Constructors	Description
E = hdf5.h5enum	Creates an h5enum object.
E = hdf5.h5enum(eNames, eVals)	Creates an h5enum object, where eNames is a cell array of strings, and eVals is vector of integers. eNames and eVals must have the same number of elements.

Data Members	Description
Data	Multidimensional array
Name	Text string specifying name of the object

Data Members	Description
EnumNames	Text string specifying the enumerations, that is, the text strings that represent values
EnumValues	Values associated with enumerations

Methods	Description
defineEnum(E, eNames, eVals)	Defines the set of enumerations with the integer values they represent where eNames is a cell array of strings, and eVals is vector of integers. eNames and eVals must have the same number of elements.
enumdata = getString(E)	Returns a cell array containing the names of the enumerations, where E is an h5enum object.
setData(E, eData)	Sets the value of the object's Data member, where E is an h5enum object and eData is a vector of integers.
setEnumNames(E, eNames)	Specifies the enumerations, where E is an h5enum object and eNames is a cell array of strings.
setEnumValues(E, eVals)	Specifies the value associated with each enumeration, where E is an h5enum object and eVals is a vector of integers.
setName(E, name)	Sets the value of the object's Name member, where E is an h5enum object and name is a string or cell array.

This example uses an HDF5 enumeration object.

- 1 Create an HDF5 enumerated object.

```
enum_obj = hdf5.h5enum;
```

- 2 Define the enumerated values and their corresponding names.

```
enum_obj.defineEnum({'RED' 'GREEN' 'BLUE'}, uint8([1 2 3]));
```

enum_obj now contains the definition of the enumeration that associates the names RED, GREEN, and BLUE with the numbers 1, 2, and 3.

- 3 Add enumerated data to the object.

```
enum_obj.setData(uint8([2 1 3 3 2 3 2 1]));
```

In the HDF5 file, these numeric values map to the enumerated values GREEN, RED, BLUE, BLUE, GREEN, etc.

- 4 Write the enumerated data to a data set named objects in an HDF5 file.

```
hdf5write('myfile3.h5', '/g1/objects', enum_obj);
```

- 5 Read the enumerated data set from the file.

```
ddd = hdf5read('myfile3.h5', '/g1/objects')
```

```
hdf5.h5enum:
```

```
    Name: ''
    Data: [8x1 int8]
    EnumNames: {'RED' 'GREEN' 'BLUE'}
    EnumValues: [1 2 3]
```

MATLAB HDF5 h5string Data Class. The h5string data class associates a name with a text string and provides optional padding behavior. The following tables list the class constructors, data members, and methods.

Constructors	Description
<code>str = hdf5.h5string</code>	Creates an h5string object.
<code>str = hdf5.h5string(data)</code>	Creates an h5string object, where data is a text string.
<code>str = hdf5.h5string(data, padtype)</code>	Creates an h5stringobject, where data is a text string and padtype specifies the type of padding to use.

Data Members	Description
Data	Multidimensional array
Name	Text string specifying name of the object

Data Members	Description
Length	Scalar defining length of string
Padding	Type of padding to use: 'spacepad' 'nullterm' 'nullpad'

Methods	Description
setData(str, data)	Sets the value of the object's Data member, where str is an h5string object and data is a text string.
setLength(str, lenVal)	Sets the value of the object's Length member, where str is an h5string object and lenVal is a scalar.
setName(str, name)	Sets the value of the object's Name member, where str is an h5string object and name is a string or cell array.
setPadding(str, padType)	Specifies the value of the object's Padding member, where str is an h5string object and padType is a text string specifying one of the supported pad types.

This example uses an HDF5 string object.

- 1 Create an HDF5 string object, specifying the text string you want it to contain.

```
myH5str = hdf5.h5string('this is a string')
```

```
hdf5.h5string:
```

```
    Name: ''
    Length: 16
    Padding: 'nullterm'
    Data: 'this is a string'
```

- 2 See how the generated object is of class hdf5.h5string in the workspace.

```
whos
```

Name	Size	Bytes	Class	Attributes
myH5str	1x1		hdf5.h5string	

- 3 Set the name of the object, using a HDF5 string object method, and view the object again.

```
setName( myH5str, 'my H5 string object')
```

```
myH5str
```

```
hdf5.h5string:
```

```
    Name: 'my H5 string object'
    Length: 16
    Padding: 'nullterm'
    Data: 'this is a string'
```

MATLAB HDF5 h5vlen Data Class. The `h5vlen` data class creates a variable-length array, that is, an array in which the elements can have different lengths. This is also called a ragged array. The following tables list the class constructors, data members, and methods.

Constructors	Description
<code>V = hdf5.h5vlen</code>	Creates an <code>h5vlen</code> object.
<code>V = hdf5.h5vlen(data)</code>	Creates an <code>h5vlen</code> object, where <code>data</code> specifies the value of the <code>Data</code> member. <code>data</code> can be numeric or an HDF5 data type.

Data Members	Description
<code>Data</code>	Multidimensional array
<code>Name</code>	Text string specifying name of the object

Methods	Description
<code>setData(V, data)</code>	Sets the value of the object's <code>Data</code> member, where <code>V</code> is an <code>h5v1en</code> object and <code>data</code> can be a scalar, vector, text string, a cell array, or an HDF5 data type.
<code>setName(V, name)</code>	Sets the value of the object's <code>Name</code> member, where <code>V</code> is an <code>h5v1en</code> object and <code>name</code> is a string or cell array.

The following example creates an array of HDF5 `h5v1en` objects. The `h5v1en` objects contain numeric vectors of various lengths.

```
v(1) = hdf5.h5v1en([1:5]);
v(2) = hdf5.h5v1en([7:-1:3]);
v(3) = hdf5.h5v1en([1:2:8]);
```

Importing Hierarchical Data Format (HDF4) Files

Hierarchical Data Format (HDF4) is a general-purpose, machine-independent standard for storing scientific data in files, developed by the National Center for Supercomputing Applications (NCSA). For more information about these file formats, read the HDF documentation at the HDF Web site (www.hdfgroup.org).

HDF-EOS is an extension of HDF4 that was developed by the National Aeronautics and Space Administration (NASA) for storage of data returned from the Earth Observing System (EOS). For more information about this extension to HDF4, see the HDF-EOS documentation at the NASA Web site (www.hdfeos.org).

MATLAB includes several options for importing HDF4 files, discussed in the following sections:

- “Using the HDF Import Tool” on page 1-74
- “Using the HDF Import Tool Subsetting Options” on page 1-78
- “Using the MATLAB HDF4 High-Level Functions” on page 1-91
- “Using the HDF4 Low-Level Functions” on page 1-95

Note For information about importing HDF5 data, which is a separate, incompatible format, see “Importing Hierarchical Data Format (HDF5) Files” on page 1-58.

Using the HDF Import Tool

The HDF Import Tool is a graphical user interface that you can use to navigate through HDF4 or HDF-EOS files and import data from them. Importing data using the HDF Import Tool involves these steps:

- “Step 1: Opening an HDF4 File in the HDF Import Tool” on page 1-74
- “Step 2: Selecting a Data Set in an HDF File” on page 1-76
- “Step 3: Specifying a Subset of the Data (Optional)” on page 1-77
- “Step 4: Importing Data and Metadata” on page 1-77
- “Step 5: Closing HDF Files and the HDF Import Tool” on page 1-78

The following sections provide more detail about each of these steps.

Step 1: Opening an HDF4 File in the HDF Import Tool. Open an HDF4 or HDF-EOS file in MATLAB using one of the following methods:

- Choose the **Import Data** option from the MATLAB **File** menu. If you select an HDF4 or HDF-EOS file, the MATLAB Import Wizard automatically starts the HDF Import Tool.
- Start the HDF Import Tool by entering the `hdfstool` command at the MATLAB command line:

```
hdfstool
```

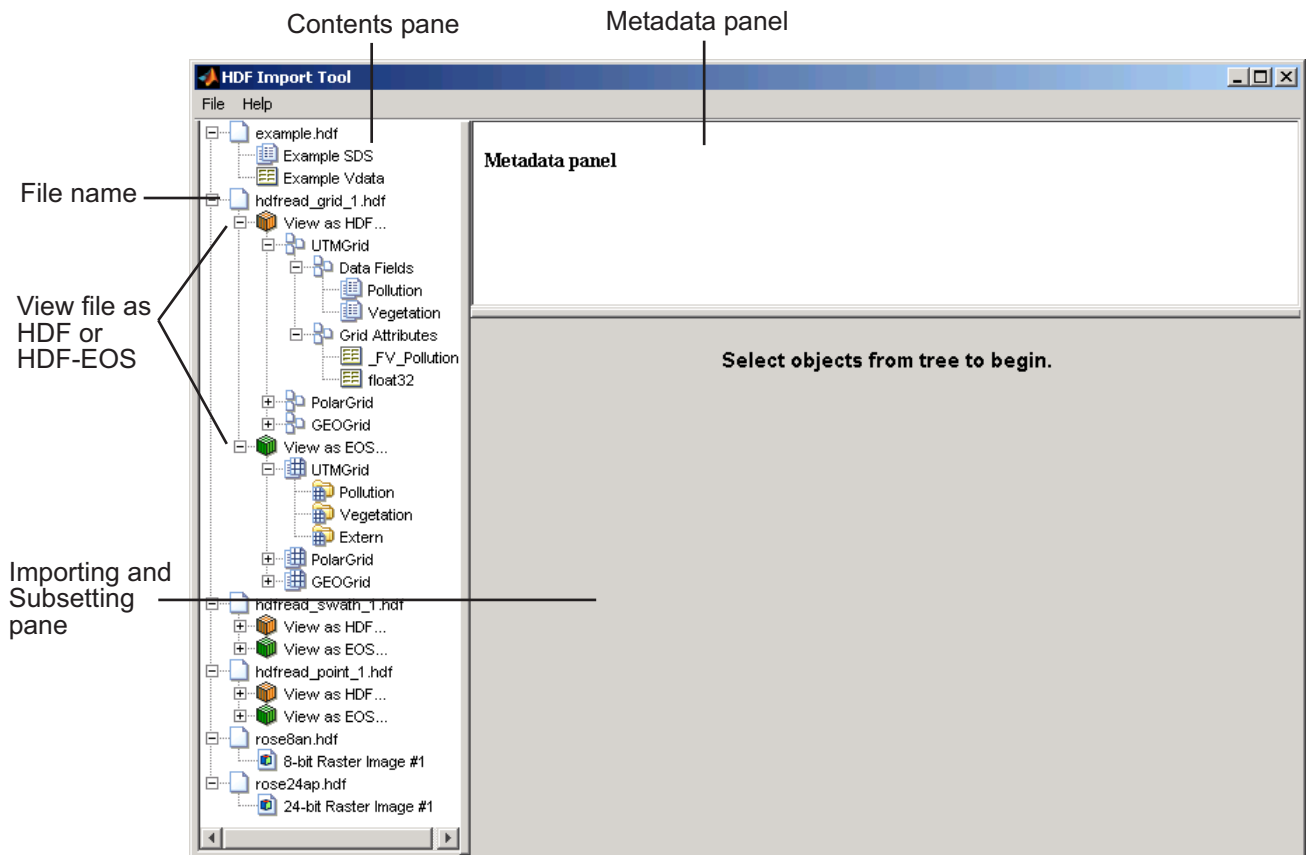
This opens an empty HDF Import Tool. To open a file, click the **Open** option on the HDFTool **File** menu and select the file you want to open. You can open multiple files in the HDF Import Tool.

- Open an HDF or HDF-EOS file by specifying the file name with the `hdfstool` command on the MATLAB command line:

```
hdfstool('example.hdf')
```

Viewing a File in the HDF Import Tool

When you open an HDF4 or HDF-EOS file in the HDF Import Tool, the tool displays the contents of the file in the Contents pane. You can use this pane to navigate within the file to see what data sets it contains. You can view the contents of HDF-EOS files as HDF data sets or as HDF-EOS files. The icon in the contents pane indicates the view, as illustrated in the following figure. Note that these are just two views of the same data.



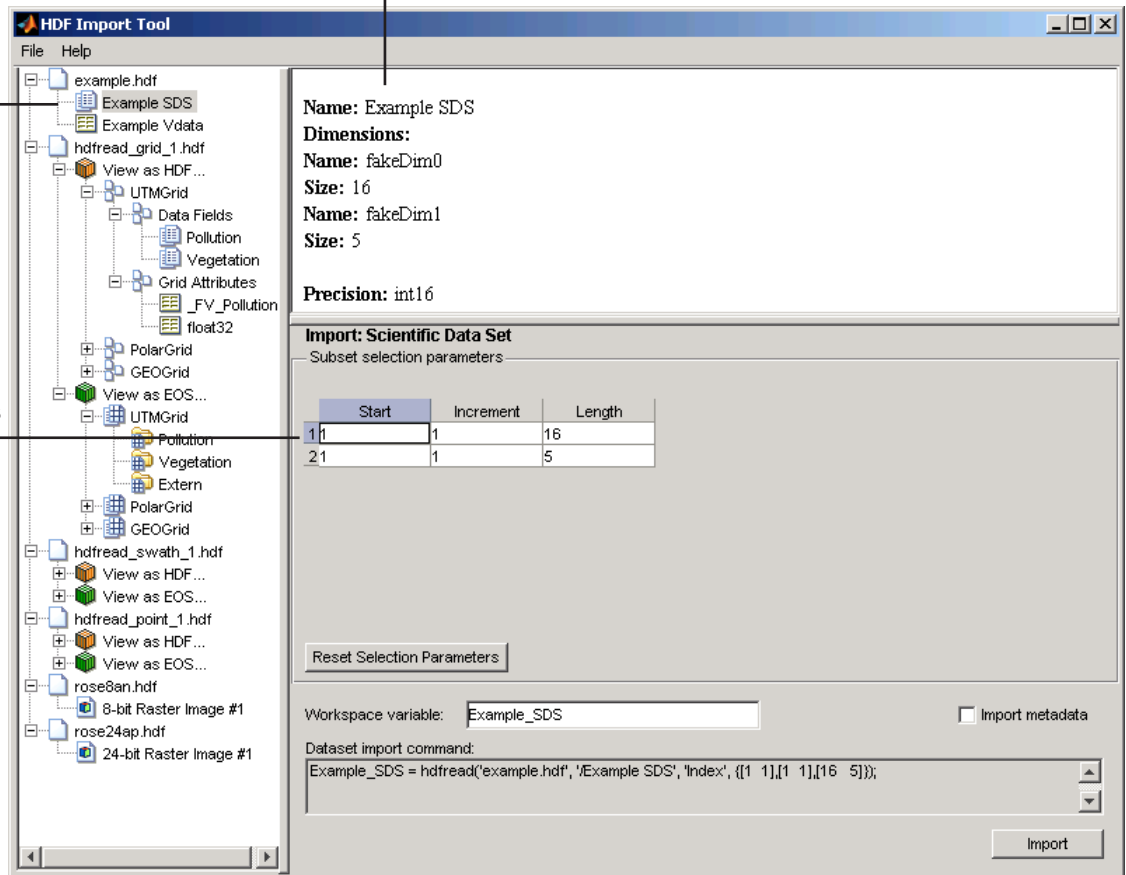
Step 2: Selecting a Data Set in an HDF File. To import a data set, you must first select the data set in the contents pane of the HDF Import Tool. Use the Contents pane to view the contents of the file and navigate to the data set you want to import.

For example, the following figure shows the data set `Example SDS` in the HDF file selected. Once you select a data set, the Metadata panel displays information about the data set and the importing and subsetting pane displays subsetting options available for this type of HDF object.

Data set metadata

Selected data set

Subsetting options for this HDF object

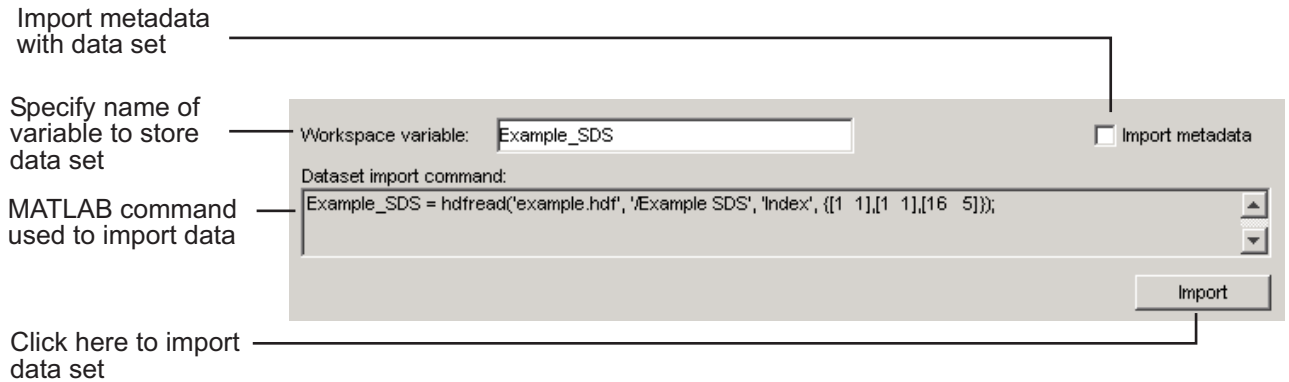


Step 3: Specifying a Subset of the Data (Optional). When you select a data set in the contents pane, the importing and subsetting pane displays the subsetting options available for that type of HDF object. The subsetting options displayed vary depending on the type of HDF object. For more information, see “Using the HDF Import Tool Subsetting Options” on page 1-78.

Step 4: Importing Data and Metadata. To import the data set you have selected, click the **Import** button, bottom right corner of the Importing and Subsetting pane. Using the Importing and Subsetting pane, you can

- Specify the name of the workspace variable — By default, the HDF Import Tool uses the name of the HDF4 data set as the name of the MATLAB workspace variable. In the following figure, the variable name is `Example_SDS`. To specify another name, enter text in the **Workspace Variable** text box.
- Specify whether to import metadata associated with the data set — To import any metadata that might be associated with the data set, select the **Import Metadata** check box. To store the metadata, the HDF Import Tool creates a second variable in the workspace with the same name with “_info” appended to it. For example, if you select this check box, the name of the metadata variable for the data set in the figure would be `Example_SDS_info`.
- Save the data set import command syntax — The **Dataset import command** text window displays the MATLAB command used to import the data set. This text is not editable, but you can copy and paste it into the MATLAB Command Window or a text editor for reuse.

The following figure shows how to specify these options in the HDF Import Tool.



Step 5: Closing HDF Files and the HDF Import Tool. To close a file, select the file in the contents pane and click **Close File** on the HDF Import Tool **File** menu.

To close all the files open in the HDF Import Tool, click **Close All Files** on the HDF Import Tool **File** menu.

To close the tool, click **Close HDFTool** in the HDF Import Tool **File** menu or click the **Close** button in the upper right corner of the tool.

If you used the `hdfstool` syntax that returns a handle to the tool,

```
h = hdfstool('example.hdf')
```

you can use the `close(h)` command to close the tool from the MATLAB command line.

Using the HDF Import Tool Subsetting Options

When you select a data set, the importing and subsetting pane displays the subsetting options available for that type of data set. The following sections provide information about these subsetting options for all supported data set types. For general information about the HDF Import tool, see “Using the HDF Import Tool” on page 1-74.

- “HDF Scientific Data Sets (SD)” on page 1-79
- “HDF Vdata” on page 1-80

- “HDF-EOS Grid Data” on page 1-81
- “HDF-EOS Point Data” on page 1-86
- “HDF-EOS Swath Data” on page 1-87
- “HDF Raster Image Data” on page 1-91

Note To use these data subsetting options effectively, you must understand the HDF and HDF-EOS data formats. Therefore, use this documentation in conjunction with the HDF documentation (www.hdfgroup.org) and the HDF-EOS documentation (www.hdfeos.org).

HDF Scientific Data Sets (SD). The HDF scientific data set (SD) is a group of data structures used to store and describe multidimensional arrays of scientific data. Using the HDF Import Tool subsetting parameters, you can import a subset of an HDF scientific data set by specifying the location, range, and number of values to be read along each dimension.

Subset selection parameters

Dimension	Start	Increment	Length
1	1	1	16
2	21	1	5

Reset Selection Parameters

The subsetting parameters are:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each

dimension. The values specified must not exceed the size of the relevant dimension of the data set.

- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.
- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

HDF Vdata. HDF Vdata data sets provide a framework for storing customized tables. A Vdata table consists of a collection of records whose values are stored in fixed-length fields. All records have the same structure and all values in each field have the same data type. Each field is identified by a name. The following figure illustrates a Vdata table.

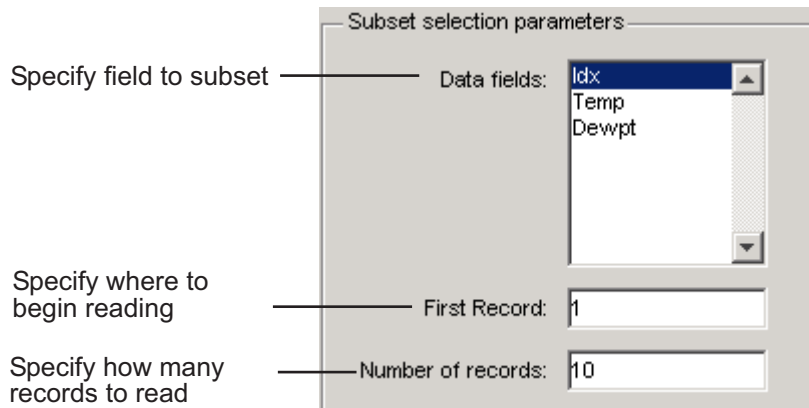
Fieldnames	idx	Temp	Dewpt
Records	1	0	5
	2	12	5
	3	3	7

Fields

You can import a subset of an HDF Vdata data set in the following ways:

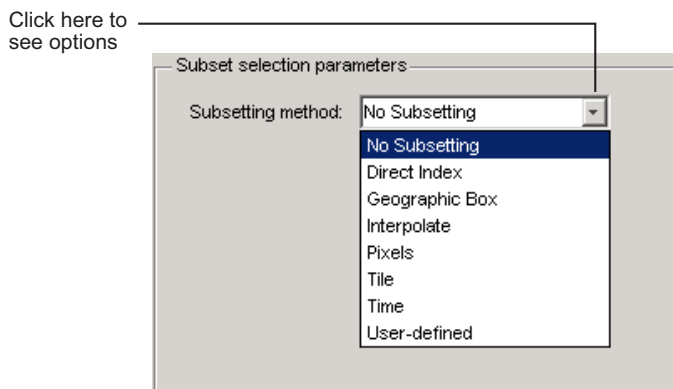
- Specifying the name of the field that you want to import
- Specifying the range of records that you want to import

The following figure shows how you specify these subsetting parameters for Vdata.



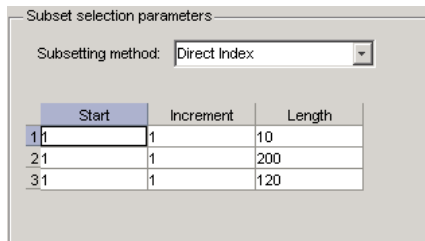
HDF-EOS Grid Data. In HDF-EOS Grid data, a rectilinear grid overlays a map. The map uses a known map projection. The HDF Import Tool supports the following mutually exclusive subsetting options for Grid data:

To access these options, click the Subsetting method menu in the importing and subsetting pane.



Direct Index

You can import a subset of an HDF-EOS Grid data set by specifying the location, range, and number of values to be read along each dimension.



Each row represents a dimension in the data set and each column represents these subsetting parameters:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of the relevant dimension of the data set.
- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.
- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

Geographic Box

You can import a subset of an HDF-EOS Grid data set by specifying the rectangular area of the grid that you are interested in. To define this rectangular area, you must specify two points, using longitude and latitude in decimal degrees. These points are two corners of the rectangular area. Typically, **Corner 1** is the upper-left corner of the box, and **Corner 2** is the lower-right corner of the box.

Subset selection parameters

Subsetting method: Geographic Box

Corner 1
Longitude: 0 Latitude: 0

Corner 2
Longitude: 0 Latitude: 0

Time (optional)
Start: Stop:

User-defined (optional)

Dimension or Field Name:	Min:	Max:
DIM: Time		
DIM: Time		
DIM: Time		

Optionally, you can further define the subset of data you are interested in by using Time parameters (see Time on page 85) or by specifying other User-Defined subsetting parameters (see User-Defined on page 86).

Interpolation

Interpolation is the process of estimating a pixel value at a location in between other pixels. In interpolation, the value of a particular pixel is determined by computing the weighted average of some set of pixels in the vicinity of the pixel.

You define the region used for bilinear interpolation by specifying two points that are corners of the interpolation area:

- **Corner 1** – Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box

Subset selection parameters

Subsetting method: Interpolate

Corner 1
Longitude: 0 Latitude: 0

Corner 2
Longitude: 0 Latitude: 0

Detailed description: This is a screenshot of a software dialog box titled "Subset selection parameters". At the top, there is a dropdown menu labeled "Subsetting method:" with "Interpolate" selected. Below this, there are two sections for defining corners. "Corner 1" has two input fields for "Longitude:" and "Latitude:", both containing the number "0". "Corner 2" also has two input fields for "Longitude:" and "Latitude:", both containing the number "0".

Pixels

You can import a subset of the pixels in a Grid data set by defining a rectangular area over the grid. You define the region used for bilinear interpolation by specifying two points that are corners of the interpolation area:

- **Corner 1** – Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box

Subset selection parameters

Subsetting method: Pixels

Corner 1
Longitude: 0 Latitude: 0

Corner 2
Longitude: 0 Latitude: 0

Detailed description: This is a screenshot of the same "Subset selection parameters" dialog box. The "Subsetting method:" dropdown menu now shows "Pixels" selected. The "Corner 1" and "Corner 2" sections remain identical, with "Longitude:" and "Latitude:" input fields both containing "0".

Tile

In HDF-EOS Grid data, a rectilinear grid overlays a map. Each rectangle defined by the horizontal and vertical lines of the grid is referred to as a *tile*. If the HDF-EOS Grid data is stored as tiles, you can import a subset of the data by specifying the coordinates of the tile you are interested in. Tile coordinates are 1-based, with the upper-left corner of a two-dimensional data set identified as 1, 1. In a three-dimensional data set, this tile would be referenced as 1, 1, 1.

Subset selection parameters

Subsetting method: Tile

Tile Coordinates: 1,1

Time

You can import a subset of the Grid data set by specifying a time period. You must specify both the start time and the stop time (the endpoint of the time span). The units (hours, minutes, seconds) used to specify the time are defined by the data set.

Subset selection parameters

Subsetting method: Time

Time

Start: 0 Stop: 0

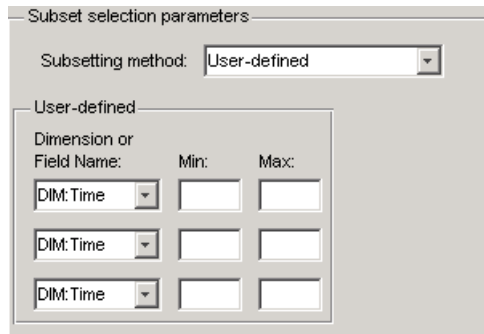
User-defined (optional)

Dimension or Field Name:	Min:	Max:
DIM: Time		
DIM: Time		
DIM: Time		

Along with these time parameters, you can optionally further define the subset of data to import by supplying user-defined parameters.

User-Defined

You can import a subset of the Grid data set by specifying user-defined subsetting parameters.

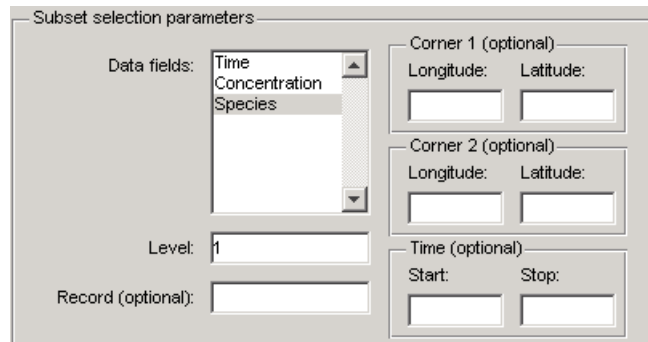


Dimension or Field Name:	Min:	Max:
DIM:Time		
DIM:Time		
DIM:Time		

When specifying user-defined parameters, you must first specify whether you are subsetting along a dimension or by field. Select the dimension or field by name using the **Dimension or Field Name** menu. Dimension names are prefixed with the characters DIM:.

Once you specify the dimension or field, you use **Min** and **Max** to specify the range of values that you want to import. For dimensions, **Min** and **Max** represent a range of *elements*. For fields, **Min** and **Max** represent a range of *values*.

HDF-EOS Point Data. HDF-EOS Point data sets are tables. You can import a subset of an HDF-EOS Point data set by specifying field names and level. Optionally, you can refine the subsetting by specifying the range of records you want to import, by defining a rectangular area, or by specifying a time period. For information about specifying a rectangular area, see Geographic Box on page 82. For information about subsetting by time, see Time on page 85.

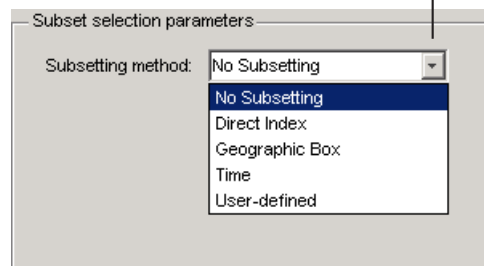


HDF-EOS Swath Data. HDF-EOS Swath data is data that is produced by a satellite as it traces a path over the earth. This path is called its ground track. The sensor aboard the satellite takes a series of scans perpendicular to the ground track. Swath data can also include a vertical measure as a third dimension. For example, this vertical dimension can represent the height above the Earth of the sensor.

The HDF Import Tool supports the following mutually exclusive subsetting options for Swath data:

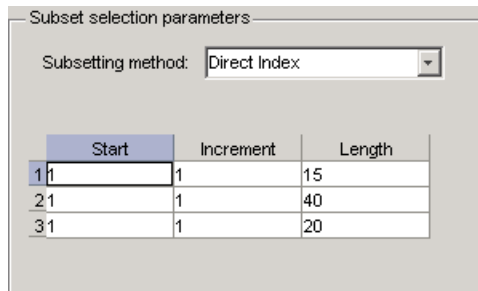
To access these options, click the **Subsetting method** menu in the **Importing and Subsetting** pane.

Click here to select a subsetting option



Direct Index

You can import a subset of an HDF-EOS Swath data set by specifying the location, range, and number of values to be read along each dimension.



	Start	Increment	Length
1	1	1	15
2	1	1	40
3	1	1	20

Each row represents a dimension in the data set and each column represents these subsetting parameters:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of the relevant dimension of the data set.
- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.
- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

Geographic Box

You can import a subset of an HDF-EOS Swath data set by specifying the rectangular area of the grid that you are interested in and by specifying the selection Mode.

You define the rectangular area by specifying two points that specify two corners of the box:

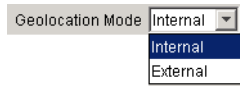
- **Corner 1** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box.

You specify the selection mode by choosing the type of **Cross Track Inclusion** and the **Geolocation mode**. The **Cross Track Inclusion** value determines how much of the area of the geographic box that you define must fall within the boundaries of the swath.

Select from these values:

- **AnyPoint** — Any part of the box overlaps with the swath.
- **Midpoint** — At least half of the box overlaps with the swath.
- **Endpoint** — All of the area defined by the box overlaps with the swath.

The **Geolocation Mode** value specifies whether geolocation fields and data must be in the same swath.

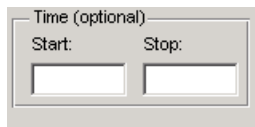


Select from these values:

- **Internal** — Geolocation fields and data fields must be in the same swath.
- **External** — Geolocation fields and data fields can be in different swaths.

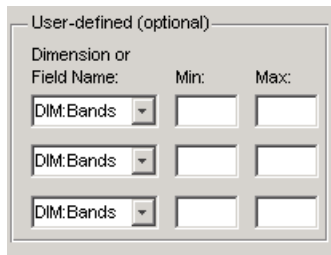
Time

You can optionally also subset swath data by specifying a time period. The units used (hours, minutes, seconds) to specify the time are defined by the data set



User-Defined

You can optionally also subset a swath data set by specifying user-defined parameters.



When specifying user-defined parameters, you must first specify whether you are subsetting along a dimension or by field. Select the dimension or field by

name using the **Dimension or Field Name** menu. Dimension names are prefixed with the characters **DIM:**.

Once you specify the dimension or field, you use **Min** and **Max** to specify the range of values that you want to import. For dimensions, **Min** and **Max** represent a range of *elements*. For fields, **Min** and **Max** represent a range of *values*.

HDF Raster Image Data. For 8-bit HDF raster image data, you can specify the colormap.

Using the MATLAB HDF4 High-Level Functions

To import data from an HDF or HDF-EOS file, you can use the MATLAB HDF4 high-level function `hdfread`. The `hdfread` function provides a programmatic way to import data from an HDF4 or HDF-EOS file that still hides many of the details that you need to know if you use the low-level HDF functions, described in “Using the HDF4 Low-Level Functions” on page 1-95. You can also import HDF4 data using an interactive GUI, described in “Using the HDF Import Tool” on page 1-74.

This section describes these high-level MATLAB HDF functions, including

- “Using `hdfinfo` to Get Information About an HDF4 File” on page 1-91
- “Using `hdfread` to Import Data from an HDF4 File” on page 1-92

To export data to an HDF4 file, you must use the MATLAB HDF4 low-level functions.

Using `hdfinfo` to Get Information About an HDF4 File. To get information about the contents of an HDF4 file, use the `hdfinfo` function. The `hdfinfo` function returns a structure that contains information about the file and the data in the file.

Note You can also use the HDF Import Tool to get information about the contents of an HDF4 file. See “Using the HDF Import Tool” on page 1-74 for more information.

This example returns information about a sample HDF4 file included with MATLAB:

```
info = hdfinfo('example.hdf')

info =

    Filename: 'matlabroot\example.hdf'
  Attributes: [1x2 struct]
     Vgroup: [1x1 struct]
        SDS: [1x1 struct]
     Vdata: [1x1 struct]
```

To get information about the data sets stored in the file, look at the SDS field.

Using `hdfread` to Import Data from an HDF4 File. To use the `hdfread` function, you must specify the data set that you want to read. You can specify the filename and the data set name as arguments, or you can specify a structure returned by the `hdfinfo` function that contains this information. The following example shows both methods. For information about how to import a subset of the data in a data set, see [Reading a Subset of the Data in a Data Set](#) on page 94.

- 1** Determine the names of data sets in the HDF4 file, using the `hdfinfo` function.

```
info = hdfinfo('example.hdf')

info =

    Filename: 'matlabroot\example.hdf'
  Attributes: [1x2 struct]
     Vgroup: [1x1 struct]
        SDS: [1x1 struct]
     Vdata: [1x1 struct]
```

To determine the names and other information about the data sets in the file, look at the contents of the SDS field. The `Name` field in the SDS structure gives the name of the data set.

```
dsets = info.SDS
```

```
dsets =  
  
    Filename: 'example.hdf'  
    Type: 'Scientific Data Set'  
    Name: 'Example SDS'  
    Rank: 2  
    DataType: 'int16'  
    Attributes: []  
    Dims: [2x1 struct]  
    Label: {}  
    Description: {}  
    Index: 0
```

- 2** Read the data set from the HDF4 file, using the `hdfread` function. Specify the name of the data set as a parameter to the function. Note that the data set name is case sensitive. This example returns a 16-by-5 array:

```
dset = hdfread('example.hdf', 'Example SDS')
```

```
dset =  
  
    3     4     5     6     7  
    4     5     6     7     8  
    5     6     7     8     9  
    6     7     8     9    10  
    7     8     9    10    11  
    8     9    10    11    12  
    9    10    11    12    13  
   10    11    12    13    14  
   11    12    13    14    15  
   12    13    14    15    16  
   13    14    15    16    17  
   14    15    16    17    18  
   15    16    17    18    19  
   16    17    18    19    20  
   17    18    19    20    21  
   18    19    20    21    22
```

Alternatively, you can specify the specific field in the structure returned by `hdfinfo` that contains this information. For example, to read a scientific data set, use the `SDS` field.

```
dset = hdfread(info.SDS);
```

Reading a Subset of the Data in a Data Set

To read a subset of a data set, you can use the optional `'index'` parameter. The value of the index parameter is a cell array of three vectors that specify the location in the data set to start reading, the skip interval (e.g., read every other data item), and the amount of data to read (e.g., the length along each dimension). In HDF4 terminology, these parameters are called the *start*, *stride*, and *edge* values.

For example, this code

- Starts reading data at the third row, third column ([3 3]).
- Reads every element in the array ([]).
- Reads 10 rows and 2 columns ([10 2]).

```
subset = hdfread('Example.hdf','Example SDS',...  
                'Index',{[3 3],[],[10 2 ]})
```

```
subset =
```

```
      7      8  
      8      9  
      9     10  
     10     11  
     11     12  
     12     13  
     13     14  
     14     15  
     15     16  
     16     17
```

Using the HDF4 Low-Level Functions

This section describes how to use MATLAB functions to access the HDF4 Application Programming Interfaces (APIs). These APIs are libraries of C routines. To import or export data, you must use the functions in the HDF4 API associated with the particular HDF4 data type you are working with. Each API has a particular programming model, that is, a prescribed way to use the routines to write data sets to the file. To illustrate this concept, this section describes the programming model of one particular HDF4 API: the HDF4 Scientific Data (SD) API. For a complete list of the HDF4 APIs supported by MATLAB and the functions you use to access each one, see the `hdf` reference page.

Note This section does not attempt to describe all HDF4 features and routines. To use the MATLAB HDF4 functions effectively, you must refer to the official NCSA documentation at the HDF Web site (www.hdfgroup.org).

This section includes the following:

- “Mapping HDF4 to MATLAB Syntax” on page 1-95
- “Step 1: Opening the HDF4 File” on page 1-96
- “Step 2: Retrieving Information About the HDF4 File” on page 1-97
- “Step 3: Retrieving Attributes from an HDF4 File (Optional)” on page 1-98
- “Step 4: Selecting the Data Sets to Import” on page 1-99
- “Step 5: Getting Information About a Data Set” on page 1-99
- “Step 6: Reading Data from the HDF4 File” on page 1-100
- “Step 7: Closing the HDF4 Data Set” on page 1-101
- “Step 8: Closing the HDF4 File” on page 1-102

Mapping HDF4 to MATLAB Syntax. Each HDF4 API includes many individual routines that you use to read data from files, write data to files, and perform other related functions. For example, the HDF4 Scientific Data (SD) API includes separate C routines to open (`SDopen`), close (`SDend`), and read data (`SDreaddata`).

Instead of supporting each routine in the HDF4 APIs, MATLAB provides a single function that serves as a gateway to all the routines in a particular HDF4 API. For example, the HDF Scientific Data (SD) API includes the C routine `SDend` to close an HDF4 file:

```
status = SDend(sd_id); /* C code */
```

To call this routine from MATLAB, use the MATLAB function associated with the SD API, `hdfsd`. You must specify the name of the routine, minus the API acronym, as the first argument and pass any other required arguments to the routine in the order they are expected. For example,

```
status = hdfsd('end',sd_id); % MATLAB code
```

Some HDF4 API routines use output arguments to return data. Because MATLAB does not support output arguments, you must specify these arguments as return values.

For example, the `SDfileinfo` routine returns data about an HDF4 file in two output arguments, `ndatasets` and `nglobal_atts`. Here is the C code:

```
status = SDfileinfo(sd_id, ndatasets, nglobal_atts);
```

To call this routine from MATLAB, change the output arguments into return values:

```
[ndatasets, nglobal_atts, status] = hdfsd('fileinfo',sd_id);
```

Specify the return values in the same order as they appear as output arguments. The function status return value is always specified as the last return value.

Step 1: Opening the HDF4 File.

Note These steps, when referring to specific routines in the HDF4 SD API, use the C library name rather than the MATLAB function name. The MATLAB syntax is used in all examples.

To import an HDF4 SD data set, you must first open the file using the SD API routine `SDstart`. (In HDF4 terminology, the numeric arrays stored in HDF4 files are called data sets.) In MATLAB, you use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `start` in this case.
- Name of the file you want to open.
- Mode in which you want to open it. The following table lists the file access modes supported by the `SDstart` routine. In MATLAB, you specify these modes as text strings. You can specify the full HDF4 mode name or one of the abbreviated forms listed in the table.

HDF4 File Creation Mode	HDF4 Mode Name	MATLAB String
Create a new file	'DFACC_CREATE'	'create'
Read access	'DFACC_RDONLY'	'read' or 'rdonly'
Read and write access	'DFACC_RDWR'	'rdwr' or 'write'

For example, this code opens the file `mydata.hdf` for read access:

```
sd_id = hdfsd('start','mydata.hdf','read');
```

If `SDstart` can find and open the file specified, it returns an HDF4 SD file identifier, named `sd_id` in the example. Otherwise, it returns `-1`.

Step 2: Retrieving Information About the HDF4 File. To get information about an HDF4 file, you must use the SD API routine `SDfileinfo`. This function returns the number of data sets in the file and the number of global attributes in the file, if any. (For more information about global attributes, see “Exporting to Hierarchical Data Format (HDF4) Files” on page 2-43.) In MATLAB, you use the `hdfsd` function, specifying the following arguments:

- Name of the SD API routine, `fileinfo` in this case
- SD file identifier, `sd_id`, returned by `SDstart`

In this example, the HDF4 file contains three data sets and one global attribute.

```
[ndatasets, nglobal_atts, stat] = hdfsd('fileinfo',sd_id)

ndatasets =
    3

nglobal_atts =
    1

status =
    0
```

Step 3: Retrieving Attributes from an HDF4 File (Optional). HDF4 files can optionally include information, called *attributes*, that describes the data the file contains. Attributes associated with an entire HDF4 file are called *global* attributes. Attributes associated with a data set are called *local* attributes. (You can also associate attributes with files or dimensions. For more information, see “Step 4: Writing Metadata to an HDF4 File” on page 2-50.)

To retrieve attributes from an HDF4 file, use the HDF4 API routine `SDreadattr`. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `readattr` in this case.
- File identifier (`sd_id`) returned by `SDstart`, for global attributes, or the data set identifier for local attributes. (See “Step 4: Selecting the Data Sets to Import” on page 1-99 to learn how to get a data set identifier.)
- Index identifying the attribute you want to view. HDF4 uses zero-based indexing. If you know the name of an attribute but not its index, use the `SDfindattr` routine to determine the index value associated with the attribute.

For example, this code returns the contents of the first global attribute, which is the character string `my global attribute`:

```
attr_idx = 0;
[attr, status] = hdfsd('readattr', sd_id, attr_idx);

attr =
    my global attribute
```

Step 4: Selecting the Data Sets to Import. To select a data set, use the SD API routine `SDselect`. In MATLAB, you use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `select` in this case
- HDF4 SD file identifier (`sd_id`) returned by `SDstart`

If `SDselect` finds the specified data set in the file, it returns an HDF4 SD data set identifier, called `sds_id` in the example. If it cannot find the data set, it returns `-1`.

Note Do not confuse HDF4 SD *file* identifiers, named `sd_id` in the examples, with HDF4 SD *data set* identifiers, named `sds_id` in the examples.

```
sds_id = hdfsd('select',sd_id,1)
```

Step 5: Getting Information About a Data Set. To read a data set, you must get information about the data set, such as its name, size, and data type. In the HDF4 SD API, you use the `SDgetinfo` routine to gather this information. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `getinfo` in this case
- HDF4 SD data set identifier (`sds_id`) returned by `SDselect`

This code retrieves information about the data set identified by `sds_id`:

```
[dsname, dsndims, dsdims, dstype, dsatts, stat] =
    hdfsd('getinfo',sds_id)
dsname =
    A
dsndims =
    2
dsdims =
    5    3
```

```
dstype =  
    double  
  
dsatts =  
    0  
  
stat =  
    0
```

Step 6: Reading Data from the HDF4 File. To read data from an HDF4 file, you must use the `SDreaddata` routine. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API function, `readdata` in this case.
- HDF4 SD data set identifier (`sds_id`) returned by `SDselect`.
- Location in the data set where you want to start reading data, specified as a vector of index values, called the *start* vector. To read from the beginning of a data set, specify zero for each element of the start vector. Use `SDgetinfo` to determine the dimensions of the data set.
- Number of elements along each dimension to skip between each read operation, specified as a vector of scalar values, called the *stride* vector. To read every element of a data set, specify 1 as the value for each element of the vector or specify an empty array (`[]`).
- Total number of elements to read along each dimension, specified as a vector of scalar values, called the *edges* vector. To read every element of a data set, set each element of the edges vector to the size of each dimension of the data set. Use `SDgetinfo` to determine these sizes.

Note `SDgetinfo` returns dimension values in row-major order, the ordering used by HDF4. Because MATLAB stores data in column-major order, you must specify the dimensions in column-major order, that is, `[columns, rows]`. In addition, you must use zero-based indexing in these arguments.

For example, to read the entire contents of a data set, use this code:

```
[ds_name, ds_ndims, ds_dims, ds_type, ds_atts, stat] =
```

```

hdfsd('getinfo',sds_id);

ds_start = zeros(1,ds_ndims); % Creates the vector [0 0]
ds_stride = [];
ds_edges = ds_dims;

[ds_data, status] =
    hdfsd('readdata',sds_id,ds_start,ds_stride,ds_edges);

disp(ds_data)
     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15

```

To read less than the entire data set, use the start, stride, and edges vectors to specify where you want to start reading data and how much data you want to read. For example, this code reads the entire second row of the sample data set:

```

ds_start = [0 1]; % Start reading at the first column, second row
ds_stride = []; % Read each element
ds_edges = [5 1]; % Read a 1-by-5 vector of data

[ds_data, status] =
    hdfsd('readdata',sds_id,ds_start,ds_stride,ds_edges);

```

Step 7: Closing the HDF4 Data Set. After writing data to a data set in an HDF4 file, you must close access to the data set. In the HDF4 SD API, you use the `SDendaccess` routine to close a data set. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `endaccess` in this case
- HDF4 SD data set identifier (`sds_id`) returned by `SDselect`

For example, this code closes the data set:

```
stat = hdfsd('endaccess',sds_id);
```

You must close access to all the data sets in an HDF4 file before closing it.

Step 8: Closing the HDF4 File. After writing data to a data set and closing the data set, you must also close the HDF4 file. In the HDF4 SD API, you use the `SDend` routine. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, end in this case
- HDF4 SD file identifier (`sd_id`) returned by `SDstart`

For example, this code closes the data set:

```
stat = hdfsd('end',sd_id);
```

Importing Images

To import data into the MATLAB workspace from a graphics file, use the `imread` function. Using this function, you can import data from files in many standard file formats, including the Tagged Image File Format (TIFF), Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), and Portable Network Graphics (PNG) formats. For a complete list of supported formats, see the `imread` reference page.

This example reads the image data stored in a file in JPEG format into the MATLAB workspace as the array `I`:

```
I = imread('ngc6543a.jpg');
```

`imread` represents the image in the workspace as a multidimensional array of class `uint8`. The dimensions of the array depend on the format of the data. For example, `imread` uses three dimensions to represent RGB color images:

```
whos I
      Name      Size              Bytes  Class
      I         650x600x3          1170000  uint8 array

Grand total is 1170000 elements using 1170000 bytes
```

For more control over reading TIFF files, use the `Tiff` object—see “Reading Image Data and Metadata from TIFF Files” on page 1-104 for more information.

Getting Information about Image Files

If you have a file in a standard graphics format, use the `imfinfo` function to get information about its contents. The `imfinfo` function returns a structure containing information about the file. The fields in the structure vary with the file format, but `imfinfo` always returns some basic information including the file name, last modification date, file size, and format.

This example returns information about a file in Joint Photographic Experts Group (JPEG) format:

```
info = imfinfo('ngc6543a.jpg')

info =

    Filename: [1x57 char]
  FileModDate: '01-Oct-1996 16:19:44'
    FileSize: 27387
      Format: 'jpg'
  FormatVersion: ''
        Width: 600
        Height: 650
    BitDepth: 24
    ColorType: 'truecolor'
  FormatSignature: ''
  NumberOfSamples: 3
    CodingMethod: 'Huffman'
  CodingProcess: 'Sequential'
    Comment: {[1x69 char]}
```

Reading Image Data and Metadata from TIFF Files

While you can use `imread` to import image data and metadata to TIFF files, the function does have some limitations. For example, a TIFF file can contain multiple images and each images can have multiple subimages. While you can read all the images from a multi-image TIFF file with `imread`, you cannot access the subimages. Using the `Tiff` object, you can read image data, metadata, and subimages from a TIFF file. When you construct a `Tiff` object, it represents your connection with a TIFF file and provides access to many of the routines in the `LibTIFF` library.

The following section provides a step-by-step example of using `Tiff` object methods and properties to read subimages from a TIFF file. To get the most out of the `Tiff` object, you must be familiar with the `LibTIFF` version 3.7.1 API, as well as the TIFF specification and technical notes. View this documentation at [LibTIFF - TIFF Library and Utilities](#)

Reading Subimages from a TIFF File

A TIFF file can contain one or more image file directories (IFD). Each IFD contains image data and the metadata (tags) associated with the image. Each IFD can contain one or more subIFDs, which can also contain image data and

metadata. These subimages are typically reduced-resolution (thumbnail) versions of the image data in the IFD containing the subIFDs.

To read the subimages in an IFD, you must get the location of the subimage from the SubIFD tag. The SubIFD tag contains an array of byte offsets that point to the subimages. You can then pass the address of the subIFD to the `setSubDirectory` method to make the subIFD the current IFD. Most `Tiff` object methods operate on the current IFD.

- 1** Open a TIFF file that contains images and subimages using the `Tiff` object constructor. This example uses the TIFF file created in “Creating Subdirectories in a TIFF File” on page 2-59, which contains one IFD directory with two subIFDs. The `Tiff` constructor opens the TIFF file, and makes the first subIFD in the file the current IFD:

```
t = Tiff('my_subimage_file.tif','r');
```

- 2** Retrieve the locations of subIFDs associated with the current IFD. Use the `getTag` method to get the value of the SubIFD tag. This returns an array of byte offsets that specify the location of subIFDs:

```
offsets = t.getTag('SubIFD')
```

- 3** Navigate to the first subIFD using the `setSubDirectory` method. Specify the byte offset of the subIFD as an argument. This call makes the subIFD the current IFD:

```
t.setSubDirectory(offsets(1));
```

- 4** Read the image data from the current IFD (the first subIFD) as you would with any other IFD in the file:

```
subimage_one = t.read();
```

- 5** View the first subimage:

```
imagesc(subimage_one)
```

- 6** To view the second subimage, call the `setSubDirectory` method again, specifying the byte offset of the second subIFD:

```
t.setSubDirectory(offsets(2));
```

- 7 Read the image data from the current IFD (the second subIFD) as you would with any other IFD in the file:

```
subimage_two = t.read();
```

- 8 View the second subimage:

```
imagesc(subimage_two)
```

- 9 Close the Tiff object.

```
t.close();
```

Importing Audio and Video

In this section...

“Getting Information about Audio or Video Files” on page 1-107

“Reading Audio Files” on page 1-108

“Recording and Playing Audio” on page 1-108

“Reading Video Files” on page 1-111

Getting Information about Audio or Video Files

To get information about files that contain audio data, video data, or both, use `mmfileinfo`. The `mmfileinfo` function returns the duration, format, number of audio channels, and height and width of video, as applicable.

To get more information about files that contain only video data, such as the number of frames, create a multimedia object with `VideoReader` and use the `get` method. For more information, see “Getting Information about Video Files” on page 1-112.

Characteristics of Audio Files

The audio signal in a file represents a series of *samples* that capture the amplitude of the sound over time. The *sample rate* is the number of discrete samples taken per second and given in hertz. The precision of the samples, measured by the *bit depth* (number of bits per sample), depends on the available audio hardware.

MATLAB audio functions read and store single-channel (mono) audio data in an m -by-1 column vector, and stereo data in an m -by-2 matrix. In either case, m is the number of samples. For stereo data, the first column contains the left channel, and the second column contains the right channel.

Typically, each sample is a double-precision value between -1 and 1. In some cases, particularly when the audio hardware does not support high bit depths, audio files store the values as 8-bit or 16-bit integers. The range of the sample values depends on the available number of bits. For example, samples stored as `uint8` values can range from 0 to 255 ($2^8 - 1$). The MATLAB `sound` and `soundsc` functions support only single- or double-precision values between

-1 and 1. Other audio functions support multiple data types, as indicated on the function reference pages.

Reading Audio Files

The easiest way to read audio data from a file is to use the Import Wizard, a graphical user interface. The Import Wizard can read WAV, AU, or SND files. To start the Import Wizard, select **File > Import Data** or double-click the file name in the Current Folder browser. To import WAV files without invoking a graphical user interface, use `wavread`.

Recording and Playing Audio

This section discusses the following topics:

- “Recording Audio” on page 1-108
- “Playing Audio” on page 1-109
- “Recording or Playing Audio within a Function” on page 1-110

Recording Audio

To record data from an audio input device (such as a microphone connected to your system) for processing in MATLAB:

- 1** Create an audiorecorder object.
- 2** Call the `record` or `recordblocking` method, where:
 - `record` returns immediate control to the calling function or the command prompt even as recording proceeds. Specify the length of the recording in seconds, or end the recording with the `stop` method. Optionally, call the `pause` and `resume` methods.
 - `recordblocking` retains control until the recording is complete. Specify the length of the recording in seconds.
- 3** Create a numeric array corresponding to the signal data using the `getaudiodata` method.

For example, connect a microphone to your system and record your voice for 5 seconds. Capture the numeric signal data and create a plot:

```
% Record your voice for 5 seconds.
recObj = audiorecorder;
disp('Start speaking. ')
recordblocking(recObj, 5);
disp('End of Recording. ');

% Play back the recording.
play(recObj);

% Store data in double-precision array.
myRecording = getaudiodata(recObj);

% Plot the samples.
plot(myRecording);
```

Specifying the Quality of the Recording. By default, an `audiorecorder` object uses a sample rate of 8000 hertz, a depth of 8 bits (8 bits per sample), and a single audio channel. These settings minimize the required amount of data storage. For higher quality recordings, increase the sample rate or bit depth.

For example, typical compact disks use a sample rate of 44,100 hertz and a 16-bit depth. Create an `audiorecorder` object to record in stereo (two channels) with those settings:

```
myRecObj = audiorecorder(44100, 16, 2);
```

For more information on the available properties and values, see the `audiorecorder` reference page.

Playing Audio

After you import or record audio, MATLAB supports several ways to listen to the data:

- For simple playback using a single function call, use `sound` or `soundsc`. For example, load a demo MAT-file that contains signal and sample rate data, and listen to the audio:

```
load chirp.mat;  
sound(y, Fs);
```

- For more flexibility during playback, including the ability to pause, resume, or define callbacks, use the `audioplayer` function. Create an `audioplayer` object, then call methods to play the audio. For example, listen to the gong demo:

```
load gong.mat;  
gong = audioplayer(y, Fs);  
play(gong);
```

For an additional example, see “Recording or Playing Audio within a Function” on page 1-110.

If you do not specify the sample rate, `sound` plays back at 8192 hertz. For any playback, specify smaller sample rates to play back more slowly, and larger sample rates to play back more quickly.

Note Most sound cards support sample rates between approximately 5,000 and 48,000 hertz. Specifying sample rates outside this range can produce unexpected results.

Recording or Playing Audio within a Function

Unlike graphics handles, if you create an `audioplayer` or `audiorecorder` object inside a function, the object exists only for the duration of the function. For example, create a player function called `playFile` and a simple callback function `showSeconds`:

```
function playFile(myfile)  
    load(myfile);  
  
    obj = audioplayer(y, Fs);  
    obj.TimerFcn = 'showSeconds';  
    obj.TimerPeriod = 1;  
  
    play(obj);  
end
```

```
function showSeconds
    disp('tick')
end
```

Call `playFile` from the command prompt to play the demo file `handel.mat`:

```
playFile('handel.mat')
```

At the recorded sample rate of 8192 samples per second, playing the 73113 samples in the file takes approximately 8.9 seconds. However, the `playFile` function typically ends before playback completes, and clears the `audioplayer` object `obj`.

To ensure complete playback or recording, consider the following options:

- Use `playblocking` or `recordblocking` instead of `play` or `record`. The blocking methods retain control until playing or recording completes. If you block control, you cannot issue any other commands or methods (such as `pause` or `resume`) during the playback or recording.
- Create an output argument for your function that generates an object in the base workspace. For example, modify the `playFile` function to include an output argument:

```
function obj = playFile(myfile)
```

Call the function:

```
h = playFile('handel.mat');
```

Because `h` exists in the base workspace, you can pause playback from the command prompt:

```
pause(h)
```

Reading Video Files

To import video data from a file, construct a reader object with `VideoReader` and read selected frames with the `read` function.

For example, import all frames in the demo file `xylophone.mpg`:

```
xyloObj = VideoReader('xylophone.mpg');  
vidFrames = read(xyloObj);
```

It is not necessary to close the multimedia object.

For more information, see:

- “Getting Information about Video Files” on page 1-112
- “Processing Frames of a Video File” on page 1-113
- “Reading Variable Frame Rate Video” on page 1-114
- “Supported Video File Formats” on page 1-115

Getting Information about Video Files

`VideoReader` creates an object that contains properties of the video file, including the duration, frame rate, format, height, and width. To view these properties, or store them in a structure, use the `get` method. For example, get the properties of the demo file `xylophone.mpg`:

```
xyloObj = VideoReader('xylophone.mpg');  
info = get(xyloObj)
```

The `get` function returns:

```
info =  
    Duration: 4.7020  
    Name: 'xylophone.mpg'  
    Path: [1x75 char]  
    Tag: ''  
    Type: 'VideoReader'  
    UserData: []  
    BitsPerPixel: 24  
    FrameRate: 29.9700  
    Height: 240  
    NumberOfFrames: 141  
    VideoFormat: 'RGB24'  
    Width: 320
```


To access a specific property of the object, such as `Duration`, use dot notation as follows:

```
duration = xyloObj.Duration;
```

Note For files with a variable frame rate, `VideoReader` cannot return the number of frames until you read the last frame of the file. For more information, see “Reading Variable Frame Rate Video” on page 1-114.

Processing Frames of a Video File

A typical video contains many frames. To save memory, process a video one frame at a time. For faster processing, preallocate memory to store the video data.

For example, convert the demo file `xylophone.mpg` to a MATLAB movie, and play it with the `movie` function:

```
xyloObj = VideoReader('xylophone.mpg');

nFrames = xyloObj.NumberOfFrames;
vidHeight = xyloObj.Height;
vidWidth = xyloObj.Width;

% Preallocate movie structure.
mov(1:nFrames) = ...
    struct('cdata', zeros(vidHeight, vidWidth, 3, 'uint8'),...
          'colormap', []);

% Read one frame at a time.
for k = 1 : nFrames
    mov(k).cdata = read(xyloObj, k);
end

% Play back the movie once at the video's frame rate.
movie(mov, 1, xyloObj.FrameRate);
```

Reading Variable Frame Rate Video

Some files store video at a variable frame rate, including many Windows Media® Video files. For these files, `VideoReader` cannot determine the number of frames until you read the last frame.

For example, consider a hypothetical file `VarFrameRate.wmv` that has a variable frame rate. A call to `VideoReader` to create the multimedia object such as

```
obj = VideoReader('VarFrameRate.wmv')
```

returns the following warning and summary information:

```
Warning: Unable to determine the number of frames in this file.
```

```
Summary of Multimedia Reader Object for 'VarFrameRate.wmv'.
```

```
Video Parameters: 23.98 frames per second, RGB24 1280x720.  
Unable to determine video frames available.
```

Counting Frames. To determine the number of frames in a variable frame rate file, call `read` with the following syntax:

```
lastFrame = read(obj, inf);
```

This command reads the last frame and sets the `NumberOfFrames` property of the multimedia object. Because `VideoReader` must decode all video data to count the frames reliably, the call to `read` sometimes takes a long time to run.

Specifying the Frames to Read. For any video file, you can specify the frames to read with a range of indices. Usually, if you request a frame beyond the end of the file, `read` returns an error.

However, if the file uses a variable frame rate, and the requested range straddles the end of the file, `read` returns a partial result. For example, given a file with 2825 frames associated with the multimedia object `obj`, a call to read frames 2800 - 3000 as follows:

```
images = read(obj, [2800 3000]);
```

returns:

Warning: The end of file was reached before the requested frames were read completely. Frames 2800 through 2825 were returned.

Supported Video File Formats

The `VideoReader` function reference page lists file formats that `VideoReader` usually can read, including AVI, MPEG-1, and Motion JPEG 2000. Sometimes `VideoReader` can read files in unlisted formats, and sometimes it cannot read files in listed formats.

For video data, the term “file format” often refers either to the *container format* or the *codec*. A container format describes the layout of the file, while a codec describes how to code/decode the data. Many container formats support multiple codecs.

To read a video file, any application must:

- Recognize the container format (such as AVI). The `VideoReader` function reference page lists the supported container formats.
- Have access to the codec associated with the particular file. Some codecs are part of standard Windows and Macintosh system installations, and allow you to play video in Windows Media Player or QuickTime®. `VideoReader` can access most, but not all, of these codecs.
- Properly interpret the codec. `VideoReader` cannot always read files associated with codecs that were not part of your original system installation.

To see the codec associated with a video file, use `mmfileinfo` and view the `Format` field. For example, given a hypothetical AVI file that uses the Indeo® 5 codec, the following code:

```
info = mmfileinfo('myfile.avi');
info.Video.Format
```

returns

```
ans =
IV50
```

Importing Binary Data with Low-Level I/O

Low-level file I/O functions allow the most direct control over reading or writing data to a file. However, these functions require that you specify more detailed information about your file than the easier-to-use *high-level functions*. For a complete list of high-level functions and the file formats they support, see the `file formats` reference page.

If the high-level functions cannot import your data, use one of the following:

- `fscanf`, which reads formatted data in a text or ASCII file; that is, a file you can view in a text editor. For more information, see “Reading Data in a Formatted Pattern” on page 1-27.
- `fgetl` and `fgets`, which read one line of a file at a time, where a newline character separates each line. For more information, see “Reading Data Line-by-Line” on page 1-29.
- `fread`, which reads a stream of data at the byte or bit level. For more information, see the following sections:
 - “Reading Binary Data in a File” on page 1-116
 - “Saving Memory” on page 1-118
 - “Reading Portions of a File” on page 1-119
 - “Reading Files Created on Other Systems” on page 1-122
 - “Opening Files with Different Character Encodings” on page 1-123

Note The low-level file I/O functions are based on functions in the ANSI Standard C Library. However, MATLAB includes *vectorized* versions of the functions, to read and write data in an array with minimal control loops.

Reading Binary Data in a File

As with any of the low-level I/O functions, before importing, open the file with `fopen`, and obtain a file identifier. When you finish processing a file, close it with `fclose(fileID)`.

By default, `fread` reads a file 1 byte at a time, and interprets each byte as an 8-bit unsigned integer (`uint8`). `fread` creates a column vector, with one element for each byte in the file. The values in the column vector are of class `double`.

For example, consider the file `nine.bin`, created as follows:

```
fid = fopen('nine.bin','w');
fwrite(fid, [1:9]);
fclose(fid);
```

To read all data in the file into a 9-by-1 column vector of class `double`:

```
fid = fopen('nine.bin');
col9 = fread(fid);
fclose(fid);
```

Changing the Dimensions of the Array

By default, `fread` reads all values in the file into a column vector. However, you can specify the number of values to read, or describe a two-dimensional output matrix.

For example, to read `nine.bin`, described in the previous example:

```
fid = fopen('nine.bin');

% Read only the first six values
col6 = fread(fid, 6);

% Return to the beginning of the file
frewind(fid);

% Read first four values into a 2-by-2 matrix
frewind(fid);
two_dim4 = fread(fid, [2, 2]);

% Read into a matrix with 3 rows and
% unspecified number of columns
frewind(fid);
two_dim9 = fread(fid, [3, inf]);
```

```
% Close the file
fclose(fid);
```

Describing the Input Values

If the values in your file are not 8-bit unsigned integers, specify the size of the values.

For example, consider the file `fpoint.bin`, created with double-precision values as follows:

```
myvals = [pi, 42, 1/3];

fid = fopen('fpoint.bin','w');
fwrite(fid, myvals, 'double');
fclose(fid);
```

To read the file:

```
fid = fopen('fpoint.bin');

% read, and transpose so samevals = myvals
samevals = fread(fid, 'double')';

fclose(fid);
```

For a complete list of precision descriptions, see the `fread` function reference page.

Saving Memory

By default, `fread` creates an array of class `double`. Storing double-precision values in an array requires more memory than storing characters, integers, or single-precision values.

To reduce the amount of memory required to store your data, specify the class of the array using one of the following methods:

- Match the class of the input values with an asterisk ('*'). For example, to read single-precision values into an array of class `single`, use the command:

```
mydata = fread(fid, '*single')
```

- Map the input values to a new class with the '=>' symbol. For example, to read `uint8` values into an `uint16` array, use the command:

```
mydata = fread(fid, 'uint8=>uint16')
```

For a complete list of precision descriptions, see the `fread` function reference page.

Reading Portions of a File

MATLAB low-level functions include several options for reading portions of binary data in a file:

- Read a specified number of values at a time, as described in “Changing the Dimensions of the Array” on page 1-117. Consider combining this method with “Testing for End of File” on page 1-119.
- Move to a specific location in a file to begin reading. For more information, see “Moving within a File” on page 1-120.
- Skip a certain number of bytes or bits after each element read. For an example, see “Writing and Reading Complex Numbers” on page 2-77.

Testing for End of File

When you open a file, MATLAB creates a pointer to indicate the current position within the file.

Note Opening an empty file does *not* move the file position indicator to the end of the file. Read operations, and the `fseek` and `frewind` functions, move the file position indicator.

Use the `feof` function to check whether you have reached the end of a file. `feof` returns a value of 1 when the file pointer is at the end of the file. Otherwise, it returns 0.

For example, read a large file in parts:

```
filename = 'largedata.dat'; % hypothetical file
segsz = 10000;

fid = fopen(filename);

while ~feof(fid)
    currData = fread(fid, segsz);
    if ~isempty(currData)
        disp('Current Data:');
        disp(currData);
    end
end

fclose(fid);
```

Moving within a File

To read or write selected portions of data, move the file position indicator to any location in the file. For example, call `fseek` with the syntax

```
fseek(fid, offset, origin);
```

where:

- *fid* is the file identifier obtained from `fopen`.
- *offset* is a positive or negative offset value, specified in bytes.
- *origin* specifies the location from which to calculate the position:

'bof'	Beginning of file
'cof'	Current position in file
'eof'	End of file

Alternatively, to move easily to the beginning of a file:

```
frewind(fid);
```

Use `ftell` to find the current position within a given file. `ftell` returns the number of bytes from the beginning of the file.

For example, create a file `five.bin`:

```
A = 1:5;
fid = fopen('five.bin','w');
fwrite(fid, A, 'short');
fclose(fid);
```

Because the call to `fwrite` specifies the `short` format, each element of `A` uses two storage bytes in `five.bin`.

Reopen `five.bin` for reading:

```
fid = fopen('five.bin','r');
```

Move the file position indicator forward 6 bytes from the beginning of the file:

```
status = fseek(fid,6,'bof');
```

File Position	bof	1	2	3	4	5	6	7	8	9	10	eof
File Contents		0	1	0	2	0	3	0	4	0	5	
File Position Indicator								↑				

Read the next element:

```
four = fread(fid,1,'short');
```

The act of reading advances the file position indicator. To determine the current file position indicator, call `ftell`:

```
position = ftell(fid)
```

```
position =
      8
```

File Position	bof	1	2	3	4	5	6	7	8	9	10	eof
File Contents		0	1	0	2	0	3	0	4	0	5	
File Position Indicator										↑		

To move the file position indicator back 4 bytes, call `fseek` again:

```
status = fseek(fid,-4,'cof');
```

File Position	bof	1	2	3	4	5	6	7	8	9	10	eof
File Contents		0	1	0	2	0	3	0	4	0	5	
File Position Indicator					↑							

Read the next value:

```
three = fread(fid,1,'short');
```

Reading Files Created on Other Systems

Different operating systems store information differently at the byte or bit level:

- *Big-endian* systems store bytes starting with the largest address in memory (that is, they start with the big end).
- *Little-endian* systems store bytes starting with the smallest address (the little end).

Windows systems use little-endian byte ordering, and UNIX systems use big-endian byte ordering.

To read a file created on an opposite-endian system, specify the byte ordering used to create the file. You can specify the ordering in the call to open the file, or in the call to read the file.

For example, consider a file with double-precision values named `little.bin`, created on a little-endian system. To read this file on a big-endian system, use one (or both) of the following commands:

- Open the file with

```
fid = fopen('little.bin', 'r', 'l')
```

- Read the file with

```
mydata = fread(fid, 'double', 'l')
```

where 'l' indicates little-endian ordering.

If you are not sure which byte ordering your system uses, call the `computer` function:

```
[cinfo, maxsize, ordering] = computer
```

The returned *ordering* is 'L' for little-endian systems, or 'B' for big-endian systems.

Opening Files with Different Character Encodings

Encoding schemes support the characters required for particular alphabets, such as those for Japanese or European languages. Common encoding schemes include US-ASCII or UTF-8.

The encoding scheme determines the number of bytes required to read or write `char` values. For example, US-ASCII characters always use 1 byte, but UTF-8 characters use up to 4 bytes. MATLAB automatically processes the required number of bytes for each `char` value based on the specified encoding scheme. However, if you specify a `uchar` precision, MATLAB processes each byte as `uint8`, regardless of the specified encoding.

If you do not specify an encoding scheme, `fopen` opens files for processing using the default encoding for your system. To determine the default, open a file, and call `fopen` again with the syntax:

```
[filename, permission, machineformat, encoding] = fopen(fid);
```

If you specify an encoding scheme when you open a file, the following functions apply that scheme: `fscanf`, `fprintf`, `fgetl`, `fgets`, `fread`, and `fwrite`.

For a complete list of supported encoding schemes, and the syntax for specifying the encoding, see the `fopen` reference page.

Exporting Data

The MATLAB application can write data to various file formats, discussed in the following sections:

- “Exporting to MAT-Files” on page 2-2
- “Exporting to Text Data Files” on page 2-8
- “Exporting to XML Documents” on page 2-19
- “Exporting to Excel Spreadsheets” on page 2-23
- “Exporting to Scientific Data Files” on page 2-26
- “Exporting to Images” on page 2-54
- “Exporting to Audio and Video” on page 2-70
- “Exporting Binary Data with Low-Level I/O” on page 2-72
- “Creating Temporary Files” on page 2-79

Exporting to MAT-Files

In this section...

“What Are MAT-Files?” on page 2-2

“Saving Variables from the Workspace” on page 2-2

“Updating an Existing MAT-File” on page 2-4

“MAT-File Versions” on page 2-5


What Are MAT-Files?

MAT-files are binary MATLAB format files that can store the variables in your current workspace for later use. MAT-files use the `.mat` extension.

This section explains how to save data to a MAT-file. For information on loading variables back into the MATLAB workspace, see “Importing MAT-Files” on page 1-14. For information on accessing MAT-files with your custom application, see “Importing and Exporting MAT-Files from C/C++ and Fortran Programs” in the MATLAB External Interfaces documentation.

Saving Variables from the Workspace

To save data interactively, use any of the following options:

- Select **File** > **Save Workspace As**.
- Click the save button  in the Workspace browser toolbar.
- Drag variables from the Workspace browser to the Current Folder browser.

For more information on these options, see “Saving the Current Workspace”.

Alternatively, call the `save` function. For more information, see the following sections:

- “Basic Saving and Viewing of MAT-Files” on page 2-3
- “Saving Selected Variables” on page 2-3
- “Saving Structures” on page 2-3

Basic Saving and Viewing of MAT-Files

To export the current contents of the workspace to a MAT-file, use the `save` function. For example, the following command creates a file named `firstfile.mat`:

```
save('firstfile.mat')
```

To view the contents of the file, use the `whos` function or the Details Panel of the Current Folder browser. For example:

```
whos('-file', 'firstfile.mat')
```

Saving Selected Variables

To save selected variables from the workspace, list the variables in the call to the `save` function. For example, if the current workspace contains variables A through Z, save B and C to `bc.mat` as follows:

```
save('bc.mat', 'B', 'C')
```

To save variables with names that match a pattern, use one of the following options:

- Include the wildcard character (*). For example, save all variables that start with `str` to a file named `strinfo.mat`:

```
save('strinfo.mat', 'str*')
```

- Apply the `-regexp` option, which matches variables to regular expressions. For example, save all variables that start with `Mon`, `Tue`, or `Wed` to a file named `myfile.mat`:

```
save('myfile.mat', '-regexp', '^Mon|^Tue|^Wed')
```

For more information, see “Regular Expressions” in the MATLAB Programming Fundamentals documentation.

Saving Structures

If any of the variables in the workspace are structure arrays, the default behavior of the `save` function is to store the entire structure. To store fields of a structure array as individual variables, use the `-struct` option.

For example, consider structure `S`:

```
S.a = 12.7; S.b = {'abc', [4 5; 6 7]}; S.c = 'Hello!';
```

Save the entire structure to `newstruct.mat` with the usual syntax:

```
save('newstruct.mat', 'S')
```

The file contains the variable `S`:

Name	Size	Bytes	Class
S	1x1	550	struct

Alternatively, save the fields individually with the `-struct` option:

```
save('newstruct.mat', '-struct', 'S')
```

The file contains variables `a`, `b`, and `c`, but not `S`:


Name	Size	Bytes	Class
a	1x1	8	double
b	1x2	158	cell
c	1x6	12	char

To save only selected fields, such as `a` and `c`:

```
save('newstruct.mat', '-struct', 'S', 'a', 'c')
```

Updating an Existing MAT-File

The MATLAB save operations update MAT-files in slightly different ways. The following table describes the behavior of each operation.

If you want to...	Use this command...	Or this desktop tool...
Replace the entire contents of the file with variables from the workspace	save	Select File > Save Workspace As. — OR — Click the save button  in the Workspace browser toolbar.
Add new variables to a file — OR — Overwrite the contents of selected variables in a file, but leave other variables intact	save -append For more information, see the save function reference page.	Drag the variables from the Workspace to the Current Folder browser.

MAT-File Versions

By default, all save operations create Version 7 MAT-files. Override the default to:

- Allow access to the file using earlier versions of MATLAB.
- Reduce the time required to load and save some files by storing uncompressed data. For more information, see “Speeding Up Save and Load Operations” on page 2-6.
- Take advantage of Version 7.3 MAT-file storage for data items larger than 2 GB on 64-bit systems.

Note Version 7.3 MAT-files use an HDF5 format that requires some overhead storage to describe the contents of the file. For complex nested cell or structure arrays, Version 7.3 MAT-files are sometimes larger than Version 7 MAT-files.

Overriding the Default MAT-File Version

To identify or change the default version, select **File > Preferences > General > MAT-Files**. Alternatively, specify the version as an option to the `save` function.

For example, to create a MAT-file named `myfile.mat` that you can load with MATLAB Version 6, use the following command:

```
save('myfile.mat', '-v6')
```

The possible version options for the `save` function include `-v4`, `-v6`, `-v7`, and `-v7.3`. For more information about the differences between previous and current MAT-file versions, see the `save` function reference page.

Speeding Up Save and Load Operations

Beginning with Version 7, MATLAB compresses data when writing to MAT-files to save storage space. Data compression and decompression slow down all save operations and some load operations. In most cases, the reduction in file size is worth the additional time spent.

In fact, loading compressed data is sometimes *faster* than loading uncompressed data. For example, consider a block of data in a numeric array saved to both a 10 MB compressed file and a 100 MB uncompressed file. Loading the first 10 MB takes the same amount of time for each file. Loading the remaining 90 MB from the uncompressed file takes nine times as long as loading the first 10 MB. Completing the load of the compressed file requires only the relatively short time to decompress the data.

However, the benefits of data compression are negligible in the following cases:

- The amount of data in each item is small relative to the complexity of its container. For example, simple numeric arrays take less time to compress and uncompress than cell or structure arrays of the same size. Compressing arrays that result in an uncompressed file size of less than 3MB offers limited benefit, unless you are transferring data over a network.
- The data is random, with no repeated patterns or consistent values.

Version 6 MAT-files do not use compression. To create a Version 6 MAT-file, use the methods described in “Overriding the Default MAT-File Version” on page 2-6.

Exporting to Text Data Files

If you want to use your data in another application that reads ASCII files, MATLAB functions offer several data export options. For example, you can create a:

- Rectangular, delimited ASCII data file from an array. For more information, see “Writing to Delimited Data Files” on page 2-8.
- Diary (or log) file of keystrokes and the resulting text output. For more information, see “Writing to a Diary File” on page 2-11.
- Specialized ASCII file using low-level functions such as `fprintf`. For more information, see “Writing to Text Data Files with Low-Level I/O” on page 2-12.
- MEX-file to access your C/C++ or Fortran routine that writes to a particular text file format. For more information, see “Calling C/C++ and Fortran Programs from MATLAB Command Line” in the MATLAB External Interfaces documentation.

Additional MATLAB functions export data to spreadsheets, scientific data formats, and other file formats. For a complete list of supported formats, see the `file formats` table.

Writing to Delimited Data Files

To export a numeric array as a delimited ASCII data file, you can use either the `save` function, specifying the `-ASCII` qualifier, or the `dlmwrite` function.

Both `save` and `dlmwrite` are easy to use. With `dlmwrite`, you can specify any character as a delimiter, and you can export subsets of an array by specifying a range of values.

However, `save -ascii` and `dlmwrite` do not accept cell arrays as input. To create a delimited ASCII file from the contents of a cell array, do one of the following:

- Convert the cell array to a matrix using the `cell2mat` function, then call `save` or `dlmwrite`. Use this approach when your cell array contains only numeric data, and easily translates to a two-dimensional numeric array.

- Export the cell array using low-level file I/O functions. For more information, see “Exporting a Cell Array to a Text File” on page 2-10.

Exporting a Numeric Array to an ASCII File Using `save`

To export the array `A`, where

```
A = [ 1 2 3 4 ; 5 6 7 8 ];
```

to a space-delimited ASCII data file, use the `save` function as follows:

```
save my_data.out A -ASCII
```

To view the file, use the `type` function:

```
type my_data.out
```

```
1.0000000e+000 2.0000000e+000 3.0000000e+000 4.0000000e+000
5.0000000e+000 6.0000000e+000 7.0000000e+000 8.0000000e+000
```

When you use `save` to write a character array to an ASCII file, it writes the ASCII equivalent of the characters to the file. For example, if you write the character string 'hello' to a file, `save` writes the values

```
104 101 108 108 111
```

to the file in 8-digit ASCII format.

To write data in 16-digit format, use the `-double` option. To create a tab-delimited file instead of a space-delimited file, use the `-tabs` option.

Exporting a Numeric Array to an ASCII File Using `dlmwrite`

To export a numeric or character array to an ASCII file with a specified delimiter, use the `dlmwrite` function.

For example, to export the array `A`,

```
A = [ 1 2 3 4 ; 5 6 7 8 ];
```

to an ASCII data file that uses semicolons as a delimiter, use this command:

```
dlmwrite('my_data.out',A, ';')
```

To view the file, use the `type` function:

```
type my_data.out
```

```
1;2;3;4  
5;6;7;8
```

By default, `dlmwrite` uses a comma as a delimiter. You can specify a space (' ') or other character as a delimiter. To specify no delimiter, use empty quotation marks ('').

Exporting a Cell Array to a Text File

To export a cell array that contains nonnumeric data to a text file, use the `fprintf` function.

The `fprintf` function is flexible, but requires that you provide details about the format of your data. Describe each field using format specifiers, such as '%s' for a string, '%d' for an integer, or '%f' for a number in fixed-point notation. (For a complete list of format specifiers, see the `fprintf` reference page.)

The character that you use to separate the format specifiers determines the delimiter for the output file. For example, a format string such as '%d,%d,%d' creates a comma-separated file, while the format '%d %d %d' creates a space-delimited file.

Preface any calls to `fprintf` with a call to `fopen` to open the file, and, when finished, close the file with `fclose`. By default, `fopen` opens a file for read-only access. Use the permission string 'w' to write to the file.

For example, consider the array `mycell`, where

```
mycell = { 'a' 1 2 3 ; 'b' 4 5 6 };
```

To export the cell array, print one row of data at a time. Include a newline character at the end of each row ('\n').

Note Some Windows text editors, including Microsoft® Notepad, require a newline character sequence of '\r\n' instead of '\n'. However, '\n' is sufficient for Microsoft Word or WordPad.

Send `fprintf` the file identifier and the format specifiers to describe the fields in each row:

```
[nrows,ncols]= size(mycell);

filename = 'celldata.dat';
fid = fopen(filename, 'w');

for row=1:nrows
    fprintf(fid, '%s %d %d %d\n', mycell{row,:});
end

fclose(fid);
```

To view the file, use the `type` function:

```
type celldata.dat

a 1 2 3
b 4 5 6
```

For more information, see “Writing to Text Data Files with Low-Level I/O” on page 2-12.

Writing to a Diary File

To keep an activity log of your MATLAB session, use the `diary` function. `diary` creates a verbatim copy of your MATLAB session in a disk file (excluding graphics).

For example, if you have the array `A` in your workspace,

```
A = [ 1 2 3 4; 5 6 7 8 ];
```

execute these commands at the MATLAB prompt to export this array using `diary`:

- 1 Turn on the `diary` function. Optionally, you can name the output file `diary` creates:

```
diary my_data.out
```

- 2 Display the contents of the array you want to export. This example displays the array `A`. You could also display a cell array or other MATLAB class:

```
A =  
    1    2    3    4  
    5    6    7    8
```

- 3 Turn off the `diary` function:

```
diary off
```

`diary` creates the file `my_data.out` and records all the commands executed in the MATLAB session until you turn it off:

```
A =  
    1    2    3    4  
    5    6    7    8
```

```
diary off
```

- 4 Open the diary file `my_data.out` in a text editor and remove the extraneous text, if desired.

Writing to Text Data Files with Low-Level I/O

To create rectangular, delimited ASCII files (such as CSV files) from numeric arrays, use high-level functions such as `dlmwrite`. For more information, see “Writing to Delimited Data Files” on page 2-8.

To create other text files, including combinations of numeric and character data, nonrectangular output files, or files with non-ASCII encoding schemes,

use the low-level `fprintf` function. For more information, see the following sections:

- “Opening the File” on page 2-13
- “Describing the Output” on page 2-13
- “Appending or Overwriting Existing Files” on page 2-15
- “Opening Files with Different Character Encodings” on page 2-18

Note `fprintf` is based on its namesake in the ANSI Standard C Library. However, MATLAB uses a *vectorized* version of `fprintf` that writes data from an array with minimal control loops.

Opening the File

As with any of the low-level I/O functions, before exporting, open or create a file with `fopen`, and obtain a file identifier. By default, `fopen` opens a file for read-only access, so you must specify the permission to write or append, such as `'w'` or `'a'`.

When you finish processing the file, close it with `fclose(fid)`.

Describing the Output

`fprintf` accepts arrays as inputs, and converts the numbers or characters in the arrays to text according to your specifications.

For example, to print floating-point numbers, specify `'%f'`. Other common conversion specifiers include `'%d'` for integers or `'%s'` for strings. For a complete list of conversion specifiers, see the `fprintf` reference page.

To move to a new line in the file, use `'\n'`.

Note Some Windows text editors, including Microsoft Notepad, require a newline character sequence of `'\r\n'` instead of `'\n'`. However, `'\n'` is sufficient for Microsoft Word or WordPad.

`fprintf` reapplies the conversion information to cycle through all values of the input arrays in column order.

For example, create a file named `exptable.txt` that contains a short table of the exponential function, and a text header:

```
% create a matrix y, with two rows
x = 0:0.1:1;
y = [x; exp(x)];

% open a file for writing
fid = fopen('exptable.txt', 'w');

% print a title, followed by a blank line
fprintf(fid, 'Exponential Function\n\n');

% print values in column order
% two values appear on each row of the file
fprintf(fid, '%f %f\n', y);
fclose(fid);
```

To view the file, use the `type` function:

```
type exptable.txt
```

This returns the contents of the file:

```
Exponential Function

0.000000  1.000000
0.100000  1.105171
0.200000  1.221403
0.300000  1.349859
0.400000  1.491825
0.500000  1.648721
0.600000  1.822119
0.700000  2.013753
0.800000  2.225541
0.900000  2.459603
1.000000  2.718282
```

Additional Formatting Options. Optionally, include additional information in the call to `fprintf` to describe field width, precision, or the order of the output values. For example, specify the field width and number of digits to the right of the decimal point in the exponential table:

```
fid = fopen('exptable_new.txt', 'w');

fprintf(fid, 'Exponential Function\n\n');
fprintf(fid, '%6.2f %12.8f\n', y);

fclose(fid);
```

`exptable_new.txt` contains the following:

```
Exponential Function

0.00    1.00000000
0.10    1.10517092
0.20    1.22140276
0.30    1.34985881
0.40    1.49182470
0.50    1.64872127
0.60    1.82211880
0.70    2.01375271
0.80    2.22554093
0.90    2.45960311
1.00    2.71828183
```

For more information, see “Formatting Strings” in the Programming Fundamentals documentation, and the `fprintf` reference page.

Appending or Overwriting Existing Files

By default, `fopen` opens files with read access. To change the type of file access, use the permission string in the call to `fopen`. Possible permission strings include:

- `r` for reading
- `w` for writing, discarding any existing contents of the file
- `a` for appending to the end of an existing file

To open a file for both reading and writing or appending, attach a plus sign to the permission, such as 'w+' or 'a+'. For a complete list of permission values, see the `fopen` reference page.

Note If you open a file for both reading and writing, you must call `fseek` or `frewind` between read and write operations.

Example – Appending to an Existing Text File. Create a file `changing.txt` as follows:

```
myformat = '%5d %5d %5d %5d\n';

fid = fopen('changing.txt','w');
fprintf(fid, myformat, magic(4));
fclose(fid);
```

The current contents of `changing.txt` are:

```
16    5    9    4
 2   11    7   14
 3   10    6   15
13    8   12    1
```

Add the values `[55 55 55 55]` to the end of file:

```
% open the file with permission to append
fid = fopen('changing.txt','a');

% write values at end of file
fprintf(fid, myformat, [55 55 55 55]);

% close the file
fclose(fid);
```

To view the file, use the `type` function:

```
type changing.txt
```

This command returns the new contents of the file:

```

16    5    9    4
 2   11    7   14
 3   10    6   15
13    8   12    1
55   55   55   55

```

Example – Overwriting an Existing Text File. Replace the third line of the file `changing.txt` from the previous example with `[33 33 33 33]`:

```

replaceLine = 3;
myformat = '%5d %5d %5d %5d\n';

% Open the file with permission to read and update.
% Use fgetl, which reads a line at a time,
% to place the file position indicator at the third line.

fid = fopen('changing.txt','r+');
for k=1:(replaceLine-1);
    fgetl(fid);
end;

% call fseek between read and write operations
fseek(fid, 0, 'cof');

% print the new values
fprintf(fid, myformat, [33 33 33 33]);

% close the file
fclose(fid);

```

To view the file, use the `type` function:

```
type changing.txt
```

This command returns the new contents of the file:

```

16    5    9    4
 2   11    7   14
33   33   33   33
13    8   12    1
55   55   55   55

```

Opening Files with Different Character Encodings

Encoding schemes support the characters required for particular alphabets, such as those for Japanese or European languages. Common encoding schemes include US-ASCII or UTF-8.

If you do not specify an encoding scheme, `fopen` opens files for processing using the default encoding for your system. To determine the default, open a file, and call `fopen` again with the syntax:

```
[filename, permission, machineformat, encoding] = fopen(fid);
```

If you specify an encoding scheme when you open a file, the following functions apply that scheme: `fscanf`, `fprintf`, `fgetl`, `fgets`, `fread`, and `fwrite`.

For a complete list of supported encoding schemes, and the syntax for specifying the encoding, see the `fopen` reference page.

Exporting to XML Documents

To write data to an XML file, use the `xmlwrite` function. `xmlwrite` requires that you describe the file in a Document Object Model (DOM) node. For an introduction to DOM nodes, see “What Is an XML Document Object Model (DOM)?” on page 1-34

Creating an XML File

Although each file is different, these are common steps for creating an XML document:

- 1 Create a document node and define the root element by calling this method:

```
docNode =  
com.mathworks.xml.XMLUtils.createDocument('root_element');
```

- 2 Get the node corresponding to the root element by calling `getDocumentElement`. The root element node is required for adding child nodes.
- 3 Add element, text, comment, and attribute nodes by calling methods on the document node. Useful methods include:

- `createElement`
- `createTextNode`
- `createComment`
- `setAttribute`

For a complete list of the methods and properties of DOM nodes, select the `org.w3c.dom` package at:
<https://jaxp-sources.dev.java.net/nonav/docs/api/>.

- 4 As needed, define parent/child relationships by calling `appendChild` on the parent node.

Tip Text nodes are always children of element nodes. To add a text node, call `createTextNode` on the document node, and then call `appendChild` on the parent element node.

Example – Creating an XML File with `xmlwrite`

Suppose that you want to create an `info.xml` file for the Upslope Area Toolbox (described in “Adding HTML Help Files to the Help Browser”), as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<toc version="2.0">
  <tocitem target="upslope_product_page.html">Upslope Area Toolbox<!-- Functions -->
    <tocitem target="demFlow_help.html">demFlow</tocitem>
    <tocitem target="facetFlow_help.html">facetFlow</tocitem>
    <tocitem target="flowMatrix_help.html">flowMatrix</tocitem>
    <tocitem target="pixelFlow_help.html">pixelFlow</tocitem>
  </tocitem>
</toc>
```

To create this file using `xmlwrite`, follow these steps:

- 1** Create the document node and root element, `toc`:

```
docNode = com.mathworks.xml.XMLUtils.createDocument('toc');
```

- 2** Identify the root element, and set the `version` attribute:

```
toc = docNode.getDocumentElement;
toc.setAttribute('version', '2.0');
```

- 3** Add the `tocitem` element node for the product page. Each `tocitem` element in this file has a `target` attribute and a child text node:

```
product = docNode.createElement('tocitem');
product.setAttribute('target', 'upslope_product_page.html');
product.appendChild(docNode.createTextNode('Upslope Area Toolbox'));
toc.appendChild(product)
```

- 4** Add the comment:


```
product.appendChild(docNode.createComment(' Functions '));
```

- 5** Add a `tocitem` element node for each function, where the target is of the form `function_help.html`:

```
functions = {'demFlow','facetFlow','flowMatrix','pixelFlow'};
for idx = 1:numel(functions)
    curr_node = docNode.createElement('tocitem');

    curr_file = [functions{idx} '_help.html'];
    curr_node.setAttribute('target',curr_file);

    % Child text is the function name.
    curr_node.appendChild(docNode.createTextNode(functions{idx}));
    product.appendChild(curr_node);
end
```

- 6** Export the DOM node to `info.xml`, and view the file with the `type` function:

```
xmlwrite('info.xml',docNode);
type('info.xml');
```

Updating an Existing XML File

To change data in an existing file, call `xmlread` to import the file into a DOM node. Traverse the node and add or change data using methods defined by the World Wide Web consortium, such as:

- `getElementsByTagName`
- `getFirstChild`
- `getNextSibling`
- `getNodeName`
- `getNodeType`

When the DOM node contains all your changes, call `xmlwrite` to overwrite the file.

For a complete list of the methods and properties of DOM nodes, select the `org.w3c.dom` package at: <https://jaxp-sources.dev.java.net/nonav/docs/api/>.

For examples that use these methods, see:

- “Example — Finding Text in an XML File” on page 1-35
- “Example — Creating an XML File with `xmlwrite`” on page 2-20
- `xmlread` and `xmlwrite`

Exporting to Excel Spreadsheets

Use `xlswrite` to export a matrix to an Excel spreadsheet file. With `xlswrite`, you can export data from the workspace to any worksheet in the file, and to any location within that worksheet. By default, `xlswrite` writes your matrix data to the first worksheet in the file, starting at cell A1.

`xlswrite` can write to any file format recognized by your version of Excel for Windows. If you have Excel 2003 installed, but want to write to a 2007 format (such as XLSX, XLSB, or XLSM), you must install the Office 2007 Compatibility Pack.

To write data to an Excel file, specify the name and extension of the output file in the call to `xlswrite`. If the file already exists, `xlswrite` writes data in the existent file format. If the file does not exist, `xlswrite` creates a new file, using the format that corresponds to the file extension you specify. If you do not specify a file extension, `xlswrite` applies the XLS extension, and writes a new file in the XLS format.

Note If you are using a system that does not have Excel for Windows installed, `xlswrite` writes your data to a comma-separated value (CSV) file.

Example – Writing to an XLS File

This example writes a mix of text and numeric data to the file `climate.xls`. Call `xlswrite`, specifying a worksheet labeled `Temperatures`, and the region within the worksheet where you want to write the data. `xlswrite` writes the 4-by-2 matrix `d` to the rectangular region that starts at cell E1 in its upper-left corner:

```
d = {'Time', 'Temp'; 12 98; 13 99; 14 97}
d =
    'Time'    'Temp'
    [ 12]    [ 98]
    [ 13]    [ 99]
    [ 14]    [ 97]

xlswrite('climate.xls', d, 'Temperatures', 'E1');
```

Adding a New Worksheet

If the target worksheet does not already exist in the file, `xlswrite` displays the following warning:

```
Warning: Added specified worksheet.
```

You can disable these warnings with this command:

```
warning off MATLAB:xlswrite:AddSheet
```

Converting Dates

In both MATLAB and Excel applications, dates can be represented as character strings or numeric values. For example, May 31, 2009, can be represented as the character string `'05/31/09'` or as the numeric value 733924. Within MATLAB, the `datestr` and `datenum` functions allow you to convert easily between string and numeric representations.

If you export a matrix with dates stored as strings, you do not need to convert the dates before processing in Excel.

However, if you export a matrix with dates stored as numbers, you must convert the dates. Both Excel and MATLAB represent numeric dates as a number of serial days elapsed from a specific reference date, but the applications use different reference dates.

The following table lists the reference dates for MATLAB and Excel. For more information on the 1900 and 1904 date systems, see the Excel help.

Application	Reference Date
MATLAB	January 0, 0000
Excel for Windows	January 1, 1900
Excel for the Macintosh	January 2, 1904

Example — Exporting to an Excel File with Numeric Dates

Consider a numeric matrix `wt_log`. The first column contains numeric dates, and the second column contains weights:

```
wt_log = [729698 174.8; ...
```

```

729726 175.3; ...
729760 190.4; ...
729787 185.7];

```

```

% To view the dates before exporting, call datestr:
datestr(wt_log(:,1))

```

The formatted dates returned by `datestr` are:

```

04-Nov-1997
02-Dec-1997
05-Jan-1998
01-Feb-1998

```

To export the numeric matrix to Excel for Windows (and use the default 1900 date system), convert the dates:

```

datecol = 1;
wt_log(:,datecol) = wt_log(:,datecol) - datenum('30-Dec-1899');
xlswrite('new_log.xls', wt_log);

```

To export for use in Excel for the Macintosh (with the default 1904 date system), convert as follows:

```

datecol = 1;
wt_log(:,datecol) = wt_log(:,datecol) - datenum('01-Jan-1904');
xlswrite('new_log.xls', wt_log);

```

Formatting Cells in Excel Files

To write data to Excel files on Windows systems with custom formats (such as fonts or colors), access the COM server directly using `actxserver` rather than `xlswrite`. For example, Technical Solution 1-QLD4K uses `actxserver` to establish a connection between MATLAB and Excel, write data to a worksheet, and specify the colors of the cells.

For more information, see:

- “COM Support for MATLAB Software”
- “MATLAB COM Client Support”

Exporting to Scientific Data Files

In this section...
“Exporting to Common Data File Format (CDF) Files” on page 2-26
“Exporting to Network Common Data Form (netCDF) Files” on page 2-29
“Exporting to Hierarchical Data Format (HDF5) Files” on page 2-32
“Exporting to Hierarchical Data Format (HDF4) Files” on page 2-43

Exporting to Common Data File Format (CDF) Files

CDF was created by the National Space Science Data Center (NSSDC) to provide a self-describing data storage and manipulation format that matches the structure of scientific data and applications (i.e., statistical and numerical methods, visualization, and management). For more information about this format, see the CDF Web site.

To export data from the MATLAB workspace to a Common Data Format (CDF) file, you can use either the MATLAB high-level function `cdfwrite` or the low-level functions in the `cdflib` package. The high-level function provides a convenient interface. The low-level functions provide more control over the export operation. The following sections provide more information.

- “Using the High-Level CDF Function to Export Data” on page 2-26
- “Using the Low-level CDF Functions to Export Data” on page 2-27

Using the High-Level CDF Function to Export Data

To write data from the MATLAB workspace to a Common Data Format file use the `cdfwrite` function. Using this function, you can write variables and attributes to the file, specifying their names and associated values.

This example shows how to write date information to a CDF file. Note how the example uses the CDF epoch object constructor, `cdfepoch`, to convert a MATLAB serial date number into a CDF epoch.

```
cdfwrite('myfile',{ 'Time_val',cdfepoch(now)});
```

You can convert a `cdfepoch` object back into a MATLAB serial date number with the `todatenum` function. For more information, see “Representing CDF Time Values” on page 1-48.

Using the Low-level CDF Functions to Export Data

To export (write) data from a Common Data Format (CDF) file, you can use the MATLAB low-level CDF functions. The MATLAB functions correspond to dozens of routines in the CDF C API library. For a complete list of all the MATLAB low-level CDF functions, see `cdflib`.

This section does not attempt to describe all features of the CDF library or explain basic CDF programming concepts. To use the MATLAB CDF low-level functions effectively, you must be familiar with the CDF C interface. Documentation about CDF, version 3.3.0, is available at the [CDF Web site](#).

The following example shows how to use low-level functions to write data to a CDF file.

- 1 Create a new CDF file. For information about opening an existing CDF file, see “Using the CDF Library Low-Level Functions to Import Data” on page 1-48.

```
cdfid = cdflib.create('my_file.cdf');
```

- 2 Create some variables in the CDF file.

```
time_id = cdflib.createVar(cdfid, 'Time', 'cdf_int4', 1, [], true, []);
```

```
lat_id = cdflib.createVar(cdfid, 'Latitude', 'cdf_int2', 1, 181, true, true);
```

```
dimSizes = [20 10];
```

```
dimVarys = [true true];
```

```
image_id = cdflib.createVar(cdfid, 'Image', 'cdf_int4', 1, dimSizes, true, [true true]);
```

- 3 Write data to the variables.

```
% Write time data
```

```
cdflib.putVarRecordData(cdfid, time_id, 0, int32(23));
```

```
cdflib.putVarRecordData(cdfid, time_id, 1, int32(24));
```

```
% Write the latitude data
data = int16([-90:90]);
recspec = [0 1 1];
dimspec = { 0 181 1 };
cdfplib.hyperPutVarData(cdfid,lat_id,recspec,dimspec,data);
```

```
% Write data for the image zVariable
recspec = [0 3 1];
dimspec = { [0 0], [20 10], [1 1] };
data = reshape(int32([0:599]), [20 10 3]);
cdfplib.hyperPutVarData(cdfid,image_id,recspec,dimspec,data);
```

4 Create a global attribute in the CDF file and write data to the attribute..

```
titleAttrNum = cdfplib.createAttr(cdfid,'TITLE','global_scope');
```

```
% Write the global attribute entries
cdfplib.putAttrEntry(cdfid,titleAttrNum,0,'CDF_CHAR','cdf Title');
cdfplib.putAttrEntry(cdfid,titleAttrNum,1,'CDF_CHAR','Author');
```

5 Create attributes associated with variables in the CDF file and write data to the attribute.

```
fieldAttrNum = cdfplib.createAttr(cdfid,'FIELDNAM','variable_scope');
unitsAttrNum = cdfplib.createAttr(cdfid,'UNITS','variable_scope');
```

```
% Write the time variable attributes
cdfplib.putAttrEntry(cdfid,fieldAttrNum,time_id,'CDF_CHAR','Time of observation');
cdfplib.putAttrEntry(cdfid,unitsAttrNum,time_id,'CDF_CHAR','Hours');
```

6 Close the CDF file.

```
cdfplib.close(cdfid);
```

Exporting to Network Common Data Form (netCDF) Files

Note For information about exporting to Common Data Format (CDF) files, which have a completely separate, incompatible format, see “Exporting to Network Common Data Form (netCDF) Files” on page 2-29.

MATLAB provides access to the routines in the netCDF C library that you can use to read data from netCDF files and write data to netCDF files. MATLAB provides this access through a set of MATLAB functions that correspond to the functions in the netCDF C library. MATLAB groups the functions into a package, called `netcdf`. To call one of the functions in the package, you must specify the package name. For a complete list of all the functions, see `netcdf`.

Note The MATLAB netCDF functions support netCDF Version 3.6.2.

This section does not attempt to describe all features of the netCDF library or explain basic netCDF programming concepts. To use the MATLAB netCDF functions effectively, you should be familiar with the information about netCDF contained in the *NetCDF C Interface Guide* for version 3.6.2.

For additional information, see “Mapping netCDF API Syntax to MATLAB Function Syntax” on page 1-51.

Storing Data in a netCDF File

To store data in a netCDF file, you can use the MATLAB netCDF functions to create a file, define dimensions in the file, create a variable in the file, and write data to the variable. Note that you must define dimensions in the file before you can create variables. To run the following example, you must have write permission in your current directory.

- 1 Create a variable in the MATLAB workspace. This example creates a 50-element vector of numeric values named `my_data`. The vector is of class `double`.

```
my_data = linspace(0,49,50);
```

- 2 Create a netCDF file (or open an existing file). The example uses the `netcdf.create` function to create a new file, named `my_file.nc`. The `NOCLOBBER` parameter is a netCDF file access constant that indicates that you do not want to overwrite an existing file with the same name. See `netcdf.create` for more information about these file access constants.

```
ncid = netcdf.create('my_file.nc','NOCLOBBER');
```

When you create a netCDF file, the file opens in define mode. You must be in define mode to define dimensions and variables.

- 3 Define a dimension in the file, using the `netcdf.defDim` function. You must define dimensions in the file before you can define variables and write data to the file. When you define a dimension, you give it a name and a length. To create an unlimited dimension, i.e., a dimension that can grow, specify the constant `NC_UNLIMITED` in place of the dimension length.

```
dimid = netcdf.defDim(ncid,'my_dim',50);
```

- 4 Define a variable on the dimension, using the `netcdf.defVar` function. When you define a variable, you give it a name, data type, and a dimension ID.

```
varid = netcdf.defVar(ncid,'my_var','NC_BYTE',dimid);
```

You must use one of the netCDF constants to specify the data type, listed in the following table.

MATLAB Class	netCDF Data Type	Notes
<code>int8</code>	<code>NC_BYTE</code>	netCDF interprets byte data as either signed or unsigned.
<code>uint8</code>	<code>NC_BYTE</code>	netCDF interprets byte data as either signed or unsigned.
<code>char</code>	<code>NC_CHAR</code>	
<code>int16</code>	<code>NC_SHORT</code>	
<code>uint16</code>	No equivalent	

MATLAB Class	netCDF Data Type	Notes
int32	NC_INT	
uint32	No equivalent	
int64	No equivalent	
uint64	No equivalent	
single	NC_FLOAT	
double	NC_DOUBLE	

- 5** Take the netCDF file out of define mode. To write data to a file, you must be in data mode.

```
netcdf.endDef(ncid);
```

- 6** Write the data from the MATLAB workspace into the variable in the netCDF file, using the `netcdf.putVar` function. Note that the data in the workspace is of class `double` but the variable in the netCDF file is of type `NC_BYTE`. The MATLAB netCDF functions automatically do the conversion.

```
netcdf.putVar(ncid,varid,my_data);
```

- 7** Close the file, using the `netcdf.close` function.

```
netcdf.close(ncid);
```

- 8** Verify that the data was written to the file by opening the file and reading the data from the variable into a new variable in the MATLAB workspace. Because the variable is the first variable in the file (and the only one), you can specify 0 (zero) for the variable ID—identifiers are zero-based indexes.

```
ncid2 = netcdf.open('my_file.nc','NC_NOWRITE');
```

```
data_in_file = netcdf.getVar(ncid2,0)
```

```
data_in_file =
```

```
0
```

```
1
```

```
2  
3  
4  
5  
6  
7  
8  
9  
.br/>.br/.
```

Because you stored the data in the file as `NC_BYTE`, MATLAB reads the data from the variable into the workspace as class `int8`.

Exporting to Hierarchical Data Format (HDF5) Files

Hierarchical Data Format, Version 5, (HDF5) is a general-purpose, machine-independent standard for storing scientific data in files, developed by the National Center for Supercomputing Applications (NCSA). HDF5 is used by a wide range of engineering and scientific fields that want a standard way to store data so that it can be shared. For more information about the HDF5 file format, read the HDF5 documentation available at the HDF Web site (<http://www.hdfgroup.org>).

The MATLAB *high-level* HDF5 function `hdf5write` provides an easy way to export data to an HDF5 file.

MATLAB also provides direct access to the over 200 functions in the HDF5 library with *low-level* functions that correspond to the functions in the HDF5 library. In this way, you can access the features of the HDF5 library from MATLAB, such as reading and writing complex data types and using the HDF5 subsetting capabilities. For more information, see “Using the MATLAB Low-Level HDF5 Functions” on page 2-34.

Note For information about exporting to HDF4 files, which have a completely separate, incompatible format, see “Exporting to Hierarchical Data Format (HDF4) Files” on page 2-43.

Using the MATLAB High-Level HDF5 Functions

The easiest way to write data or metadata from the MATLAB workspace to an HDF5 file is to use `hdf5write`. As arguments, specify:

- Name of an existing HDF5 file, or the name you want to assign to a new file.
- Name of an existing data set or attribute, or the name you want to assign to a new data set or attribute. To learn how to determine the name of data sets in an existing HDF5 file, see “Determining the Contents of an HDF5 File” on page 1-59.
- Data or metadata you want to write to the file. `hdf5write` converts MATLAB data types to the appropriate HDF5 data type automatically. For nonatomic data types, you can also create HDF5 objects to represent the data.

This example creates a 5-by-5 array of `uint8` values and then writes the array to an HDF5 file. By default, `hdf5write` overwrites the file, if it already exists. The example specifies an `hdf5write` mode option to append data to existing file.

- 1 Create a MATLAB variable in the workspace. This example creates a 5-by-5 array of `uint8` values.

```
testdata = uint8(magic(5))
```

- 2 Write the data to an HDF5 file. As arguments to `hdf5write`, the example specifies the name you want to assign to the HDF5 file, the name you want to assign to the data set, and the MATLAB variable.

```
hdf5write('myfile.h5', '/dataset1', testdata)
```

To add data to an existing file, you must use the `'writemode'` option, specifying the `'append'` value. The file must already exist and it cannot already contain a data set with the same name

```
hdf5write('myfile.h5', '/dataset12', testdata, 'writemode', 'append')
```

If you are writing simple data sets, such as scalars, strings, or a simple compound data types, you can just pass the data directly to `hdf5write`; this function automatically maps the MATLAB data types to appropriate HDF5 data types. However, if your data is a complex data set, you might need to

use one of the predefined MATLAB HDF5 objects to pass the data to the `hdf5write` function. The HDF5 objects are designed for situations where the mapping between MATLAB and HDF5 types is ambiguous.

For example, when passed a cell array of strings, the `hdf5write` function writes a data set made up of strings, not a data set of arrays containing strings. If that is not the mapping you intend, use HDF5 objects to specify the correct mapping. In addition, note that HDF5 makes a distinction between the size of a data set and the size of a data type. In MATLAB, data types are always scalar. In HDF5, data types can have a size; that is, types can be either scalar (like MATLAB) or m-by-n. In HDF5, a 5-by-5 data set containing a single `uint8` value in each element is distinct from a 1-by-1 data set containing a 5-by-5 array of `uint8` values. In the first case, the data set contains 25 observations of a single value; in the second case, the data set contains a single observation with 25 values. For more information about the MATLAB HDF5 data type objects, see “Mapping Composite Data Types” on page 1-65.

Using the MATLAB Low-Level HDF5 Functions

MATLAB provides direct access to the over 200 functions in the HDF5 library by creating MATLAB functions that correspond to the functions in the HDF5 library. In this way, you can access the features of the HDF5 library from MATLAB, such as reading and writing complex data types and using the HDF5 subsetting capabilities.

The HDF5 library organizes the library functions into groups, called *interfaces*. For example, all the routines related to working with files, such as opening and closing, are in the H5F interface, where *F* stands for file. MATLAB organizes the low-level HDF5 functions into classes that correspond to each HDF5 interface. For example, the MATLAB functions that correspond to the HDF5 file interface (H5F) are in the `@H5F` class directory. For a complete list of the HDF5 interfaces and the corresponding MATLAB class directories, see `hdf5`.

The following sections provide more details about how to use the MATLAB HDF5 low-level functions. Topics covered include:

- “Mapping HDF5 Function Syntax to MATLAB Function Syntax” on page 2-35

- “Mapping Between HDF5 Data Types and MATLAB Data Types” on page 2-37
- “Example: Using the MATLAB HDF5 Low-level Functions” on page 2-39
- “Preserving the Correct Layout of Your Data” on page 2-43

Note This section does not attempt to describe all features of the HDF5 library or explain basic HDF5 programming concepts. To use the MATLAB HDF5 low-level functions effectively, you must refer to the official HDF5 documentation available at the HDF Web site (<http://www.hdfgroup.org>).

Mapping HDF5 Function Syntax to MATLAB Function Syntax. In most cases, the syntax of the MATLAB low-level HDF5 functions is identical to the syntax of the corresponding HDF5 library functions. For example, the following is the function signature of the `H5Fopen` function in the HDF5 library. In the HDF5 function signatures, `hid_t` and `herr_t` are HDF5 types that return numeric values that represent object identifiers or error status values.

```
hid_t H5Fopen(const char *name, unsigned flags, hid_t access_id ) /* C syntax */
```

In MATLAB, each function in an HDF5 interface is a method of a MATLAB class. To view the function signature for a function, specify the class directory name and then the function name, as in the following.

```
help @H5F/open
```

The following shows the signature of the corresponding MATLAB function. First note that, because it’s a method of a class, you must use the dot notation to call the MATLAB function: `H5F.open`. This MATLAB function accepts the same three arguments as the HDF5 function: a text string for the name, an HDF5-defined constant for the flags argument, and an HDF5 property list ID. You use property lists to specify characteristics of many different HDF5 objects. In this case, it’s a file access property list. Refer to the HDF5 documentation to see which constants can be used with a particular function and note that, in MATLAB, constants are passed as text strings.

```
file_id = H5F.open(name, flags, plist_id)
```

There are, however, some functions where the MATLAB function signature is different than the corresponding HDF5 library function. The following sections describe some general differences that you should keep in mind when using the MATLAB low-level HDF5 functions.

Output Parameters Become Return Values

Some HDF5 library functions use function parameters to return data on the right-hand side (RHS) of the function signature, i.e. as input parameters. The corresponding MATLAB function, because MATLAB allows multiple return values, moves these output parameters to the left-hand side (LHS) of the function signature, i.e. as return values. To illustrate, look at the `H5Dread` function. This function returns data in the `buf` parameter.

```
herr_t H5Dread(hid_t dataset_id, hid_t mem_type_id, hid_t mem_space_id,
              hid_t file_space_id, hid_t xfer_plist_id, void * buf ) /* C syntax */
```

The corresponding MATLAB function changes the output parameter `buf` into a return value. Note that the HDF5 error return is not used. In MATLAB, the nonzero or negative value `herr_t` return values become MATLAB errors. Use MATLAB try-catch statements to handle errors.

```
buf = H5D.read(dataset_id, mem_type_id, mem_space_id, file_space_id, plist_id)
```

String Length Parameters Unnecessary

The length parameter used by some HDF5 library functions to specify the length of string parameters are not necessary in the corresponding MATLAB function. For example, the `H5Aget_name` function in the HDF5 library includes a buffer as an output parameter and the size of the buffer as an input parameter.

```
ssize_t H5Aget_name(hid_t attr_id, size_t buf_size, char *buf ) /* C syntax */
```

The corresponding MATLAB function changes the output parameter `buf` into a return value and drops the `buf_size` parameter:

```
attr_name = H5A.get_name(attr_id)
```


Use Empty Array to Specify NULL

The MATLAB functions use empty arrays ([]) where HDF5 library functions accept the value NULL. For example, the `H5Dfill` function in the HDF5 library accepts the value NULL in place of a specified fill value.

```
herr_t H5Dfill(const void *fill, hid_t fill_type_id, void *buf,
              hid_t buf_type_id, hid_t space_id ) /* C syntax */
```

When using the corresponding MATLAB function, you can specify an empty array ([]) instead of NULL.

Specifying Multiple Constants

Some functions in the HDF5 library require you to specify an array of constants. For example, in the `H5Screate_simple` function, if you want to specify that each dimension in the data space can be unlimited, you use the constant `H5S_UNLIMITED` for each dimension in `maxdims`. In MATLAB, because you pass constants as text strings, you must use a cell array to achieve the same result. The following code fragment provides an example of using a cell array to specify this constant for each dimension of this data space.

```
ds_id = H5S.create_simple(2,[3 4],{'H5S_UNLIMITED' 'H5S_UNLIMITED'});
```

Mapping Between HDF5 Data Types and MATLAB Data Types. When the HDF5 low-level functions read data from an HDF5 file or write data to an HDF5 file, the functions map HDF5 data types to MATLAB data types automatically.

For *atomic* data types, such as commonly used binary formats for numbers (integers and floating point) and characters (ASCII), the mapping is typically straightforward because MATLAB supports similar types. See the table Mapping Between HDF5 Atomic Data Types and MATLAB® Data Types on page 2-38 for a list of these mappings.

Mapping Between HDF5 Atomic Data Types and MATLAB Data Types

HDF5 Atomic Data Type	MATLAB Data Type
Bit-field	Array of packed 8-bit integers
Float	MATLAB single and double types, provided that they occupy 64 bits or fewer
Integer types, signed and unsigned	Equivalent MATLAB integer types, signed and unsigned
Opaque	Array of uint8 values
Reference	Array of uint8 values
String	MATLAB character arrays.

For *composite* data types, such as aggregations of one or more atomic data types into structures, multidimensional arrays, and variable-length data types (one-dimensional arrays), the mapping is sometimes ambiguous with reference to the HDF5 data type. In HDF5, a 5-by-5 data set containing a single uint8 value in each element is distinct from a 1-by-1 data set containing a 5-by-5 array of uint8 values. In the first case, the data set contains 25 observations of a single value; in the second case, the data set contains a single observation with 25 values. In MATLAB both of these data sets are represented by a 5-by-5 matrix.

If your data is a complex data set, you might need to create HDF5 data types directly to make sure you have the mapping you intend. See the table Mapping Between HDF5 Composite Data Types and MATLAB® Data Types on page 2-39 for a list of the default mappings. You can specify the data type when you write data to the file using the `H5Dwrite` function. See the HDF5 data type interface documentation for more information.

Mapping Between HDF5 Composite Data Types and MATLAB Data Types

HDF5 Composite Data Type	MATLAB Data Type
Array	Extends the dimensionality of the data type which it contains. For example, an array of an array of integers in HDF5 would map onto a two dimensional array of integers in MATLAB.
Compound	MATLAB structure. Note: All structures representing HDF5 data in MATLAB are scalar.
Enumeration	Array of integers which each have an associated name
Variable Length	MATLAB 1-D cell arrays

Reporting Data Set Dimensions

The MATLAB low-level HDF5 functions report data set dimensions and the shape of data sets differently than the MATLAB high-level functions. For ease of use, the MATLAB high-level functions report data set dimensions consistent with MATLAB column-major indexing. To be consistent with the HDF5 library, and to support the possibility of nested data sets and complicated data types, the MATLAB low-level functions report array dimensions using the C row-major orientation.

Example: Using the MATLAB HDF5 Low-level Functions. This example shows how to use the MATLAB HDF5 low-level functions to write a data set to an HDF5 file and then read the data set from the file.

- 1 Create the MATLAB variable that you want to write to the HDF5 file. The examples creates a two-dimensional array of uint8 data.

```
testdata = [1 3 5; 2 4 6];
```

- 2 Create the HDF5 file or open an existing file. The example creates a new HDF5 file, named `my_file.h5`, in the system temp directory.

```
filename = fullfile(tempdir,'my_file.h5');
```

```
fileID = H5F.create(filename, 'H5F_ACC_TRUNC', 'H5P_DEFAULT', 'H5P_DEFAULT');
```

In HDF5, you use the `H5Fcreate` function to create a file. The example uses the MATLAB equivalent, `H5F.create`. As arguments, specify the name you want to assign to the file, the type of access you want to the file ('`H5F_ACC_TRUNC`' in the example), and optional additional characteristics specified by a file creation property list and a file access property list. This example uses default values for these property lists ('`H5P_DEFAULT`'). In the example, note how the C constants are passed to the MATLAB functions as strings. The function returns an ID to the HDF5 file.

- 3** Create the data set in the file to hold the MATLAB variable. In the HDF5 programming model, you must define the data type and dimensionality (data space) of the data set as separate entities.

- a** Specify the data type used by the data set. In HDF5, you use the `H5Tcopy` function to create integer or floating-point data types. The example uses the corresponding MATLAB function, `H5T.copy`, to create a double data type because the MATLAB data is double. The function returns a data type ID.

```
datatypeID = H5T.copy('H5T_NATIVE_DOUBLE');
```

- b** Specify the dimensions of the data set. In HDF5, you use the `H5Screate_simple` routine to create a data space. The example uses the corresponding MATLAB function, `H5S.create_simple`, to specify the dimensions. The function returns a data space ID.

Because HDF5 stores data in row-major order and the MATLAB array is organized in column-major order, you should reverse the ordering of the dimension extents before using `H5Screate_simple` to preserve the layout of the data. You can use `fliplr` for this purpose. For a list of other HDF5 functions that require dimension flipping, see “Preserving the Correct Layout of Your Data” on page 2-43.

```
dims = size(testdata);  
dataspaceID = H5S.create_simple(2, fliplr(dims), []);
```

Other software programs that use row-major ordering (such as H5DUMP from the HDF Group) may report the size of the dataset to be 3-by-2 instead of 2-by-3.

- c Create the data set. In HDF5, you use the H5Dcreate routine to create a data set. The example uses the corresponding MATLAB function, H5D.create, specifying the file ID, the name you want to assign to the data set, data type ID, the data space ID, and a data set creation property list ID as arguments. The example uses the defaults for the property lists. The function returns a data set ID.

```
dsetname = 'my_dataset';
datasetID = H5D.create(fileID,dsetname,datatypeID,dataspaceID,'H5P_DEFAULT');
```

Note To write a large data set, you must use the chunking capability of the HDF5 library. To do this, create a property list and use the H5P.set_chunk function to set the chunk size in the property list. In the following example, the dimensions of the data set are `dims = [2^16 2^16]` and the chunk size is 1024-by-1024. You then pass the property list as the last argument to the data set creation function, H5D.create, instead of using the H5P_DEFAULT value.

```
plistID = H5P.create('H5P_DATASET_CREATE'); % create property list

chunk_size = min([1024 1024], dims); % define chunk size
H5P.set_chunk(plistID, chunk_size); % set chunk size in property list

datasetID = H5D.create(fileID, dsetname, datatypeID, dataspaceID, plistID);
```

- 4 Write the data to the data set. In HDF5, you use the H5Dwrite routine to write data to a data set. The example uses the corresponding MATLAB function, H5D.write, specifying as arguments the data set ID, the memory data type ID, the memory space ID, the data space ID, the transfer property list ID and the name of the MATLAB variable to be written to the data set.

You can use the memory data type to specify the data type used to represent the data in the file. The example uses the constant 'H5ML_DEFAULT' which lets the MATLAB function do an automatic mapping to HDF5 data types.

The memory data space ID and the data set's data space ID specify to write subsets of the data set to the file. The example uses the constant 'H5S_ALL' to write all the data to the file and uses the default property list.

```
H5D.write(datasetID, 'H5ML_DEFAULT', 'H5S_ALL', 'H5S_ALL', ...  
         'H5P_DEFAULT', testdata);
```

If you had not reversed the ordering of the dimension extents in step 3b above, you would have been required to permute the MATLAB array before using `H5D.write`, which could result in an enormous performance penalty.

- 5** Close the data set, data space, data type, and file objects. If used inside a MATLAB function, these identifiers are closed automatically when they go out of scope.

```
H5D.close(datasetID);  
H5S.close(dataspaceID);  
H5T.close(datatypeID);  
H5F.close(fileID);
```

- 6** To read the data set you wrote to the file, you must open the file. In HDF5, you use the `H5Fopen` routine to open an HDF5 file, specifying the name of the file, the access mode, and a property list as arguments. The example uses the corresponding MATLAB function, `H5F.open`, opening the file for read-only access.

```
fileID = H5F.open(filename, 'H5F_ACC_RDONLY', 'H5P_DEFAULT');
```

- 7** After opening the file, you must open the data set. In HDF5, you use the `H5Dopen` function to open a data set. The example uses the corresponding MATLAB function, `H5D.open`, specifying as arguments the file ID and the name of the data set, defined earlier in the example.

```
datasetID = H5D.open(fileID, dsetname);
```

- 8** After opening the data set, you can read the data into the MATLAB workspace. In HDF5, you use the `H5Dread` function to read an HDF5 file. The example uses the corresponding MATLAB function, `H5D.read`, specifying as arguments the data set ID, the memory data type ID, the memory space ID, the data space ID, and the transfer property list ID.

```
returned_data = H5D.read(datasetID, 'H5ML_DEFAULT', ...  
                        'H5S_ALL', 'H5S_ALL', 'H5P_DEFAULT');
```

You can compare the original MATLAB variable, `testdata`, with the variable just created, `data`, to see if they are the same.

Preserving the Correct Layout of Your Data. When you use any of the following functions that deal with dataspace, you should flip dimension extents to preserve the correct layout of the data, as illustrated in step 3b in “Example: Using the MATLAB HDF5 Low-level Functions” on page 2-39.

- `H5D.set_extent`
- `H5P.get_chunk`
- `H5P.set_chunk`
- `H5S.create_simple`
- `H5S.get_simple_extent_dims`
- `H5S.select_hyperslab`
- `H5T.array_create`
- `H5T.get_array_dims`

Exporting to Hierarchical Data Format (HDF4) Files

Hierarchical Data Format (HDF4) is a general-purpose, machine-independent standard for storing scientific data in files, developed by the National Center for Supercomputing Applications (NCSA). For more information about these file formats, read the HDF documentation at the HDF Web site (www.hdfgroup.org).

HDF-EOS is an extension of HDF4 that was developed by the National Aeronautics and Space Administration (NASA) for storage of data returned from the Earth Observing System (EOS). For more information about this extension to HDF4, see the HDF-EOS documentation at the NASA Web site (www.hdfeos.org).

This section describes how to use MATLAB functions to access the HDF4 Application Programming Interfaces (APIs). These APIs are libraries of C routines. To import or export data, you must use the functions in the HDF4

API associated with the particular HDF4 data type you are working with. Each API has a particular programming model, that is, a prescribed way to use the routines to write data sets to the file. To illustrate this concept, this section describes the programming model of one particular HDF4 API: the HDF4 Scientific Data (SD) API. For a complete list of the HDF4 APIs supported by MATLAB and the functions you use to access each one, see the `hdf` reference page.

Note This section does not attempt to describe all HDF4 features and routines. To use the MATLAB HDF4 functions effectively, you must refer to the official NCSA documentation at the HDF Web site (www.hdfgroup.org).

- “Mapping HDF4 to MATLAB Syntax” on page 2-44
- “Step 1: Creating an HDF4 File” on page 2-45
- “Step 2: Creating an HDF4 Data Set” on page 2-46
- “Step 3: Writing MATLAB Data to an HDF4 File” on page 2-47
- “Step 4: Writing Metadata to an HDF4 File” on page 2-50
- “Step 5: Closing HDF4 Data Sets” on page 2-51
- “Step 6: Closing an HDF4 File” on page 2-52
- “Using the MATLAB HDF4 Utility API” on page 2-52

Mapping HDF4 to MATLAB Syntax

Each HDF4 API includes many individual routines that you use to read data from files, write data to files, and perform other related functions. For example, the HDF4 Scientific Data (SD) API includes separate C routines to open (`SDopen`), close (`SDend`), and read data (`SDreaddata`).

Instead of supporting each routine in the HDF4 APIs, MATLAB provides a single function that serves as a gateway to all the routines in a particular HDF4 API. For example, the HDF Scientific Data (SD) API includes the C routine `SDend` to close an HDF4 file:

```
status = SDend(sd_id); /* C code */
```


To call this routine from MATLAB, use the MATLAB function associated with the SD API, `hdfsd`. You must specify the name of the routine, minus the API acronym, as the first argument and pass any other required arguments to the routine in the order they are expected. For example,

```
status = hdfsd('end',sd_id); % MATLAB code
```

Some HDF4 API routines use output arguments to return data. Because MATLAB does not support output arguments, you must specify these arguments as return values.

For example, the `SDfileinfo` routine returns data about an HDF4 file in two output arguments, `ndatasets` and `nglobal_atts`. Here is the C code:

```
status = SDfileinfo(sd_id, ndatasets, nglobal_atts);
```

To call this routine from MATLAB, change the output arguments into return values:

```
[ndatasets, nglobal_atts, status] = hdfsd('fileinfo',sd_id);
```

Specify the return values in the same order as they appear as output arguments. The function status return value is always specified as the last return value.

Step 1: Creating an HDF4 File

To export MATLAB data in HDF4 format, you must first create an HDF4 file, or open an existing one. In the HDF4 SD API, you use the `SDstart` routine. In MATLAB, use the `hdfsd` function, specifying `start` as the first argument. As other arguments, specify

- A text string specifying the name you want to assign to the HDF4 file (or the name of an existing HDF4 file)
- A text string specifying the HDF4 SD interface file access mode

For example, this code creates an HDF4 file named `mydata.hdf`:

```
sd_id = hdfsd('start','mydata.hdf','DFACC_CREATE');
```

When you specify the `DFACC_CREATE` access mode, `SDstart` creates the file and initializes the HDF4 SD multifile interface, returning an HDF4 SD file identifier, named `sd_id` in the example.

If you specify `DFACC_CREATE` mode and the file already exists, `SDstart` fails, returning `-1`. To open an existing HDF4 file, you must use HDF4 read or write modes. For information about using `SDstart` in these modes, see “Step 1: Opening the HDF4 File” on page 1-96.

Step 2: Creating an HDF4 Data Set

After creating the HDF4 file, or opening an existing one, you must create a data set in the file for each MATLAB array you want to export. If you are writing data to an existing data set, you can skip ahead to the next step.

In the HDF4 SD API, you use the `SDcreate` routine to create data sets. In MATLAB, you use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, 'create' in this case
- Valid HDF4 SD file identifier, `sd_id`, returned by `SDstart`
- Name you want assigned to the data set
- Data type of the data set.
- Number of dimensions in the data set. This is called the *rank* of the data set in HDF4 terminology.
- Size of each dimension, specified as a vector

Once you create a data set, you cannot change its name, data type, or dimensions.

For example, to create a data set in which you can write the following MATLAB 3-by-5 array of doubles,

```
A = [ 1 2 3 4 5 ; 6 7 8 9 10 ; 11 12 13 14 15 ];
```

you could call `hdfsd`, specifying as arguments 'create' and a valid HDF file identifier, `sd_id`. In addition, set the values of the other arguments as in this code fragment:

```

ds_name = 'A';
ds_type = 'double';
ds_rank = ndims(A);
ds_dims = fliplr(size(A));

sds_id = hdfsd('create',sds_id,ds_name,ds_type,ds_rank,ds_dims);

```

If `SDcreate` can successfully create the data set, it returns an HDF4 SD data set identifier, (`sds_id`). Otherwise, `SDcreate` returns -1.

In this example, note the following:

- The data type you specify in `ds_type` must match the data type of the MATLAB array that you want to write to the data set. In the example, the array is of class `double` so the value of `ds_type` is set to `'double'`. If you wanted to use another data type, such as `uint8`, convert the MATLAB array to use this data type,

```
A = uint8([ 1 2 3 4 5 ; 6 7 8 9 10 ; 11 12 13 14 15 ]);
```

and specify the name of the MATLAB data type, `uint8` in this case, in the `ds_type` argument.

```
ds_type = 'uint8';
```

- The code fragment reverses the order of the values in the dimensions argument (`ds_dims`). This processing is necessary because the MATLAB `size` function returns the dimensions in column-major order and HDF4 expects to receive dimensions in row-major order.

Step 3: Writing MATLAB Data to an HDF4 File

After creating an HDF4 file and creating a data set in the file, you can write data to the entire data set or just a portion of the data set. In the HDF4 SD API, you use the `SDwritedata` routine. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `'writedata'` in this case
- Valid HDF4 SD data set identifier, `sds_id`, returned by `SDcreate`

- Location in the data set where you want to start writing data, called the *start* vector in HDF4 terminology
- Number of elements along each dimension to skip between each write operation, called the *stride* vector in HDF4 terminology
- Total number of elements to write along each dimension, called the *edges* vector in HDF4 terminology
- MATLAB array to be written

Note You must specify the values of the *start*, *stride*, and *edges* arguments in row-major order, rather than the column-major order used in MATLAB. Note how the example uses `fliplr` to reverse the order of the dimensions in the vector returned by the `size` function before assigning it as the value of the *edges* argument.

The values you assign to these arguments depend on the MATLAB array you want to export. For example, the following code fragment writes this MATLAB 3-by-5 array of doubles,

```
A = [ 1 2 3 4 5; 6 7 8 9 10; 11 12 13 14 15 ];
```

into an HDF4 file:

```
ds_start = zeros(1:ndims(A)); % Start at the beginning
ds_stride = [];              % Write every element.
ds_edges = fliplr(size(A));  % Reverse the dimensions.

stat = hdfsd('writedata',sds_id,...
             ds_start, ds_stride, ds_edges, A);
```

If it can write the data to the data set, `SDwritedata` returns 0; otherwise, it returns -1.

Note `SDwritedata` queues write operations. To ensure that these queued write operations are executed, you must close the file, using the `SDend` routine. See “Step 6: Closing an HDF4 File” on page 2-52 for more information. As a convenience, MATLAB provides a function, `MLcloseall`, that you can use to close all open data sets and file identifiers with a single call. See “Using the MATLAB HDF4 Utility API” on page 2-52 for more information.

To write less than the entire data set, use the start, stride, and edges vectors to specify where you want to start writing data and how much data you want to write.

For example, the following code fragment uses `SDwritedata` to replace the values of the entire second row of the sample data set:

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
```

with the vector `B`:

```
B = [ 9 9 9 9 9];
```

In the example, the start vector specifies that you want to start the write operation in the first column of the second row. Note how HDF4 uses zero-based indexing and specifies the column dimension first. In MATLAB, you would specify this location as `(2,1)`. The edges argument specifies the dimensions of the data to be written. Note that the size of the array of data to be written must match the edge specification.

```
ds_start = [0 1]; % Start writing at the first column, second row.
ds_stride = []; % Write every element.
ds_edges = [5 1]; % Each row is a 1-by-5 vector.

stat = hdfsd('writedata',sds_id,ds_start,ds_stride,ds_edges,B);
```

Step 4: Writing Metadata to an HDF4 File

You can optionally include information in an HDF4 file, called attributes, that describes the file and its contents. Using the HDF4 SD API, you can associate attributes with three types of HDF4 objects:

- An entire HDF4 file — File attributes, also called *global* attributes, generally contain information pertinent to all the data sets in the file.
- A data set in an HDF4 file — Data set attributes, also called *local* attributes, describe individual data sets.
- A dimension of a data set — Dimension attributes provide information about one particular dimension of a data set.

To create an attribute in the HDF4 SD API, use the `SDsetattr` routine. In MATLAB, use the `hdfsd` function, specifying `'setattr'` as the first argument. As other arguments, specify

- A valid HDF4 SD identifier associated with the object. This value can be a file identifier (`sd_id`), a data set identifier (`sds_id`), or a dimension identifier (`dim_id`).
- A text string that defines the name of the attribute.
- The attribute value.

For example, this code creates a global attribute, named `my_global_attr`, and associates it with the HDF4 file identified by `sd_id`:

```
status = hdfsd('setattr',sd_id,'my_global_attr','my_attr_val');
```

Note In the NCSA documentation, the `SDsetattr` routine has two additional arguments: data type and the number of values in the attribute. When calling this routine from MATLAB, you do not have to include these arguments. The MATLAB HDF4 function can determine the data type and size of the attribute from the value you specify.

The SD interface supports predefined attributes that have reserved names and, in some cases, data types. Predefined attributes are identical to user-defined attributes except that the HDF4 SD API has already defined

their names and data types. For example, the HDF4 SD API defines an attribute, named `cordsys`, in which you can specify the coordinate system used by the data set. Possible values of this attribute include the text strings `'cartesian'`, `'polar'`, and `'spherical'`.

Predefined attributes can be useful because they establish conventions that applications can depend on. The HDF4 SD API supports predefined attributes for data sets and dimensions only; there are no predefined attributes for files. For a complete list of the predefined attributes, see the NCSA documentation.

In the HDF4 SD API, you create predefined attributes the same way you create user-defined attributes, using the `SDsetattr` routine. In MATLAB, use the `hdfsd` function, specifying `setattr` as the first argument:

```
attr_name = 'cordsys';
attr_value = 'polar';

status = hdfsd('setattr',sds_id,attr_name,attr_value);
```

The HDF4 SD API also includes specialized functions for writing and reading the predefined attributes. These specialized functions, such as `SDsetdatastrs`, are sometimes easier to use, especially when you are reading or writing multiple related predefined attributes. You must use specialized functions to read or write the predefined dimension attributes.

You can associate multiple attributes with a single HDF4 object. HDF4 maintains an attribute index for each object. The attribute index is zero-based. The first attribute has index value 0, the second has index value 1, and so on. You access an attribute by its index value.

Each attribute has the format `name=value`, where `name` (called `label` in HDF4 terminology) is a text string up to 256 characters in length and `value` contains one or more entries of the same data type. A single attribute can have multiple values.

Step 5: Closing HDF4 Data Sets

After writing data to a data set in an HDF4 file, you must close access to the data set. In the HDF4 SD API, you use the `SDendaccess` routine to close a data set. In MATLAB, use the `hdfsd` function, specifying `endaccess` as the

first argument. As the only other argument, specify a valid HDF4 SD data set identifier, `sds_id` in this example:

```
stat = hdfsd('endaccess',sds_id);
```

Step 6: Closing an HDF4 File

After writing data to a data set and closing the data set, you must also close the HDF4 file. In the HDF4 SD API, you use the `SDend` routine. In MATLAB, use the `hdfsd` function, specifying `end` as the first argument. As the only other argument, specify a valid HDF4 SD file identifier, `sd_id` in this example:

```
stat = hdfsd('end',sd_id);
```

You must close access to all the data sets in an HDF4 file before closing it.

Note Closing an HDF4 file executes all the write operations that have been queued using `SDwritedata`. As a convenience, the MATLAB HDF Utility API provides a function that can close all open data set and file identifiers with a single call. See “Using the MATLAB HDF4 Utility API” on page 2-52 for more information.

Using the MATLAB HDF4 Utility API

In addition to the standard HDF4 APIs, listed in the `hdfreference` page, MATLAB supports utility functions that are designed to make it easier to use HDF4 in the MATLAB environment.

For example, using the gateway function to the MATLAB HDF4 utility API, `hdfml`, and specifying the name of the `listinfo` routine as an argument, you can view all the currently open HDF4 identifiers. MATLAB updates this list whenever HDF identifiers are created or closed. In the following example only two identifiers are open.

```
hdfml('listinfo')
No open RI identifiers
No open GR identifiers
No open grid identifiers
No open grid file identifiers
```



```
No open annotation identifiers
No open AN identifiers
Open scientific dataset identifiers:
  262144
Open scientific data file identifiers:
  393216
No open Vdata identifiers
No open Vgroup identifiers
No open Vfile identifiers
No open point identifiers
No open point file identifiers
No open swath identifiers
No open swath file identifiers
No open access identifiers
No open file identifiers
```

Closing All Open HDF4 Identifiers. To close all the currently open HDF4 identifiers in a single call, use the gateway function to the MATLAB HDF4 utility API, `hdfml`, specifying the name of the `closeall` routine as an argument. The following example closes all the currently open HDF4 identifiers.

```
hdfml('closeall')
```

Exporting to Images

To export data from the MATLAB workspace using one of the standard graphics file formats, use the `imwrite` function. Using this function, you can export data in formats such as the Tagged Image File Format (TIFF), Joint Photographic Experts Group (JPEG), and Portable Network Graphics (PNG). For a complete list of supported formats, see the `imwrite` reference page.

The following example writes a multidimensional array of `uint8` data `I` from the MATLAB workspace into a file in TIFF format. The class of the output image written to the file depends on the format specified. For most formats, if the input array is of class `uint8`, `imwrite` outputs the data as 8-bit values. See the `imwrite` reference page for details.

```
whos I
      Name      Size              Bytes  Class
      I         650x600x3          1170000  uint8 array

Grand total is 1170000 elements using 1170000 bytes
imwrite(I, 'my_graphics_file.tif','tif');
```

Note `imwrite` supports different syntaxes for several of the standard formats. For example, with TIFF file format, you can specify the type of compression MATLAB uses to store the image. See the `imwrite` reference page for details.

For more control writing data to a TIFF file, use the `Tiff` object—see “Exporting Image Data and Metadata to TIFF Files” on page 2-54 for more information.

Exporting Image Data and Metadata to TIFF Files

While you can use `imwrite` to export image data and metadata (tags) to Tagged Image File Format (TIFF) files, the function does have some limitations. For example, when you want to modify image data or metadata in the file, you must write the all the data to the file. You cannot write only the updated portion. Using the `Tiff` object, you can write portions of the image data and modify or add individual tags to a TIFF file. When you construct a

`Tiff` object, it represents your connection with a TIFF file and provides access to many of the routines in the LibTIFF library.

The following sections provide step-by-step examples of using `Tiff` object methods and properties to perform some common tasks with TIFF files. To get the most out of the `Tiff` object, you must be familiar with the LibTIFF version 3.7.1 API, as well as the TIFF specification and technical notes. View this documentation at [LibTIFF - TIFF Library and Utilities](#)

Creating a New TIFF File

- 1 Create some image data. This example reads image data from a JPEG file included with MATLAB:

```
imgdata = imread('ngc6543a.jpg');
```

- 2 Create a new TIFF file by constructing a `Tiff` object, specifying the name of the new file as an argument. To create a file you must specify either write mode ('w') or append mode ('a'):

```
t = Tiff('myfile.tif','w');
```

When you create a new TIFF file, the `Tiff` constructor creates a file containing an image file directory (IFD). A TIFF file uses this IFD to organize all the data and metadata associated with a particular image. A TIFF file can contain multiple IFDs. The `Tiff` object makes the IFD it creates the *current* IFD. `Tiff` object methods operate on the current IFD. You can navigate among IFDs in a TIFF file and specify which IFD is the current IFD using `Tiff` object methods.

- 3 Set required TIFF tags using the `setTag` method of the `Tiff` object. These required tags specify information about the image, such as its length and width. To break the image data into strips, specify a value for the `RowsPerStrip` tag. To break the image data into tiles, specify values for the `TileWidth` and `TileLength` tags. The example creates a structure that contains tag names and values and passes that to `setTag`. You also can set each tag individually.

```
tagstruct.ImageLength = size(imgdata,1)
tagstruct.ImageWidth = size(imgdata,2)
tagstruct.Photometric = Tiff.Photometric.RGB
```

```
tagstruct.BitsPerSample = 8
tagstruct.SamplesPerPixel = 3
tagstruct.RowsPerStrip = 16
tagstruct.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky
tagstruct.Software = 'MATLAB'
t.setTag(tagstruct)
```

For information about supported TIFF tags and how to set their values, see “Setting Tag Values” on page 2-61. For example, the `Tiff` object supports properties that you can use to set the values of certain properties. This example uses the `Tiff` object `PlanarConfiguration` property to specify the correct value for the chunky configuration: `Tiff.PlanarConfiguration.Chunky`.

- 4 Write the image data and metadata to the current directory using the `write` method of the `Tiff` object.

```
t.write(imgdata);
```

If you wanted to put multiple images into your file, call the `writeDirectory` method right after performing this write operation. The `writeDirectory` method sets up a new image file directory in the file and makes this new directory the current directory.

- 5 Close your connection to the file by closing the `Tiff` object:

```
t.close();
```

- 6 Test that you created a valid TIFF file by using the `imread` function to read the file, and then display the image:

```
imagesc(imread('myfile.tif'));
```

Writing a Strip or Tile of Image Data

Note You can only modify a strip or a tile of image data if the data is not compressed.

- 1 Open an existing TIFF file for modification by creating a `Tiff` object. This example uses the file created in “Creating a New TIFF File” on page 2-55. The `Tiff` constructor returns a handle to a `Tiff` object.

```
t = Tiff('myfile.tif','r');
```

- 2 Generate some data to write to a strip in the image. This example creates a three-dimensional array of zeros that is the size of a strip. The code uses the number of rows in a strip, the width of the image, and the number of samples per pixel as dimensions. The array is an array of `uint8` values.

```
width = t.getTag('ImageWidth');  
height = t.getTag('RowsPerStrip');  
numSamples = t.getTag('SamplesPerPixel');  
stripData = zeros(height,width,numSamples,'uint8');
```

If the image data had a tiled layout, you would use the `TileWidth` and `TileLength` tags to specify the dimensions.

- 3 Write the data to a strip in the file using the `writeEncodedStrip` method. Specify the index number that identifies the strip you want to modify. The example picks strip 18 because it is easier to see the change in the image.

```
t.writeEncodedStrip(18, stripData);
```

If the image had a tiled layout, you would use the `writeEncodedTile` method to modify the tile.

- 4 Close your connection to the file by closing the `Tiff` object.

```
t.close();
```

- 5 Test that you modified a strip of the image in the TIFF file by using the `imread` function to read the file, and then display the image.

```
modified_imgdata = imread('myfile.tif');  
imagesc(modified_imgdata)
```

Note the black strip across the middle of the image.

Modifying TIFF File Metadata (Tags)

- 1 Open an existing TIFF file for modification using the `Tiff` object. This example uses the file created in “Creating a New TIFF File” on page 2-55. The `Tiff` constructor returns a handle to a `Tiff` object.

```
t = Tiff('myfile.tif', 'r');
```

- 2 Verify that the file does not contain the `Artist` tag, using the `getTag` method. This code should issue an error message saying that it was unable to retrieve the tag.

```
artist_value = t.getTag('Artist');
```

- 3 Add the `Artist` tag using the `setTag` method.

```
t.setTag('Artist', 'Pablo Picasso');
```

- 4 Write the new tag data to the TIFF file using the `rewriteDirectory` method. Use the `rewriteDirectory` method when modifying existing metadata in a file or adding new metadata to a file.

```
t.rewriteDirectory();
```

- 5 Close your connection to the file by closing the `Tiff` object.

```
t.close();
```

- 6 Test your work by reopening the TIFF file and getting the value of the `Artist` tag, using the `getTag` method.

```
t = Tiff('myfile.tif', 'r');
```

```
t.getTag('Artist')
```

```
ans =
```

```
Pablo Picasso
```

```
t.close();
```

Creating Subdirectories in a TIFF File

- 1 Create some image data. This example reads image data from a JPEG file included with MATLAB. The example then creates two reduced-resolution (thumbnail) versions of the image data.

```
imgdata = imread('ngc6543a.jpg');
%
% Reduce number of pixels by a half.
img_half = imgdata(1:2:end,1:2:end,:);
%
% Reduce number of pixels by a third.
img_third = imgdata(1:3:end,1:3:end,:);
```

- 2 Create a new TIFF file by constructing a `Tiff` object and specifying the name of the new file as an argument. To create a file you must specify either write mode ('w') or append mode ('a'). The `Tiff` constructor returns a handle to a `Tiff` object.

```
t = Tiff('my_subimage_file.tif','w');
```

- 3 Set required TIFF tags using the `setTag` method of the `Tiff` object. These required tags specify information about the image, such as its length and width. To break the image data into strips, specify a value for the `RowsPerStrip` tag. To break the image data into tiles, use the `TileWidth` and `TileLength` tags. The example creates a structure that contains tag names and values and passes that to `setTag`. You can also set each tag individually.

To create subdirectories, you must set the `SubIFD` tag, specifying the number of subdirectories you want to create. Note that the number you specify isn't the value of the `SubIFD` tag. The number tells the `Tiff` software to create a `SubIFD` that points to two subdirectories. The actual value of the `SubIFD` tag will be the byte offsets of the two subdirectories.

```
tagstruct.ImageLength = size(imgdata,1)
tagstruct.ImageWidth = size(imgdata,2)
tagstruct.Photometric = Tiff.Photometric.RGB
tagstruct.BitsPerSample = 8
tagstruct.SamplesPerPixel = 3
tagstruct.RowsPerStrip = 16
```

```
tagstruct.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky
tagstruct.Software = 'MATLAB'
tagstruct.SubIFD = 2 % required to create subdirectories
t.setTag(tagstruct)
```

For information about supported TIFF tags and how to set their values, see “Setting Tag Values” on page 2-61. For example, the `Tiff` object supports properties that you can use to set the values of certain properties. This example uses the `Tiff` object `PlanarConfiguration` property to specify the correct value for the chunky configuration: `Tiff.PlanarConfiguration.Chunky`.

- 4** Write the image data and metadata to the current directory using the `write` method of the `Tiff` object.

```
t.write(imgdata);
```

- 5** Set up the first subdirectory by calling the `writeDirectory` method. The `writeDirectory` method sets up the subdirectory and make the new directory the current directory. Because you specified that you wanted to create two subdirectories, `writeDirectory` sets up a subdirectory.

```
t.writeDirectory();
```

- 6** Set required tags, just as you did for the regular directory. According to the LibTIFF API, a subdirectory cannot contain a `SubIFD` tag.

```
tagstruct2.ImageLength = size(img_half,1)
tagstruct2.ImageWidth = size(img_half,2)
tagstruct2.Photometric = Tiff.Photometric.RGB
tagstruct2.BitsPerSample = 8
tagstruct2.SamplesPerPixel = 3
tagstruct2.RowsPerStrip = 16
tagstruct2.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky
tagstruct2.Software = 'MATLAB'
t.setTag(tagstruct2)
```

- 7** Write the image data and metadata to the subdirectory using the `write` method of the `Tiff` object.

```
t.write(img_half);
```


- 8** Set up the second subdirectory by calling the `writeDirectory` method. The `writeDirectory` method sets up the subdirectory and makes it the current directory.

```
t.writeDirectory();
```

- 9** Set required tags, just as you would for any directory. According to the LibTIFF API, a subdirectory cannot contain a `SubIFD` tag.

```
tagstruct3.ImageLength = size(img_third,1)
tagstruct3.ImageWidth = size(img_third,2)
tagstruct3.Photometric = Tiff.Photometric.RGB
tagstruct3.BitsPerSample = 8
tagstruct3.SamplesPerPixel = 3
tagstruct3.RowsPerStrip = 16
tagstruct3.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky
tagstruct3.Software = 'MATLAB'
t.setTag(tagstruct3)
```

- 10** Write the image data and metadata to the subdirectory using the `write` method of the `Tiff` object:

```
t.write(img_third);
```

- 11** Close your connection to the file by closing the `Tiff` object:

```
t.close();
```

Setting Tag Values

The following table lists all the TIFF tags that the `Tiff` object supports and includes information about their MATLAB class and size. For certain tags, the table also indicates the set of values that the `Tiff` object supports, which is a subset of all the possible values defined by the TIFF specification. You can use `Tiff` object properties to specify the supported values for these tags. For example, use `Tiff.Compression.JPEG` to specify JPEG compression. See the `Tiff` class reference page for a full list of properties.

Table 1: Supported TIFF Tags

TIFF Tag	Class	Size	Supported Values	Notes
Artist	char	1xN		
BitsPerSample	double	1x1	1,8,16,32,64	See Table 2
ColorMap	double	256x3	Values should be normalized between 0–1. Stored internally as uint16 values.	Photometric must be Palette
Compression	double	1x1	None: 1 CCITTRLE: 2 CCITTFax3: 3 CCITTFax4: 4 LZW: 5 JPEG: 7 CCITTRLEW: 32771 PackBits: 32773 Deflate: 32946 AdobeDeflate: 8	See Table 3.
Copyright	char	1xN		
DateTime	char	1x19	Return value is padded to 19 chars if required.	
DocumentName	char	1xN		
DotRange	double	1x2		Photometric must be Separated
ExtraSamples	double	1xN	Unspecified: 0 AssociatedAlpha: 1 UnassociatedAlpha: 2	See Table 4.
FillOrder	double	1x1		
GeoAsciiParamsTag	char	1xN		

Table 1: Supported TIFF Tags (Continued)

TIFF Tag	Class	Size	Supported Values	Notes
GeoDoubleParamsTag	double	1xN		
GeoKeyDirectoryTag	double	Nx4		
Group3Options	double	1x1		Compression must be CCITTFax3
Group4Options	double	1x1		Compression must be CCITTFax4
HalfToneHints	double	1x2		
HostComputer	char	1xn		
ICCProfile	uint8	1xn		
ImageDescription	char	1xn		
ImageLength	double	1x1		
ImageWidth	double	1x1		
InkNames	char cell array	1x NumInks		Photometric must be Separated
InkSet	double	1x1	CMYK: 1 MultiInk: 2	Photometric must be Separated
JPEGQuality	double	1x1	A value between 1 and 100	
Make	char	1xn		
MaxSampleValue	double	1x1	0–65,535	
MinSampleValue	double	1x1	0–65,535	
Model	char	1xN		
ModelPixelScaleTag	double	1x3		
ModelTiepointTag	double	Nx6		
ModelTransformationMatrixTag	double	1x16		

Table 1: Supported TIFF Tags (Continued)

TIFF Tag	Class	Size	Supported Values	Notes
NumberOfInks	double	1x1		Must be equal to SamplesPerPixel
Orientation	double	1x1	TopLeft: 1 TopRight: 2 BottomRight: 3 BottomLeft: 4 LeftTop: 5 RightTop: 6 RightBottom: 7 LeftBottom: 8	
PageName	char	1xN		
PageNumber	double	1x2		
Photometric	double	1x1	MinIsWhite: 0 MinIsBlack: 1 RGB: 2 Palette: 3 Mask: 4 Separated: 5 YCbCr: 6 CIELab: 8 ICCLab: 9 ITULab: 10	See Table 2.
Photoshop	uint8	1xN		
PlanarConfiguration	double	1x1	Chunky: 1 Separate: 2	
PrimaryChromaticities	double	1x6		
ReferenceBlackWhite	double	1x6		
ResolutionUnit	double	1x1		

Table 1: Supported TIFF Tags (Continued)

TIFF Tag	Class	Size	Supported Values	Notes
RICTIFFIPTC	uint8	1xN		
RowsPerStrip	double	1x1		
SampleFormat	double	1x1	Uint: 1 Int: 2 IEEEFP: 3	See Table 2
SamplesPerPixel	double	1x1		
SMaxSampleValue	double	1x1	Range of MATLAB data type specified for Image data	
SMinSampleValue	double	1x1	Range of MATLAB data type specified for Image data	
Software	char	1xN		
StripByteCounts	double	1xN		Read-only
StripOffsets	double	1xN		Read-only
SubFileType	double	1x1	Default: 0 ReducedImage: 1 Page: 2 Mask: 4	
SubIFD	double	1x1		
TargetPrinter	char	1xN		
Thresholding	double	1x1	BiLevel: 1 HalfTone: 2 ErrorDiffuse: 3	Photometric can be either: MinIsWhite MinIsBlack

Table 1: Supported TIFF Tags (Continued)

TIFF Tag	Class	Size	Supported Values	Notes
TileByteCounts	double	1xN		Read-only
TileLength	double	1x1	Must be a multiple of 16	
TileOffsets	double	1xN		Read-only
TileWidth	double	1x1	Must be a multiple of 16	
TransferFunction	double	See note ¹	Each value should be within 0–2 ¹⁶ -1	SamplePerPixel can be either 1 or 3
WhitePoint	double	1x2		Photometric can be: RGB Palette YCbCr CIE Lab ICCLab ITULab
XMP	char	1xn		N>5
XPosition	double	1x1		
XResolution	double	1x1		
YCbCrCoefficients	double	1x3		Photometric must be YCbCr
YCbCrPositioning	double	1x1	Centered: 1 Cosited: 2	Photometric must be YCbCr
YCbCrSubSampling	double	1x2		Photometric must be YCbCr
YPosition	double	1x1		
YResolution	double	1x1		
ZipQuality	double	1x1	Value between 1 and 9	

¹Size is $1 \times 2^{\text{BitsPerSample}}$ or $3 \times 2^{\text{BitsPerSample}}$.

Table 2: Valid SampleFormat Values for BitsPerSample Settings

BitsPerSample	SampleFormat	MATLAB Data Type
1	Uint	logical
8	Uint, Int	uint8, int8
16	Uint, Int	uint16, int16
32	Uint, Int, IEEEFP	uint32, int32, single
64	IEEEFP	double

Table 3: Valid SampleFormat Values for BitsPerSample and Photometric Combinations

Photometric Values	BitsPerSample Values				
	1	8	16	32	64
MinIsWhite	Uint	Uint/Int	Uint Int	Uint Int IEEEFP	IEEEFP
MinIsBlack	Uint	Uint/Int	Uint Int	Uint Int IEEEFP	IEEEFP
RGB		Uint	Uint	Uint IEEEFP	IEEEFP
Palette		Uint			
Mask	Uint				
Separated		Uint	Uint	Uint IEEEFP	IEEEFP
YCbCr		Uint	Uint	Uint IEEEFP	IEEEFP
CIELab		Uint	Uint		
ICCLab		Uint	Uint		
ITULab		Uint	Uint		

Table 4: Valid SampleFormat Values for BitsPerSample and Compression Combinations

Compression Values	BitsPerSample Values				
	1	8	16	32	64
None	Uint	Uint Int	Uint Int	Uint Int IEEEFP	IEEEFP
CCITTRLE	Uint				
CCITTFax3	Uint				
CCITTFax4	Uint				
LZW	Uint	Uint Int	Uint Int	Uint Int IEEEFP	IEEEFP
JPEG		Uint Int			
CCITTRLEW	Uint				
PackBits	Uint	Uint Int	Uint Int	Uint Int IEEEFP	IEEEFP
Deflate	Uint	Uint Int	Uint Int	Uint Int IEEEFP	IEEEFP
AdobeDeflate	Uint	Uint Int	Uint Int	Uint Int IEEEFP	IEEEFP

Table 5: Valid SamplesPerPixel Values for Photometric Settings

Photometric Values	SamplesPerPixel ¹
MinIsWhite	1+
MinIsBlack	1+
RGB	3+

Table 5: Valid SamplesPerPixel Values for Photometric Settings (Continued)

Photometric Values	SamplesPerPixel¹
Palette	1
Mask	1
Separated	1+
YCbCr	3
CIELab	3+
ICCLab	3+
ITULab	3+

¹ When you specify more than the expected number of samples per pixel (n+), you must set the ExtraSamples tag accordingly.

Exporting to Audio and Video

In this section...
“Exporting to Audio Files” on page 2-70
“Exporting Video to AVI Files” on page 2-70

Exporting to Audio Files

In MATLAB, audio data is simply numeric data that you can export using standard MATLAB data export functions, such as `save`.

MATLAB also includes functions that write audio data to files in specific file formats:

- `auwrite` — Exports sound data in AU file format
- `wavwrite` — Exports sound data in WAV file format

Exporting Video to AVI Files

To create an Audio/Video Interleaved (AVI) file from MATLAB graphics animations or from still images, follow these steps:

- 1 Create a `VideoWriter` object by calling the `VideoWriter` function. For example:

```
myVideo = VideoWriter('myfile.avi');
```

By default, `VideoWriter` prepares to create an AVI file using Motion JPEG compression. To create an uncompressed file, specify the Uncompressed AVI profile, as follows:

```
myVideo = VideoWriter('myfile.avi', 'Uncompressed AVI');
```

- 2 Optionally, adjust the frame rate (number of frames to display per second) or the quality setting (a percentage from 0 through 100). For example:

```
myVideo.FrameRate = 15; % Default 30  
myVideo.Quality = 50; % Default 75
```

Note Quality settings only apply to compressed files. Higher quality settings result in higher video quality, but also increase the file size. Lower quality settings decrease the file size and video quality.

3 Open the file:

```
open(myVideo);
```

Note After you call `open`, you cannot change the frame rate or quality settings.

4 Write frames, still images, or an existing MATLAB movie to the file by calling `writeVideo`. For example, suppose that you have created a MATLAB movie called `myMovie`. Write your movie to a file:

```
writeVideo(myVideo, myMovie);
```

Alternatively, `writeVideo` accepts single frames or arrays of still images as the second input argument. For more information, see the `writeVideo` reference page.

5 Close the file:

```
close(myVideo);
```

Exporting Binary Data with Low-Level I/O

Low-level file I/O functions allow the most direct control over reading or writing data to a file. However, these functions require that you specify more detailed information about your file than the easier-to-use *high-level functions*. For a complete list of high-level functions and the file formats they support, see the `file formats` reference page.

If the high-level functions cannot export your data, use one of the following:

- `fprintf`, which writes formatted data to a text or ASCII file; that is, a file you can view in a text editor or import into a spreadsheet. For more information, see “Writing to Text Data Files with Low-Level I/O” on page 2-12.
- `fwrite`, which writes a stream of binary data to a file. For more information, see the following sections:
 - “Writing Binary Data to a File” on page 2-72
 - “Overwriting or Appending to an Existing File” on page 2-73
 - “Creating a File for Use on a Different System” on page 2-75
 - “Opening Files with Different Character Encodings” on page 2-76
 - “Writing and Reading Complex Numbers” on page 2-77

Note The low-level file I/O functions are based on functions in the ANSI Standard C Library. However, MATLAB includes *vectorized* versions of the functions, to read and write data in an array with minimal control loops.

Writing Binary Data to a File

As with any of the low-level I/O functions, before exporting, open or create a file with `fopen`, and obtain a file identifier. When you finish processing a file, close it with `fclose(fid)`.

By default, `fwrite` writes values from an array in column order as 8-bit unsigned integers (`uint8`).

For example, create a file `nine.bin` with the integers from 1 to 9:

```
fid = fopen('nine.bin','w');
fwrite(fid, [1:9]);
fclose(fid);
```

If the values in your matrix are not 8-bit unsigned integers, specify the precision of the values. For example, to create a file with double-precision values:

```
mydata = [pi, 42, 1/3];

fid = fopen('double.bin','w');
fwrite(fid, mydata, 'double');
fclose(fid);
```

For a complete list of precision descriptions, see the `fwrite` function reference page.

Overwriting or Appending to an Existing File

By default, `fopen` opens files with read access. To change the type of file access, use the permission string in the call to `fopen`. Possible permission strings include:

- `r` for reading
- `w` for writing, discarding any existing contents of the file
- `a` for appending to the end of an existing file

To open a file for both reading and writing or appending, attach a plus sign to the permission, such as `'w+'` or `'a+'`. For a complete list of permission values, see the `fopen` reference page.

Note If you open a file for both reading and writing, you must call `fseek` or `frewind` between read and write operations.

When you open a file, MATLAB creates a pointer to indicate the current position within the file. To read or write selected portions of data, move this

pointer to any location in the file. For more information, see “Moving within a File” on page 1-120.

Example – Overwriting Binary Data in an Existing File

Create a file `magic4.bin` as follows, specifying permission to write and read:

```
fid = fopen('changing.bin','w+');  
fwrite(fid,magic(4));
```

The original `magic(4)` matrix is:

16	2	3	13
5	11	10	8
9	7	6	12
4	14	15	1

The file contains 16 bytes, 1 for each value in the matrix. Replace the second set of four values (the values in the second column of the matrix) with the vector `[44 44 44 44]`:

```
% fseek to the fourth byte after the beginning of the file  
fseek(fid, 4, 'bof');  
  
%write the four values  
fwrite(fid,[44 44 44 44]);  
  
% read the results from the file into a 4-by-4 matrix  
frewind(fid);  
newdata = fread(fid, [4,4])  
  
% close the file  
fclose(fid);
```

The `newdata` in the file `changing.bin` is:

16	44	3	13
5	44	10	8
9	44	6	12
4	44	15	1

Example — Appending Binary Data to an Existing File

Add the values [55 55 55 55] to the end of the `changing.bin` file created in the previous example.

```
% open the file to append and read
fid = fopen('changing.bin','a+');

% write values at end of file
fwrite(fid,[55 55 55 55]);

% read the results from the file into a 4-by-5 matrix
frewind(fid);
appended = fread(fid, [4,5])

% close the file
fclose(fid);
```

The appended data in the file `changing.bin` is:

16	44	3	13	55
5	44	10	8	55
9	44	6	12	55
4	44	15	1	55

Creating a File for Use on a Different System

Different operating systems store information differently at the byte or bit level:

- *Big-endian* systems store bytes starting with the largest address in memory (that is, they start with the big end).
- *Little-endian* systems store bytes starting with the smallest address (the little end).

Windows systems use little-endian byte ordering, and UNIX systems use big-endian byte ordering.

To create a file for use on an opposite-endian system, specify the byte ordering for the target system. You can specify the ordering in the call to open the file, or in the call to write the file.

For example, to create a file named `myfile.bin` on a big-endian system for use on a little-endian system, use one (or both) of the following commands:

- Open the file with

```
fid = fopen('myfile.bin', 'w', 'l')
```

- Write the file with

```
fwrite(fid, mydata, precision, 'l')
```

where `'l'` indicates little-endian ordering.

If you are not sure which byte ordering your system uses, call the `computer` function:

```
[cinfo, maxsize, ordering] = computer
```

The returned `ordering` is `'L'` for little-endian systems, or `'B'` for big-endian systems.

Opening Files with Different Character Encodings

Encoding schemes support the characters required for particular alphabets, such as those for Japanese or European languages. Common encoding schemes include US-ASCII or UTF-8.

The encoding scheme determines the number of bytes required to read or write `char` values. For example, US-ASCII characters always use 1 byte, but UTF-8 characters use up to 4 bytes. MATLAB automatically processes the required number of bytes for each `char` value based on the specified encoding scheme. However, if you specify a `uchar` precision, MATLAB processes each byte as `uint8`, regardless of the specified encoding.

If you do not specify an encoding scheme, `fopen` opens files for processing using the default encoding for your system. To determine the default, open a file, and call `fopen` again with the syntax:

```
[filename, permission, machineformat, encoding] = fopen(fid);
```

If you specify an encoding scheme when you open a file, the following functions apply that scheme: `fscanf`, `fprintf`, `fgetl`, `fgets`, `fread`, and `fwrite`.

For a complete list of supported encoding schemes, and the syntax for specifying the encoding, see the `fopen` reference page.

Writing and Reading Complex Numbers

The available precision values for `fwrite` do not explicitly support complex numbers. To store complex numbers in a file, separate the real and imaginary components and write them separately to the file.

After separating the values, write all real components followed by all imaginary components, or interleave the components. Use the method that allows you to read the data in your target application.

For example, consider the following set of complex numbers:

```
nrows = 5;
ncols = 5;
z = complex(rand(nrows, ncols), rand(nrows, ncols));

% Divide into real and imaginary components
z_real = real(z);
z_imag = imag(z);
```

One approach: write all the real components, followed by all the imaginary components:

```
adjacent = [z_real z_imag];

fid = fopen('complex_adj.bin', 'w');
fwrite(fid, adjacent, 'double');
fclose(fid);

% To read these values back in, so that:
%   same_real = z_real
%   same_imag = z_imag
%   same_z = z

fid = fopen('complex_adj.bin');
same_real = fread(fid, [nrows, ncols], 'double');
same_imag = fread(fid, [nrows, ncols], 'double');
```

```
fclose(fid);

same_z = complex(same_real, same_imag);
```

An alternate approach: interleave the real and imaginary components for each value. `fwrite` writes values in column order, so build an array that combines the real and imaginary parts by alternating rows.

```
% Preallocate the interleaved array
interleaved = zeros(nrows*2, ncols);

% Alternate real and imaginary data
newrow = 1;
for row = 1:nrows
    interleaved(newrow,:) = z_real(row,:);
    interleaved(newrow + 1,:) = z_imag(row,:);
    newrow = newrow + 2;
end

% Write the interleaved values
fid = fopen('complex_int.bin','w');
fwrite(fid, interleaved, 'double');
fclose(fid);

% To read these values back in, so that:
%   same_real = z_real
%   same_imag = z_imag
%   same_z = z
% Use the skip parameter in fread (double = 8 bytes)

fid = fopen('complex_int.bin');
same_real = fread(fid, [nrows, ncols], 'double', 8);

% Return to the first imaginary value in the file
fseek(fid, 8, 'bof');
same_imag = fread(fid, [nrows, ncols], 'double', 8);
fclose(fid);

same_z = complex(same_real, same_imag);
```

Creating Temporary Files

The `tempdir` and `tempname` functions assist in locating temporary data on your system.

Function	Purpose
<code>tempdir</code>	Get temporary folder name.
<code>tempname</code>	Get temporary filename.

Use these functions to create temporary files. Some systems delete temporary files every time you reboot the system. On other systems, designating a file as temporary can mean only that the file is not backed up.

The `tempdir` function returns the name of the folder that has been designated to hold temporary files on your system. For example, issuing `tempdir` on The Open Group UNIX systems returns the `/tmp` folder.

MATLAB also provides a `tempname` function that returns a filename in the temporary folder. The returned filename is a suitable destination for temporary data. For example, if you need to store some data in a temporary file, then you might issue the following command first:

```
fid = fopen(tempname, 'w');
```

Note The filename that `tempname` generates is not guaranteed to be unique; however, it is likely to be so.

Memory-Mapping Data Files

- “Overview of Memory-Mapping” on page 3-2
- “The memmapfile Class” on page 3-7
- “Constructing a memmapfile Object” on page 3-9
- “Reading a Mapped File” on page 3-23
- “Writing to a Mapped File” on page 3-29
- “Methods of the memmapfile Class” on page 3-37
- “Deleting a Memory Map” on page 3-39
- “Memory-Mapping Demo” on page 3-40

Overview of Memory-Mapping

Memory-mapping is a mechanism that maps a portion of a file, or an entire file, on disk to a range of addresses within an application's address space. The application can then access files on disk in the same way it accesses dynamic memory. This makes file reads and writes faster in comparison with using functions such as `fread` and `fwrite`.

Another advantage of using memory-mapping in your MATLAB application is that it enables you to access file data using standard MATLAB indexing operations. Once you have mapped a file to memory, you can read the contents of that file using the same type of MATLAB statements used to read variables from the MATLAB workspace. The contents of the mapped file appear as if they were an array in the currently active workspace. You simply index into this array to read or write the desired data from the file.

This section describes the benefits and limitations of memory-mapping in MATLAB. The last part of this section gives details on which types of applications derive the greatest advantage from using memory-mapping:

- “Benefits of Memory-Mapping” on page 3-2
- “Limitations of MATLAB Memory-Mapping” on page 3-4
- “When to Use Memory-Mapping” on page 3-5

Benefits of Memory-Mapping

The principal benefits of memory-mapping are efficiency, faster file access, the ability to share memory between applications, and more efficient coding.

Faster File Access

Accessing files via memory map is faster than using I/O functions such as `fread` and `fwrite`. Data are read and written using the virtual memory capabilities that are built in to the operating system rather than having to allocate, copy into, and then deallocate data buffers owned by the process.

MATLAB does not access data from the disk when the map is first constructed. It only reads or writes the file on disk when a specified part of the memory

map is accessed, and then it only reads that specific part. This provides faster random access to the mapped data.

Efficiency

Mapping a file into memory allows access to data in the file as if that data had been read into an array in the application's address space. Initially, MATLAB only allocates address space for the array; it does not actually read data from the file until you access the mapped region. As a result, memory-mapped files provide a mechanism by which applications can access data segments in an extremely large file without having to read the entire file into memory first.

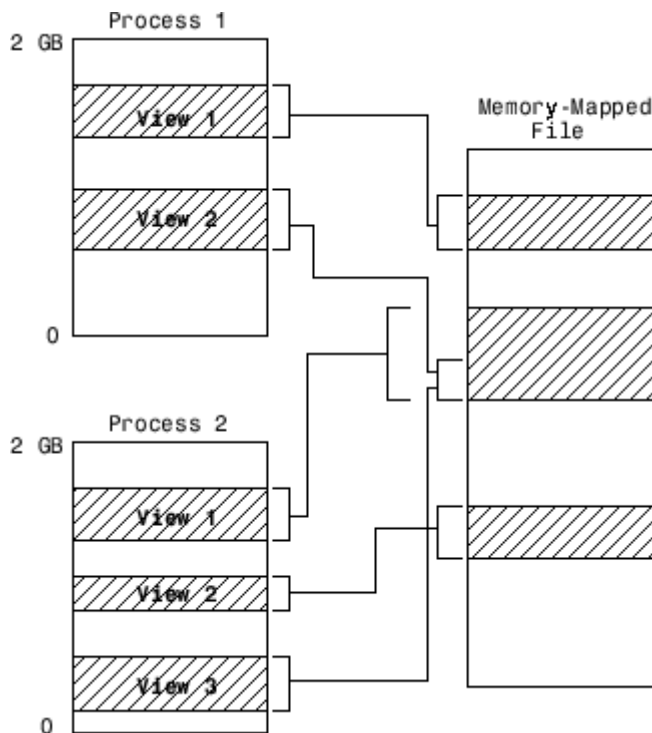
Efficient Coding Style

Memory-mapping eliminates the need for explicit calls to the `fread` and `fwrite` functions. In MATLAB, if `x` is a memory-mapped variable, and `y` is the data to be written to a file, then writing to the file is as simple as

```
x.Data = y;
```

Sharing Memory Between Applications

Memory-mapped files also provide a mechanism for sharing data between applications, as shown in the figure below. This is achieved by having each application map sections of the same file. You can use this feature to transfer large data sets between MATLAB and other applications.



Also, within a single application, you can map the same segment of a file more than once.

Limitations of MATLAB Memory-Mapping

MATLAB restricts the size of a memory map to 2 gigabytes, and on some platforms, requires that you set up your memory-mapping so that all data access is aligned properly. See the following section, “Maximum Size of a Memory Map”, for more information.

Maximum Size of a Memory Map

Due to limits set by the operating system and MATLAB, the maximum amount of data you can map with a single instance of a memory map is 2 gigabytes on 32-bit systems, and 256 terabytes on 64-bit systems. If you need to map more than this limit, you can either create separate maps for different regions of the file, or you can move the window of one map to different locations in the file.

Aligned Access on Sol64

The Sol64 platform only supports aligned data access. This means that numeric values of type `double` that are to be read from a memory-mapped file must start at some multiple of 8 bytes from the start of the file. (Note that this is from the start of the *file*, and not the start of the mapped region.) Furthermore, numeric values of type `single` and also 32-bit integers must start at multiples of 4 bytes, and 16-bit integers at 2-byte multiples.

If you attempt to map a file on Sol64, which does not take into account these alignment considerations, MATLAB generates an error.

Byte Ordering

Memory-mapping works only with data that have the same byte ordering scheme as the native byte ordering of your operating system. For example, because both Linus Torvald's Linux[®] and Microsoft Windows systems use little-endian byte ordering, data created on a Linux system can be read on Windows systems. You can use the `computer` function to determine the native byte ordering of your current system.

When to Use Memory-Mapping

Just how much advantage you get from mapping a file to memory depends mostly on the size and format of the file, the way in which data in the file is used, and the computer platform you are using.

When Memory-Mapping Is Most Useful

Memory-mapping works best with binary files, and in the following scenarios:

- For large files that you want to access randomly one or more times

- For small files that you want to read into memory once and access frequently
- For data that you want to share between applications
- When you want to work with data in a file as if it were a MATLAB array

When the Advantage Is Less Significant

The following types of files do not fully use the benefits of memory-mapping:

- Formatted binary files like HDF or TIFF that require customized readers are not good for memory-mapping. Describing the data contained in these files can be a very complex task. Also, you cannot access data directly from the mapped segment, but must instead create arrays to hold the data.
- Text or ASCII files require that you convert the text in the mapped region to an appropriate type for the data to be meaningful. This takes up additional address space.
- Files that are larger than several hundred megabytes in size consume a significant amount of the virtual address space needed by MATLAB to process your program. Mapping files of this size may result in MATLAB reporting out-of-memory errors more often. This is more likely if MATLAB has been running for some time, or if the memory used by MATLAB becomes fragmented.

The memmapfile Class

MATLAB implements memory-mapping using an object-oriented class called `memmapfile`. The `memmapfile` class has the properties and methods you need to map to a file, read and write the file via the map, and remove the map from memory when you are done.

Properties of the memmapfile Class

There are six properties defined for the `memmapfile` class. These are shown in the table below. These properties control which file is being mapped, where in the file the mapping is to begin and end, how the contents of the file are to be formatted, and whether or not the file is writable. One property of the file contains the file data itself.

Property	Description	Data Type	Default
Data	Contains the data read from the file or to be written to the file. (See “Reading a Mapped File” on page 3-23 and “Writing to a Mapped File” on page 3-29)	Any of the numeric types	None
Filename	Path and name of the file to map into memory. (See “Selecting the File to Map” on page 3-12)	char array	None
Format	Format of the contents of the mapped region, including class, array shape, and variable or field name by which to access the data. (See “Identifying the Contents of the Mapped Region” on page 3-14)	char array or N-by-3 cell array	uint8
Offset	Number of bytes from the start of the file to the start of the mapped region. This number is zero-based. That is, offset 0 represents the start of the file. Must be a nonnegative integer value. (See “Setting the Start of the Mapped Region” on page 3-13)	double	0
Repeat	Number of times to apply the specified format to the mapped region of the file. Must be a positive integer value or Inf. (See “Repeating a Format Scheme” on page 3-20)	double	Inf
Writable	Type of access allowed to the mapped region. Must be logical 1 or logical 0. (See “Setting the Type of Access” on page 3-21)	logical	false

You can set the values for any property except for `Data` at the time you call the `memmapfile` constructor, or at any time after that while the map is still valid. Any properties that are not explicitly set when you construct the object are given their default values as shown in the table above. For information on calling the constructor, see “Constructing a `memmapfile` Object” on page 3-9.

Once a `memmapfile` object has been constructed, you can change the value of any of its properties. Use the `objname.property` syntax in assigning the new value. For example, to set a new `Offset` value for memory map object `m`, type

```
m.Offset = 2048;
```

Note Property names are not case sensitive. For example, MATLAB considers `m.offset` to be the same as `m.Offset`.

To display the value of all properties of a `memmapfile` object, simply type the object name. For a `memmapfile` object `m`, typing the variable name `m` displays the following. Note that this example requires the file `records.dat` which you will create at the beginning of the next section.

```
m =  
  Filename: 'records.dat'  
  Writable: true  
  Offset: 1024  
  Format: 'uint32'  
  Repeat: Inf  
  Data: 4778x1 uint32 array
```

To display the value of any individual property, for example the `Writable` property of object `m`, type

```
m.Writable  
ans =  
  true
```

Constructing a memmapfile Object

The first step in mapping to any file is to construct an instance of the `memmapfile` class using the class constructor function. You can have MATLAB assign default values to each of the new object's properties, or you can specify property values yourself in the call to the `memmapfile` constructor.

In this section...

“Constructing the Object with Default Property Values” on page 3-10

“Changing Property Values” on page 3-10

“Selecting the File to Map” on page 3-12

“Setting the Start of the Mapped Region” on page 3-13

“Identifying the Contents of the Mapped Region” on page 3-14

“Mapping of the Example File” on page 3-18

“Repeating a Format Scheme” on page 3-20

“Setting the Type of Access” on page 3-21

Most of the examples in this section use a file named `records.dat` that contains a 5000-by-1 matrix of double-precision floating point numbers. Use the following code to generate this file before going on to the next sections of this documentation.

First, save this function in your current working directory:

```
function gendatafile(filename, count)
    dmax32 = double(intmax('uint32'));
    randData = gallery('uniformdata', [count, 1], 0) * dmax32;

    fid = fopen(filename, 'w');
    fwrite(fid, randData, 'double');
    fclose(fid);
```

Now execute the `gendatafile` function to generate the `records.dat` file that is referenced in this section. You can use this function at any time to regenerate the file:

```
gendatafile('records.dat', 5000);
```

Constructing the Object with Default Property Values

The simplest and most general way to call the constructor is with one input argument that specifies the name of the file you want to map. All other properties are optional and are given their default values. Use the syntax shown here:

```
objname = memmapfile(filename)
```

To construct a map for the file `records.dat` that resides in your current working directory, type the following:

```
m = memmapfile('records.dat')
m =
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: false
    Offset: 0
    Format: 'uint8'
    Repeat: Inf
    Data: 40000x1 uint8 array
```

MATLAB constructs an instance of the `memmapfile` class, assigns it to the variable `m`, and maps the entire `records.dat` file to memory, setting all object properties to their default values. In this example, the command maps the entire file as a sequence of unsigned 8-bit integers and gives the caller read-only access to its contents.

Changing Property Values

You can make the memory map more specific to your needs by including more information when calling the constructor. In addition to the `filename` argument, there are four other parameters that you can pass to the constructor. Each of these parameters represents a property of the object, and each requires an accompanying value to be passed, as well:

```
objname = memmapfile(filename, prop1, value1, prop2, value2, ...)
```

For example, to construct a map using nondefault values for the `Offset`, `Format`, and `Writable` properties, type the following, enclosing all property names and string parameter values in quotes:

```
m = memmapfile('records.dat', ...
              'Offset', 1024, ...
              'Format', 'double', ...
              'Writable', true);
```

Type the object name to see the current settings for all properties:

```
m

m =
  Filename: 'd:\matlab\mfiles\records.dat'
  Writable: true
  Offset: 1024
  Format: 'double'
  Repeat: Inf
  Data: 4872x1 double array
```

You can also change the value of any property after the object has been constructed. Use the syntax:

```
objname.property = newvalue;
```

For example, to set the format to `uint16`, type the following. (Property names, like `Format`, are not case sensitive.)

```
m.format = 'uint16'
m =
  Filename: 'd:\matlab\mfiles\records.dat'
  Writable: true
  Offset: 1024
  Format: 'uint16'
  Repeat: Inf
  Data: 19488x1 uint16 array
```

Further read and write operations to the region mapped by `m` now treat the data in the file as a sequence of unsigned 16-bit integers. Whenever you change the value of a `memmapfile` property, MATLAB remaps the file to memory.

Selecting the File to Map

`filename` is the only required argument when you call the `memmapfile` constructor. When you call the `memmapfile` constructor, MATLAB assigns the file name that you specify to the `Filename` property of the new object instance.

Specify the file name as a quoted string, (e.g., `'records.dat'`). It must be first in the argument list and not specified as a parameter-value pair. `filename` must include a file name extension if the name of the file being mapped has an extension. The `filename` argument cannot include any wildcard characters (e.g., `*` or `?`), and is not case sensitive.

Note Unlike the other `memmapfile` constructor arguments, you must specify `filename` as a single string, and not as a parameter-value pair.

If the file to be mapped resides somewhere on the MATLAB path, you can use a partial pathname. If the path to the file is not fully specified, MATLAB searches for the file in your current working directory first, and then on the MATLAB path.

Once `memmapfile` locates the file, MATLAB stores the absolute path name for the file internally, and then uses this stored path to locate the file from that point on. This enables you to work in other directories outside your current work directory and retain access to the mapped file.

You can change the value of the `Filename` property at any time after constructing the `memmapfile` object. You might want to do this if:

- You want to use the same `memmapfile` object on more than one file.
- You save your `memmapfile` object to a MAT-file, and then later load it back into MATLAB in an environment where the mapped file has been moved to a different location. This requires that you modify the path segment of the `Filename` string to represent the new location.

For example, save `memmapfile` object `m` to file `mymap.mat`:

```
disp(m.Filename)
d:\matlab\mfiles\records.dat
```



```
save mymat m
```

Now move the file to another location, load the object back into MATLAB, and update the path in the `Filename` property:

```
load mymat m
m.Filename = 'f:\testfiles\oct1\records.dat'
```

Note You can only map an existing file. You cannot create a new file and map that file to memory in one operation. Use the MATLAB file I/O functions to create the file before attempting to map it to memory.

Setting the Start of the Mapped Region

By default, MATLAB begins a memory map at the start of the file. To begin the mapped region at some point beyond the start of the file, specify an `Offset` parameter in the call to the `memmapfile` constructor:

```
objname = memmapfile(filename, 'Offset', bytecount)
```

The `bytecount` value is the number of bytes from the beginning of the file to the point in the file where you want the memory map to start (a zero-based offset). To map the file `records.dat` from a point 1024 bytes from the start and extending to the end of the file, type

```
m = memmapfile('records.dat', 'Offset', 1024);
```

You can change the starting position of an existing memory map by setting the `Offset` property for the associated object to a new value. The following command sets the offset of `memmapfile` object `m` to be 2,048 bytes from the start of the mapped file:

```
m.Offset = 2048;
```

Note The Sol64 platform supports aligned data access only. If you attempt to use a `memmapfile` offset on Sol64 that does not take the necessary alignment considerations into account, MATLAB generates an error. (See “Aligned Access on Sol64” on page 3-5).

Identifying the Contents of the Mapped Region

By default, MATLAB considers all the data in a mapped file to be a sequence of unsigned 8-bit integers. To have the data interpreted otherwise as it is read or written to in the mapped file, specify a `Format` parameter and value in your call to the constructor:

```
objname = memmapfile(filename, 'Format', formatspec)
```

The `formatspec` argument can either be a character string that identifies a single class used throughout the mapped region, or a cell array that specifies more than one class.

For example, say that you map a file that is 12 kilobytes in length. Data read from this file could be treated as a sequence of 6,000 16-bit (2-byte) integers, or as 1,500 8-byte double-precision floating-point numbers, to name just a couple of possibilities. Or you could read this data in as a combination of different types: for example, as 4,000 8-bit (1-byte) integers followed by 1,000 64-bit (8-byte) integers. You determine how MATLAB will interpret the mapped data by setting the `Format` property of the `memmapfile` object when you call its constructor.

MATLAB arrays are stored on disk in column-major order. (The sequence of array elements is column 1, row 1; column 1, row 2; column 1, last row; column 2, row 1, and so on.) You might need to transpose or rearrange the order of array elements when reading or writing via a memory map.

Note The Sol64 platform supports aligned data access only. If you attempt to use a `memmapfile` format on Sol64 that does not take the necessary alignment considerations into account, MATLAB generates an error. (See “Aligned Access on Sol64” on page 3-5).

For a list of data types supported for the `Format` property, see “Supported Data Types for the Format Property” on page 3-19.

For more information on format options see these sections:

- “Mapping a Single Data Type” on page 3-15
- “Formatting the Mapped Data to an Array” on page 3-16
- “Mapping Multiple Data Types and Arrays” on page 3-17

Mapping a Single Data Type

If the file region being mapped contains data of only one type, specify the `Format` value as a character string identifying that type:

```
objname = memmapfile(filename, 'Format', datatype)
```

The following command constructs a `memmapfile` object for the entire file `records.dat`, and sets the `Format` property for that object to `uint64`. Any read or write operations made via the memory map will read and write the file contents as a sequence of unsigned 64-bit integers:

```
m = memmapfile('records.dat', 'Format', 'uint64')
  Filename: 'd:\matlab\mfiles\records.dat'
  Writable: false
  Offset: 0
  Format: 'uint64'
  Repeat: Inf
  Data: 5000x1 uint64 array
```

You can change the value of the `Format` property at any time after the `memmapfile` object is constructed. Use the `object.property` syntax shown here in assigning the new value:

```
m.Format = 'int32';
```

Further read and write operations to the region mapped by `m` now treat the data in the file as a sequence of signed 32-bit integers.

Property names, like `Format`, are not case sensitive.

Formatting the Mapped Data to an Array

You can also specify an array shape to be applied to the data read or written to the mapped file, and a field name to be used in referencing this array. Use a cell array to hold these values either when calling the `memmapfile` constructor or when modifying `m.Format` after the object has been constructed. The cell array contains three elements: the class to be applied to the mapped region, the dimensions of the array shape that is applied to the region, and a field name to use in referencing the data:

```
objname = memmapfile(filename, ...  
                    'Format', {datatype, dimensions, varname})
```

The following command constructs a `memmapfile` object for a region of `records.dat` such that MATLAB handles the contents of the region as a 4-by-10-by-18 array of unsigned 32-bit integers, which you can reference in the structure of the returned object using the field name `x`:

```
m = memmapfile('records.dat', ...  
              'Offset', 1024, ...  
              'Format', {'uint32' [4 10 18] 'x'})  
m =  
  Filename: 'd:\matlab\mfiles\records.dat'  
  Writable: false  
  Offset: 1024  
  Format: {'uint32' [4 10 18] 'x'}  
  Repeat: Inf  
  Data: 13x1 struct array with fields:  
        x
```

```
A = m.Data(1).x;
```

```
whos A  
  Name      Size      Bytes  Class  Attributes  
  A         4x10x18    2880  uint32
```

You can change the class, array shape, or field name that MATLAB applies to the mapped region at any time by setting a new value for the `Format` property of the object:

```
m.Format = {'uint64' [30 4 10] 'x'};
```

```
A = m.Data(1).x;
```

```
whos A
  Name      Size      Bytes  Class      Attributes
  A         30x4x10    9600   uint64
```

Mapping Multiple Data Types and Arrays

If the region being mapped is composed of segments of varying classes or array shapes, you can specify an individual format for each segment using an N-by-3 cell array, where N is the number of segments. The cells of each cell array row identify the class for that segment, the array dimensions to map the data to, and a field name by which to reference that segment:

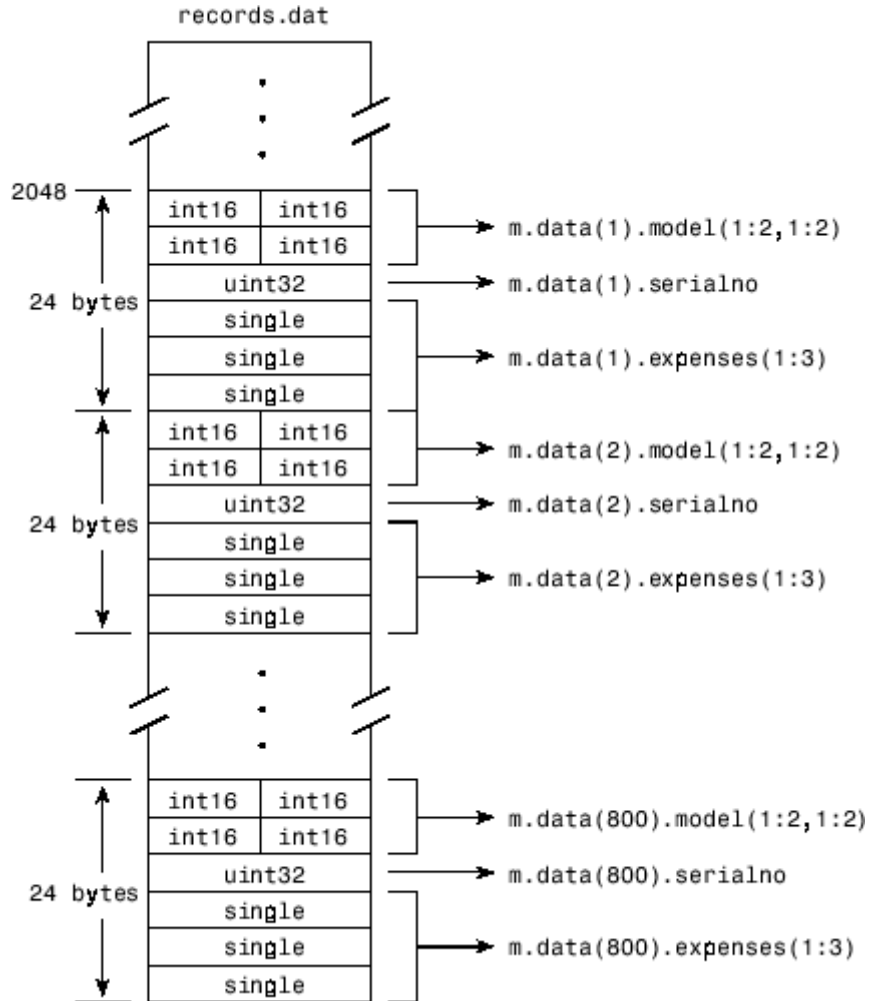
```
objname = memmapfile(filename, ...
    'Format', { ...
        datatype1, dimensions1, fieldname1; ...
        datatype2, dimensions2, fieldname2; ...
        :           :           : ...
        datatypeN, dimensionsN, fieldnameN});
```

The following command maps data in a 20.75-kilobyte file to three different classes: `int16`, `uint32`, and `single`. The `int16` data is mapped as a 2-by-2 matrix that can be accessed using the field name `model`. The `uint32` data is a scalar value accessed as field `serialno`. The `single` data is a 1-by-3 matrix named `expenses`.

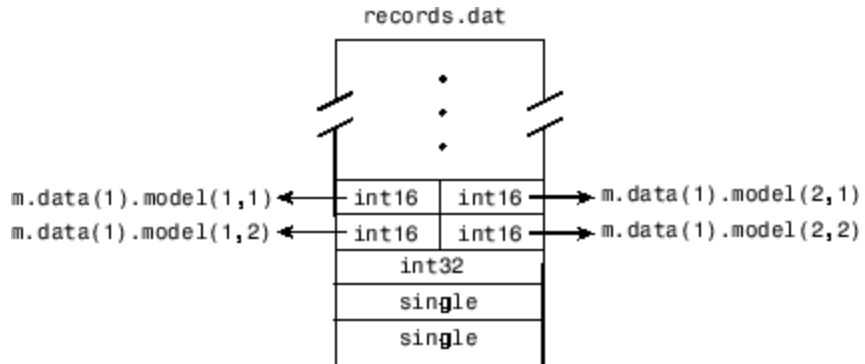
Each of these fields belongs to the 800-by-1 structure array `m.Data`:

```
m = memmapfile('records.dat', ...
    'Offset', 2048, ...
    'Format', { ...
        'int16' [2 2] 'model'; ...
        'uint32' [1 1] 'serialno'; ...
        'single' [1 3] 'expenses'});
```

Mapping of the Example File



The figure below shows the ordering of the array elements more closely. In particular, it illustrates that MATLAB arrays are stored on the disk in column-major order. The sequence of array elements in the mapped file is row 1, column 1; row 2, column 1; row 1, column 2; and row 2, column 2.



If the data in your file is not stored in this order, you might need to transpose or rearrange the order of array elements when reading or writing via a memory map.

Supported Data Types for the Format Property

You can use any of the following classes when you specify a Format value. The default type is `uint8`.

Format String	Data Type Description
'int8'	Signed 8-bit integers
'int16'	Signed 16-bit integers
'int32'	Signed 32-bit integers
'int64'	Signed 64-bit integers
'uint8'	Unsigned 8-bit integers
'uint16'	Unsigned 16-bit integers
'uint32'	Unsigned 32-bit integers
'uint64'	Unsigned 64-bit integers
'single'	32-bit floating-point
'double'	64-bit floating-point

Repeating a Format Scheme

After you set a `Format` value for the `memmapfile` object, you can have MATLAB apply that format to the file data multiple times by specifying a `Repeat` value when you call the `memmapfile` constructor:

```
objname = memmapfile(filename, ...
                    'Format', formatspec, ...
                    'Repeat', count)
```

The `Repeat` value applies to the whole format specifier, whether that specifier describes just a single class that repeats, or a more complex format that includes various classes and array shapes. The default `Repeat` value is infinity (`inf`), which means that the full extent of the `Format` specifier repeats as many times as possible within the mapped region.

The next example maps a file region identical to that of the previous example, except the pattern of `int16`, `uint32`, and `single` classes is repeated only three times within the mapped region of the file:

```
m = memmapfile('records.dat', ...
              'Offset', 2048, ...
              'Format', {
                  'int16' [2 2] 'model'; ...
                  'uint32' [1 1] 'serialno'; ...
                  'single' [1 3] 'expenses'}, ...
              'Repeat', 3);
```

You can change the value of the `Repeat` property at any time. To change the repeat value to 5, type

```
m.Repeat = 5;
```

Property names, like `Repeat`, are not case sensitive.

Keeping the Repeated Format Within the Mapped Region

MATLAB maps only the *full* pattern specified by the `Format` property. If you repeat a format such that it would cause the map to extend beyond the end of the file, then either of two things can happen:

- If you specify a repeat value of `Inf`, MATLAB applies to the map only those repeated segments that fit within the file in their entirety.
- If you specify a repeat value other than `Inf`, and that value would cause the map to extend beyond the end of the file, MATLAB generates an error.

Considering the last example, if the part of the file from `m.Offset` to the end were 70 bytes (instead of the 72 bytes required to repeat `m.Format` three times) and you used a `Repeat` value of `Inf`, then only two full repetitions of the specified format would have been mapped. The end result is as if you had constructed the map with this command:

```
m = memmapfile('records.dat',      ...
              'Offset', 2048,      ...
              'Format', {          ...
                  'int16' [2 2] 'model'; ...
                  'uint32' [1 1] 'serialno'; ...
                  'single' [1 3] 'expenses'}, ...
              'Repeat', 2);
```

If `Repeat` were set to 3 and you had only 70 bytes to the end of the file, you would get an error.

Note `memmapfile` does not expand or append to a mapped file. Use standard file I/O functions like `fopen` and `fwrite` to do this.

Setting the Type of Access

You can map a file region to allow either read-only or read and write access to its contents. Pass a `Writable` parameter and value in the `memmapfile` constructor, or set `m.Writable` on an existing object to set the type of access allowed:

```
objname = memmapfile(filename, 'Writable', trueorfalse)
```

The value passed can be either `true` (equal to `logical(1)`) or `false` (equal to `logical(0)`). By default, it is `false`, meaning that the mapped region is read only.

To map a read and write region of the file `records.dat` in memory, type

```
m = memmapfile('records.dat', 'Writable', true);
```

Note To successfully modify the file you are mapping to, you must have write permission for that file. If you do not have write permission, you can still set the `Writable` property to `true`, but attempting to write to the file generates an error.

You can change the value of the `Writable` property at any time. To make the memory map to `records.dat` read only, type:

```
m.Writable = false;
```

Property names, like `Writable`, are not case sensitive.

Reading a Mapped File

The most commonly used property of the `memmapfile` class is the `Data` property. It is through this property of the memory-map object that MATLAB provides all read and write access to the contents of the mapped file.

The actual mapping of a file to the MATLAB address space does not take place when you construct a `memmapfile` object. A memory map, based on the information currently stored in the mapped object, is generated the first time you reference or modify the `Data` property for that object.

After you map a file to memory, you can read the contents of that file using the same MATLAB statements used to read variables from the MATLAB workspace. By accessing the `Data` property of the memory map object, the contents of the mapped file appear as if they were an array in the currently active workspace. You simply index into this array to read the desired data from the file.

In this section...

“Improving Performance” on page 3-23

“Example 1 — Reading a Single Data Type” on page 3-24

“Example 2 — Formatting File Data as a Matrix” on page 3-25

“Example 3 — Reading Multiple Data Types” on page 3-26

“Example 4 — Modifying Map Parameters” on page 3-27

Improving Performance

MATLAB accesses data in structures more efficiently than it does data contained in objects. The main reason is that structures do not require the extra overhead of a `subsref` routine. Instead of reading directly from the `memmapfile` object, as shown here:

```
for k = 1 : N
    y(k) = m.Data(k);
end
```

you will get better performance when you assign the `Data` field to a variable, and then read the mapped file through this variable, as shown in this second example:

```
dataRef = m.Data;
for k = 1 : N
    y(k) = dataRef(k);
end
```

Example 1 — Reading a Single Data Type

This example maps a file of 100 double-precision floating-point numbers to memory. The map begins 1024 bytes from the start of the file, and ends 800 bytes (8 bytes per double times a `Repeat` value of 100) from that point.

If you haven't done so already, generate a test data file for use in the following examples by executing the `gendatafile` function defined under “Constructing a `memmapfile` Object” on page 3-9:

```
gendatafile('records.dat', 5000);
```

Now, construct the `memmapfile` object `m`, and show the format of its `Data` property:

```
m = memmapfile('records.dat', 'Format', 'double', ...
    'Offset', 1024, 'Repeat', 100);
```

```
d = m.Data;
```

```
whos d
      Name      Size      Bytes  Class  Attributes
      d          100x1      800    double
```

Read a selected set of numbers from the file by indexing into the single-precision array `m.Data`:

```
d(15:20)
ans =
    1.0e+009 *
    3.6045
    2.7006
```

```

0.5745
0.8896
2.6079
2.7053

```

Example 2 – Formatting File Data as a Matrix

This example is similar to the last, except that the constructor of the `memmapfile` object now specifies an array shape of 4-by-6 to be applied to the data as it is read from the mapped file. MATLAB maps the file contents into a structure array rather than a numeric array, as in the previous example:

```

m = memmapfile('records.dat', ...
    'Format', {'double', [4 6], 'x'}, ...
    'Offset', 1024, 'Repeat', 100);

d = m.Data;

```

```

whos d

```

Name	Size	Bytes	Class	Attributes
d	100x1	25264	struct	

When you read an element of the structure array, MATLAB presents the data in the form of a 4-by-6 array:

```

d(5).x
ans =
1.0e+009 *
    3.1564    0.6684    2.1056    1.9357    1.2773    4.2219
    2.9520    0.8208    3.5044    1.7705    0.2112    2.3737
    1.4865    1.8144    1.9790    3.8724    2.9772    1.7183
    0.7131    3.6764    1.9643    0.0240    2.7922    0.8538

```

To index into the structure array field, use:

```

d(5).x(3,2:6)
ans =
1.0e+009 *
    1.8144    1.9790    3.8724    2.9772    1.7183

```

Example 3 – Reading Multiple Data Types

This example maps a file containing more than one class. The different classes contained in the file are mapped as fields of the returned structure array `m.Data`.

The `Format` parameter passed in the constructor specifies that the first 80 bytes of the file are to be treated as a 5-by-8 matrix of `uint16`, and the 160 bytes after that as a 4-by-5 matrix of `double`. This pattern repeats until the end of the file is reached. The example shows different ways of reading the `Data` property of the object.

Start by calling the `memmapfile` constructor to create a memory map object, `m`:

```
m = memmapfile('records.dat', ...
    'Format', { ...
        'uint16' [5 8] 'x'; ...
        'double' [4 5] 'y' });
```

If you examine the `Data` property, MATLAB shows a 166-element structure array with two fields, one for each format specifier in the constructor:

```
d = m.Data
ans =
166x1 struct array with fields:
    x
    y
```

Examine one structure in the array to show the format of each field:

```
d(3)
ans =
    x: [5x8 uint16]
    y: [4x5 double]
```

Now read the `x` and `y` fields of that structure from the file. MATLAB formats the first block of data as a 5-by-8 matrix of `uint16`, as specified in the `Format` property, and the second block as a 4-by-5 matrix of `double`:

```
d(3).x
ans =
    34432    47500    19145    16868    38165    47956    35550    16853
```

```

60654  51944  16874  47166  35397  58072  16850  56576
51075  16876  12471  34369   8341  16853  44509  57652
16863  16453   6666  11480  16869  58695  36217  5932
57883  15551  41755  16874  37774  31693  54813  16865

```

```

d(3).y
ans =
1.0e+009 *
  3.1229    1.5909    2.9831    2.2445    1.1659
  1.3284    3.0182    2.6685    3.7802    1.0837
  3.6013    2.3475    3.4137    0.7428    3.7613
  2.4399    1.9107    4.1096    4.2080    3.1667

```

Example 4 – Modifying Map Parameters

This example plots the Fourier transform output of data read from a file via a memory map. It then modifies several parameters of the existing map, reads from a different part of the data file, and plots a histogram from that data.

Create a memory-mapped object, mapping 1,000 elements of type `double` starting at the 1025th byte:

```

m = memmapfile('mybinary.bin', 'Offset', 1024, ...
              'Format', 'double', 'Repeat', 1000);

```

Get data associated with the map and plot the FFT of the first 1000 values of the map. This is when the map is actually created, because no data has been referenced until this point:

```

plot(abs(fft(m.Data(1:1000))));

```

Get information about the memory map:

```

mapStruct = get(m)

mapStruct =
  Filename: 'd:\matlab\mfiles\mybinary.bin'
  Writable: 0
  Offset: 1024
  Format: 'double'
  Repeat: 1000

```

```
Data: [1000x1 double]
```

Change the map, but continue using the same file:

```
m.Offset = 4096;  
m.Format = 'single';  
m.Repeat = 800;
```

Read from a different area of the file, and plot a histogram of the data. This maps a new region and unmaps the previous region:

```
hist(m.Data)
```


Writing to a Mapped File

Writing to a mapped file is done with standard MATLAB subscripted assignment commands. To write to a particular location in the file mapped to `memmapfile` object `m`, assign the value to the `m.Data` structure array index and field that map to that location.

If you haven't done so already, generate a test data file for use in the following examples by executing the `gendatafile` function defined under “Constructing a `memmapfile` Object” on page 3-9:

```
gendatafile('records.dat', 5000);
```

Now call the `memmapfile` constructor to create the object:

```
m = memmapfile('records.dat', ...
    'Format', {
        'uint16' [5 8] 'x'; ...
        'double' [4 5] 'y' });
```

If you are going to modify the mapped file, be sure that you have write permission, and that you set the `Writable` property of the `memmapfile` object to `true` (logical 1):

```
m.Writable = true;
```

Note You do not have to set `Writable` as a separate command, as done here. You can include a `Writable` parameter-value argument in the call to the `memmapfile` constructor.

View the 5-by-8 matrix `x` at `m.Data(2)`:

```
m.Data(2).x

ans =
    35330    4902    31861    16877    23791    61500    52748    16841
    51314    58795    16860    43523     8957     5182    16864    60110
    18415    16871    59373    61001    52007    16875    26374    28570
    16783     4356    52847    53977    16858    38427    16067    33318
```

```
65372 48883 53612 16861 18882 39824 61529 16869
```

Update all values in that matrix using a standard MATLAB assignment statement:

```
m.Data(2).x = m.Data(2).x * 1.5;
```

Verify the results:

```
m.Data(2).x

ans =
  52995   7353  47792  25316  35687  65535  65535  25262
  65535  65535  25290  65285  13436   7773  25296  65535
  27623  25307  65535  65535  65535  25313  39561  42855
  25175   6534  65535  65535  25287  57641  24101  49977
  65535  65535  65535  25292  28323  59736  65535  25304
```

In this section...

“Dimensions of the Data Field” on page 3-30

“Writing Matrices to a Mapped File” on page 3-31

“Selecting Appropriate Data Types” on page 3-34

“Working with Copies of the Mapped Data” on page 3-35

Dimensions of the Data Field

Although you can expand the dimensions of a typical MATLAB array by assigning outside its current dimensions, this does not apply to the `Data` property of a `memmapfile` object. The dimensions of a `memmapfile` object's `Data` field are set at the time you construct the object and cannot be changed.

For example, you can add a new column to the field of a MATLAB structure:

```
A.s = ones(4,5);

A.s(:,6) = [1 2 3 4];           % Add new column to A.s
size(A.s)
ans =
     4     6
```

However, you cannot add a new column to a similar field of a structure that represents data mapped from a file. The following assignment to `m.Data(60).y` does not expand the size of `y`, but instead generates an error:

```
m.Data(60)
ans =
    x: [5x8 uint16]
    y: [4x5 double]

m.Data(60).y(:,6) = [1 2 3 4];           % Generates an error.
```

Thus, if you map an entire file and then append to that file after constructing the map, the appended data is not included in the mapped region. If you need to modify the dimensions of data that you have mapped to a `memmapfile` object, you must either modify the `Format` or `Repeat` properties for the object, or reconstruct the object.

Examples of Invalid Syntax

Several examples of statements that attempt to modify the dimensions of a mapped `Data` field are shown here. These statements result in an error.

The first example attempts to diminish the size of the array by removing a row from the mapped array `m.Data`.

```
m.Data(5) = [];
```

The second example attempts to expand the size of a 50-row mapped array `x` by adding another row to it:

```
m.Data(2).x(1:51,31) = 1:51;
```

Similarly, if `m.Data` has only 100 elements, the following operation is invalid:

```
m.Data(120) = x;
```

Writing Matrices to a Mapped File

The syntax to use when writing to mapped memory can depend on what format was used when you mapped memory to the file.

When Memory Is Mapped in Nonstructure Format

When you map a file as a sequence of a single class (e.g., a sequence of `uint16`), you can use the following syntax to write matrix `X` to the file:

```
m.Data = X;
```

This statement is valid only if all of the following conditions are true:

- The file is mapped as a sequence of elements of the same class, making `m.Data` an array of a nonstructure type.
- The class of `X` is the same as the class of `m.Data`.
- The number of elements in `X` equals the number of elements in `m.Data`.

This example maps a file as a sequence of 16-bit unsigned integers, and then uses the syntax shown above to write a matrix to the file. Map only a small part of the file, using a `uint16` format for this segment:

```
m = memmapfile('records.dat', 'Writable', true, ...  
              'Offset', 2000, 'Format', 'uint16', 'Repeat', 15);
```

Create a matrix `X` of the same size and write it to the mapped part of the file:

```
X = uint16(5:5:75);    % Sequence of 5 to 75, counting by fives.  
m.data = X;
```

Verify that new values were written to the file:

```
m.offset = 1980;    m.repeat = 35;  
reshape(m.data,5,7)'  
ans =  
    29158    16841    32915    37696     421           % <== At offset 1980  
    16868    51434    17455    30645    16871  
         5         10         15         20         25           % <== At offset 2000  
        30        35        40        45        50  
        55        60        65        70        75  
    16872    50155    51100    26469    16873  
    56776     6257    28746    16877    34374
```

When Memory Is Mapped in Scalar Structure Format

When you map a file as a sequence of a single class (e.g., a sequence of `uint16`), you can use the following syntax to write matrix `X` to the file:

```
m.Data.f = X;
```

This statement is valid only if all of the following conditions are true:

- The file is mapped as containing multiple classes that do not repeat, making `m.Data` a scalar structure.
- The class of `X` is the same as the class of `m.Data.f`.
- The number of elements in `X` equals that of `m.Data.f`.

This example maps a file as a 300-by-8 matrix of type `uint16` followed by a 200-by-5 matrix of type `double`, and then uses the syntax shown above to write a matrix to the file.

```
m = memmapfile('records.dat', ...
    'Format', {
        'uint16' [300 8] 'x'; ...
        'double' [200 5] 'y' }, ...
    'Repeat', 1, 'Writable', true);

m.Data.x = ones(300, 8, 'uint16');
```

When Memory Is Mapped in Nonscalar Structure Format

When you map a file as a repeating sequence of multiple classes, you can use the following syntax to write matrix `X` to the file, providing that `k` is a scalar index:

```
m.Data(k).field = X;
```

To do this, the following conditions must be true:

- The file is mapped as containing multiple classes that can repeat, making `m.Data` a nonscalar structure.
- `k` is a scalar index.
- The class of `X` is the same as the class of `m.Data(k).field`.

- The number of elements in `X` equals that of `m.Data(k).field`.

This example maps a file as a matrix of type `uint16` followed by a matrix of type `double` that repeat 20 times, and then uses the syntax shown above to write a matrix to the file.

```
m = memmapfile('records.dat', ...
    'Format', {
        'uint16' [25 8] 'x'; ...
        'double' [15 5] 'y' }, ...
    'Repeat', 20, 'Writable', true);

m.Data(12).x = ones(25,8,'uint16');
```

You can write to specific elements of field `x` as shown here:

```
m.Data(12).x(3:5,1:end) = uint16(500);
m.Data(12).x(3:5,1:end)

ans =
    500    500    500    500    500    500    500    500
    500    500    500    500    500    500    500    500
    500    500    500    500    500    500    500    500
```

Selecting Appropriate Data Types

All of the usual MATLAB indexing and class rules apply when assigning values to data via a memory map. The class that you assign to must be big enough to hold the value being assigned. For example,

```
m = memmapfile('records.dat', 'Format', 'uint8', ...
    'Writable', true);

d = m.Data;
d(5) = 300;
```

saturates the `d` variable because `d` is defined as an 8-bit integer:

```
d(5)
ans =
    255
```

Working with Copies of the Mapped Data

In the following code, the data in variable `d` is a *copy* of the file data mapped by `m.Data(2)`. Because it is a copy, modifying array data in `d` does not modify the data contained in the file:

First, destroy the `memmapfile` object and restore the test file `records.dat`, since you modified it by running the previous examples:

```
clear m
gendatafile('records.dat',50000);
```

Map the file as a series of `uint16` and `double` matrices and make a copy of `m.Data(2)` in `d`:

```
m = memmapfile('records.dat', ...
    'Format', { ...
        'uint16' [5 8] 'x'; ...
        'double' [4 5] 'y' });
```

```
d = m.Data;
```

Write all zeros to the copy:

```
d(2).x(1:5,1:8) = 0;
```

```
d(2).x
ans =
    0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0
```

Verify that the data in the mapped file is not changed even though the copy of `m.Data(2).x` is written with zeros:

```
m.Data(2).x
ans =
    35330    4902    31861    16877    23791    61500    52748    16841
    51314    58795    16860    43523     8957     5182    16864    60110
    18415    16871    59373    61001    52007    16875    26374    28570
```

16783	4356	52847	53977	16858	38427	16067	33318
65372	48883	53612	16861	18882	39824	61529	16869

Methods of the memmapfile Class

You can use the following methods on objects constructed from the `memmapfile` class.

Syntax	Description
<code>disp</code>	Displays properties of the object. The display does not include the object's name.
<code>get(obj)</code>	Returns the values of all properties of the <code>memmapfile</code> object in a structure array.
<code>get(obj, property)</code>	Returns the value of the specified property. <code>property</code> can be a string or cell array of strings, each containing a property name.

Using the disp Method

Use the `disp` method to display all properties of a `memmapfile` object. The text displayed includes only the property value, and not the object name or the MATLAB response string, `ans =`.

Construct object `m`:

```
m = memmapfile('records.dat', ...
    'Offset', 2048, ...
    'Format', {
        'int16' [2 2] 'model'; ...
        'uint32' [1 1] 'serialno'; ...
        'single' [1 3] 'expenses'});
```

and display all of its properties:

```
disp(m)
Filename: 'd:\matlab\mfiles\records.dat'
Writable: false
Offset: 2048
Format: {'int16' [2 2] 'model'
         'uint32' [1 1] 'serialno'
         'single' [1 3] 'expenses'}
```

```
Repeat: Inf
Data: 16581x1 struct array with fields:
    model
    serialno
    expenses
```

Using the get Method

You can use the `get` method of the `memmapfile` class to return information on any or all of the object's properties. Specify one or more property names to get the values of specific properties.

This example returns the values of the `Offset`, `Repeat`, and `Format` properties for a `memmapfile` object. Use the `get` method to return the specified property values in a 1-by-3 cell array, `m_props`:

```
m_props = get(m, {'Offset', 'Repeat', 'Format'})
m_props =
    [2048]    [Inf]    {3x3 cell}

m_props{3}
ans =
    'int16'    [1x2 double]    'model'
    'uint32'    [1x2 double]    'serialno'
    'single'    [1x2 double]    'expenses'
```

You also can choose to use the `objname.property` syntax:

```
m_props = {m.Offset, m.Repeat, m.Format}
m_props =
    [2048]    [Inf]    {3x3 cell}
```

To return the values for all properties with `get`, pass just the object name:

```
get(m)
Filename: 'd:\matlab\mfiles\records.dat'
Writable: 0
Offset: 2048
Format: {3x3 cell}
Repeat: Inf
Data: [16581x1 struct]
```

Deleting a Memory Map

It is not necessary to explicitly call a destructor method to clear a `memmapfile` object from memory when you no longer need it. MATLAB calls the destructor for you whenever you do any of the following:

- Reassign another value to the `memmapfile` object's variable
- Clear the object's variable from memory
- Exit the function scope in which the object was created

The Effect of Shared Data Copies On Performance

When you assign the `Data` field of the `memmapfile` object to a variable, MATLAB makes a shared data copy of the mapped data. This is very efficient as no memory actually gets copied. In the following statement, `memdat` is a shared data copy of the data mapped from the file:

```
memdat = m.Data;
```

When you finish using the mapped data, make sure to clear any variables that shared data with the mapped file before clearing the object itself. If you clear the object first, then the sharing of data between the file and dependent variables is broken, and the data assigned to such variables must be copied into memory before the object is destroyed. If access to the mapped file was over a network, then copying this data to local memory can take considerable time. So, if the statement shown above assigns data to the variable `memdat`, you should be sure to clear `memdat` before clearing `m` when you are finished with the object.

Note Keep in mind that the `memmapfile` object can be cleared in any of the three ways described under “Deleting a Memory Map” on page 3-39.

Memory-Mapping Demo

In this demonstration, two separate MATLAB processes communicate with each other by writing and reading from a shared file. They share the file by mapping part of their memory space to a common location in the file. A write operation to the memory map belonging to the first process can be read from the map belonging to the second, and vice versa.

One MATLAB process (running `send.m`) writes a message to the file via its memory map. It also writes the length of the message to byte 1 in the file, which serves as a means of notifying the other process that a message is available. The second process (running `answer.m`) monitors byte 1 and, upon seeing it set, displays the received message, puts it into uppercase, and echoes the message back to the sender.

The `send` Function

This function prompts you to enter a string and then, using memory-mapping, passes the string to another instance of MATLAB that is running the `answer` function.

Copy the `send` and `answer` functions to files `send.m` and `answer.m` in your current working directory. Begin the demonstration by calling `send` with no inputs. Next, start a second MATLAB session on the same machine, and call the `answer` function in this session. To exit, press **Enter**.

```
function send
% Interactively send a message to ANSWER using memmapfile class.

filename = fullfile(tempdir, 'talk_answer.dat');

% Create the communications file if it is not already there.
if ~exist(filename, 'file')
    [f, msg] = fopen(filename, 'wb');
    if f ~= -1
        fwrite(f, zeros(1,256), 'uint8');
        fclose(f);
    else
        error('MATLAB:demo:send:cannotOpenFile', ...
            'Cannot open file "%s": %s.', filename, msg);
    end
end
```

```

        end
    end

    % Memory map the file.
    m = memmapfile(filename, 'Writable', true, 'Format', 'uint8');

    while true
        % Set first byte to zero, indicating a message is not
        % yet ready.
        m.Data(1) = 0;

        str = input('Enter send string (or RETURN to end): ', 's');

        len = length(str);
        if (len == 0)
            disp('Terminating SEND function.')
            break;
        end

        str = str(1:min(len, 255)); % Message limited to 255 chars.

        % Update the file via the memory map.
        m.Data(2:len+1) = str;
        m.Data(1)=len;

        % Wait until the first byte is set back to zero,
        % indicating that a response is available.
        while (m.Data(1) ~= 0)
            pause(.25);
        end

        % Display the response.
        disp('response from ANSWER is:')
        disp(char(m.Data(2:len+1)))
    end
end

```

The answer Function

The answer function starts a server that, using memory-mapping, watches for a message from send. When the message is received, answer replaces the

message with an uppercase version of it, and sends this new message back to send.

To use `answer`, call it with no inputs:

```
function answer
% Respond to SEND using memmapfile class.

disp('ANSWER server is awaiting message');

filename = fullfile(tempdir, 'talk_answer.dat');

% Create the communications file if it is not already there.
if ~exist(filename, 'file')
    [f, msg] = fopen(filename, 'wb');
    if f ~= -1
        fwrite(f, zeros(1,256), 'uint8');
        fclose(f);
    else
        error('MATLAB:demo:answer:cannotOpenFile', ...
            'Cannot open file "%s": %s.', filename, msg);
    end
end

% Memory map the file.
m = memmapfile(filename, 'Writable', true, 'Format', 'uint8');

while true
    % Wait till first byte is not zero.
    while m.Data(1) == 0
        pause(.25);
    end

    % The first byte now contains the length of the message.
    % Get it from m.
    msg = char(m.Data(2:1+m.Data(1)))';

    % Display the message.
    disp('Received message from SEND:')
    disp(msg)
end
```

```

% Transform the message to all uppercase.
m.Data(2:1+m.Data(1)) = upper(msg);

% Signal to SEND that the response is ready.
m.Data(1) = 0;
end

```

Running the Demo

To see what the demonstration looks like when it is run, first, start two separate MATLAB sessions on the same computer system. Call the `send` function in one and the `answer` function in the other to create a map in each of the processes' memory to the common file:

```

% Run SEND in the first MATLAB session.
send
Enter send string (or RETURN to end):

% Run ANSWER in the second MATLAB session.
answer
ANSWER server is awaiting message

```

Next, enter a message at the prompt displayed by the `send` function. MATLAB writes the message to the shared file. The second MATLAB session, running the `answer` function, loops on byte 1 of the shared file and, when the byte is written by `send`, `answer` reads the message from the file via its memory map. The `answer` function then puts the message into uppercase and writes it back to the file, and `send` (waiting for a reply) reads the message and displays it:

```

% SEND writes a message and reads the uppercase reply.
Hello. Is there anybody out there?
response from ANSWER is:
HELLO. IS THERE ANYBODY OUT THERE?
Enter send string (or RETURN to end):

% ANSWER reads the message from SEND.
Received message from SEND:
Hello. Is there anybody out there?

```

send writes a second message to the file. answer reads it, put it into uppercase, and then writes the message to the file:

```
% SEND writes a second message to the shared file.  
I received your reply.  
response from ANSWER is:  
I RECEIVED YOUR REPLY.  
Enter send string (or RETURN to end): <Enter>  
Terminating SEND function.
```

```
% ANSWER reads the second message.  
Received message from SEND:  
I received your reply.
```


Internet File Access

MATLAB software provides functions for exchanging files over the Internet. You can exchange files using common protocols, such as File Transfer Protocol (FTP), Simple Mail Transport Protocol (SMTP), and HyperText Transfer Protocol (HTTP). In addition, you can create zip archives to minimize the transmitted file size, and also save and work with Web pages.

- “Downloading Web Content and Files” on page 4-2
- “Creating and Decompressing Zip Archives” on page 4-4
- “Sending E-Mail” on page 4-5
- “Performing FTP File Operations” on page 4-8

Downloading Web Content and Files

MATLAB provides two functions for downloading Web pages and files using HTTP: `urlread` and `urlwrite`. With the `urlread` function, you can read and save the contents of a Web page to a string variable in the MATLAB workspace. With the `urlwrite` function, you can save a Web page's content to a file.

Because it creates a string variable in the workspace, the `urlread` function is useful for working with the contents of Web pages in MATLAB. The `urlwrite` function is useful for saving Web pages to a local folder.

Note When using `urlread`, remember that only the HTML in that specific Web page is retrieved. The hyperlink targets, images, and so on are not retrieved.

If you need to pass parameters to a Web page, the `urlread` and `urlwrite` functions let you use HTTP `post` and `get` methods. For more information, see the `urlread` and `urlwrite` reference pages.

Example – Using the `urlread` Function

The following procedure demonstrates how to retrieve the contents of the Web page listing the files submitted to the MATLAB Central File Exchange. It assigns the results to a string variable, `fullList`:

```
filex = 'http://www.mathworks.com/matlabcentral/fileexchange/';  
fullList = urlread(filex);
```

To pass arguments, you can include them manually using the URL, or pass parameters using standard HTTP methods, including `post` and `get`.

For example, to pass arguments as part of the URL, and retrieve only the files uploaded to the Central File Exchange within the past 7 days that contain the word `Simulink`:

```
filex = sprintf('%s%s',...  
    'http://www.mathworks.com/matlabcentral/fileexchange/',...)
```

```
'?duration=7&term=simulink');  
recent = urlread(filex);
```

Alternatively, use the HTTP get method to query the list of files:

```
filex = 'http://www.mathworks.com/matlabcentral/fileexchange/';  
params = {'duration','7','term','simulink'};  
  
recent = urlread(filex,'get',params);
```

For more information, see the `urlread` reference page.

Example – Using the `urlwrite` Function

The following example builds on the procedure in the previous section, but saves the content to a file:

```
% Locate the list of files at the MATLAB Central File Exchange  
% uploaded within the past 7 days, that contain "Simulink."  
filex = 'http://www.mathworks.com/matlabcentral/fileexchange/';  
params = {'duration','7','term','simulink'};  
  
% Save the Web content to a file.  
urlwrite(filex,'contains_simulink.html','get',params);
```

MATLAB saves the Web page as `contains_simulink.html`.

Creating and Decompressing Zip Archives

Using the `zip` and `unzip` functions, you can compress and decompress files and folders. The `zip` function compresses files or folders into a zip archive. The `unzip` function decompresses zip archives.

Example – Using the `zip` Function

Again building on the example from previous sections, the following code creates a zip archive of the retrieved Web page:

```
% Locate the list of files at the MATLAB Central File Exchange
% uploaded within the past 7 days, that contain "Simulink."
filex = 'http://www.mathworks.com/matlabcentral/fileexchange/';
params = {'duration','7','term','simulink'};

% Save the Web content to a file.
urlwrite(filex,'contains_simulink.html','get',params);

% Create a zip archive of the retrieved Web page.
zip('simulink_matches.zip','contains_simulink.html');
```

Sending E-Mail

To send an e-mail from MATLAB, use the `sendmail` function. You can also attach files to an e-mail, which lets you mail files directly from MATLAB. To use `sendmail`, you must first set up your e-mail address and your SMTP server information with the `setpref` function.

The `setpref` function defines two mail-related preferences:

- E-mail address: This preference sets your e-mail address that will appear on the message. Here is an example of the syntax:

```
setpref('Internet','E_mail','youraddress@yourserver.com');
```

- SMTP server: This preference sets your outgoing SMTP server address, which can be almost any e-mail server that supports the Post Office Protocol (POP) or the Internet Message Access Protocol (IMAP). Here is an example of the syntax:

```
setpref('Internet','SMTP_Server','mail.server.network');
```

You should be able to find your outgoing SMTP server address in your e-mail account settings in your e-mail client application. You can also contact your system administrator for the information.

Note The `sendmail` function does not support e-mail servers that require authentication.

Once you have properly configured MATLAB, you can use the `sendmail` function. The `sendmail` function requires at least two arguments: the recipient's e-mail address and the e-mail subject:

```
sendmail('recipient@someserver.com','Hello From MATLAB!');
```

You can supply multiple e-mail addresses using a cell array of strings, such as:

```
sendmail({'recipient@someserver.com', ...  
'recipient2@someserver.com'}, 'Hello From MATLAB!');
```

You can also specify a message body with the `sendmail` function, such as:

```
sendmail('recipient@someserver.com', 'Hello From MATLAB!', ...
'Thanks for using sendmail.');
```

In addition, you can also attach files to an e-mail using the `sendmail` function, such as:

```
sendmail('recipient@someserver.com', 'Hello from MATLAB!', ...
'Thanks for using sendmail.', 'C:\yourFileSystem\message.txt');
```

You cannot attach a file without including a message. However, the message can be empty. You can also attach multiple files to an e-mail with the `sendmail` function, such as:

```
sendmail('recipient@someserver.com', 'Hello from MATLAB!', ...
'Thanks for using sendmail.', ...
{'C:\yourFileSystem\message.txt',...
'C:\yourFileSystem\message2.txt'});
```

Example – Using the sendmail Function

The following example sends e-mail with the retrieved Web page archive attached:

```
% NOTE: CHANGE THESE 2 LINES OF CODE TO REFLECT YOUR SETTINGS.
mySMTP = 'mail.server.network';
myEmail = 'youraddress@yourserver.com';

% Set your e-mail and SMTP server address in MATLAB.
setpref('Internet','SMTP_Server',mySMTP);
setpref('Internet','E_mail',myEmail);

% Locate the list of files at the MATLAB Central File Exchange
% uploaded within the past 7 days, that contain "Simulink."
filex = 'http://www.mathworks.com/matlabcentral/fileexchange/';
params = {'duration','7','term','simulink'};

% Save the Web content to a file.
urlwrite(filex,'contains_simulink.html','get',params);
```

```
% Create a zip archive of the retrieved Web page.
zip('simulink_matches.zip','contains_simulink.html');

% Send an e-mail (to yourself) with the zip archive attached.
recipient = myEmail;
subj = 'List of New Simulink Files';
msg = ...
    'Attached: new Similink files uploaded to MATLAB Central.';
attFile = 'simulink_matches.zip';
sendmail(recipient,subj,msg,attFile);
```

Performing FTP File Operations

From MATLAB, you can connect to an FTP server to perform remote file operations. The following procedure uses a public MathWorks® FTP server (ftp.mathworks.com). To perform any file operation on an FTP server, follow these steps:

- 1** Connect to the server using the `ftp` function.
- 2** Perform file operations using appropriate MATLAB FTP functions. For all operations, specify the server object. For a complete list of functions, see the FTP reference page.
- 3** When you finish working on the server, close the connection object using the `close` function.

Example – Retrieving a File from an FTP Server

List the contents of the MathWorks FTP server and retrieve a file named `README`. To view the file, use the `type` function.

```
tmw = ftp('ftp.mathworks.com');
dir(tmw)

mget(tmw, 'README');
type README
```

`README` contains the following text:

```
Welcome to the MathWorks FTP site!
The MathWorks FTP site has a new structure:

/incoming - where you upload files to
/outgoing - where you pick up files from
```

NOTE: Files in the above directories will be removed after 30 days.

You may also want to visit the MathWorks Web site at

```
http://www.mathworks.com
```


Send questions/comments/suggestions to ftpadmin@mathworks.com

View the contents of the pub folder:

```
cd(tmw, 'pub')  
dir(tmw)  
  
% Close the connection  
close(tmw)
```


A

- access modes
 - HDF4 files 1-97
- ASCII data
 - exporting 2-8
 - exporting delimited data 2-8
 - exporting with `diary` function 2-11
 - formats 1-18
 - importing 1-18
 - importing mixed numeric and nonnumeric data 1-23
 - importing nonrectangular data 1-24
 - importing numeric data 1-19
 - importing numeric data with headers 1-20
 - reading formatted text 1-27
 - saving 2-9
 - writing 2-8
- ASCII files
 - reading 1-18
- attributes
 - retrieving from HDF4 files 1-98
 - writing to an HDF4 file 2-50

B

- binary data
 - controlling class of values read 1-118
 - using the Import Wizard 1-5
 - writing to 2-72

C

- `cdfepoch` object
 - representing CDF time values 1-48
- characters
 - used as delimiters 1-18
- classes
 - precision 1-118
 - reading files 1-118
 - specifying for input 1-118

- clipboard
 - importing binary data 1-2
- Common Data Format (CDF)
 - combining records to improve read performance 1-46
 - converting CDF epoch values to MATLAB datenum values 1-47
 - reading CDF files using the high-level functions 1-46
 - reading metadata from CDF files using high-level functions 1-45
 - representing time values 1-48
 - speeding up read operations 1-46
 - writing data to CDF files 2-26

D

- delimiters
 - defined 1-18
- `diary` 2-11
- directories
 - temporary 2-79
- downloading files 4-2

E

- Earth Observing System (EOS) 1-73 2-43
- end of file 1-30
- EOS (Earth Observing System)
 - sources of information 1-73 2-43
- exporting
 - ASCII data 2-8
 - in HDF4 format 2-43
 - in HDF5 format 2-33

F

- `feof` 1-119
- file exchange
 - over Internet 4-1
- file I/O

- audio/video files
 - exporting 2-70
 - importing 1-107
 - graphics files
 - exporting 2-54
 - importing 1-103
 - internet 4-1
 - downloading from web 4-2
 - FTP operations 4-8
 - sending e-mail 4-5
 - ZIP files 4-4
 - low-level functions
 - ASCII files:exporting 2-12
 - ASCII files:importing 1-27
 - binary files:exporting 2-72
 - binary files:importing 1-116
 - text files:exporting 2-12
 - MAT-files
 - exporting 2-2
 - MATLAB HDF4 utility API 2-52
 - memory mapping. *See* memory mapping
 - overview
 - toolboxes for importing data 1-4
 - scientific formats
 - FITS files 1-56
 - HDF4 and HDF-DOS files 1-91
 - HDF4 files 1-73 1-95 2-43
 - HDF5 files 1-58
 - spreadsheet files
 - Microsoft Excel 1-38
 - supported file types 1-2
 - text files
 - exporting 2-8
 - importing 1-18
 - using Import Wizard 1-5
 - file import and export
 - supported file types 1-2
 - file operations
 - FTP 4-8
 - file types
 - supported by MATLAB 1-2
 - files
 - ASCII
 - reading 1-18
 - reading formatted text 1-27
 - writing 2-8
 - beginning of 1-120
 - binary
 - classes 1-118
 - controlling class values read 1-118
 - reading 1-116
 - writing to 2-72
 - current position 1-120
 - end of 1-30
 - MAT 2-2
 - permissions 2-15 2-73
 - position 1-119
 - specifying delimiter used in ASCII files 1-18
 - temporary 2-79
 - text
 - reading 1-18
 - FITS. *See* Flexible Image Transport System
 - Flexible Image Transport System (FITS)
 - reading 1-56
 - reading data 1-56
 - reading metadata 1-57
 - fread 1-118
 - frewind 1-119
 - fseek 1-119
 - ftell 1-119
 - FTP file operations 4-8
 - fwrite 2-72
- G**
- global attributes
 - HDF4 files 1-98

H

HDF Import Tool

- using 1-74
- using subsetting options 1-78

HDF-EOS

- Earth Observing System 1-73 2-43

HDF4 1-73

- closing a data set 2-51
 - closing a file 2-52
 - closing all open identifiers 2-53
 - closing data sets 1-101
 - creating a file 2-45
 - creating data sets 2-46
 - exporting in HDF4 format 2-43
 - importing data 1-92
 - importing subsets of data 1-77
 - listing all open identifiers 2-52
 - low-level functions
 - overview 1-95
 - mapping HDF4 syntax to MATLAB
 - syntax 1-95 2-44
 - MATLAB utility API 2-52
 - opening files 1-97
 - overview 1-73
 - reading data 1-100
 - reading data set metadata 1-99
 - reading data sets 1-99
 - reading global attributes 1-98
 - reading metadata 1-97
 - selecting data sets to import 1-76
 - specifying file access modes 1-97
 - using `hdfinfo` to import metadata 1-91
 - using high-level functions
 - overview 1-91
 - using predefined attributes 2-50
 - using the HDF Import Tool 1-74
 - writing data 2-43 2-47
 - writing metadata 2-50
- See also* HDF5

HDF5 1-58

- exporting data in HDF5 format 2-33

low-level functions

- mapping HDF5 data types to MATLAB
 - data types 2-37
- mapping HDF5 syntax to MATLAB
 - syntax 2-35
- reading and writing data 2-39

overview 1-58

- using `hdf5info` to read metadata 1-59
- using `hdf5read` to import data 1-62
- using high-level functions 1-58
- using low-level functions 2-34

See also HDF4

Hierarchical Data Format. *See* HDF4. *See* HDF5

I

Import Data option 1-5

Import Wizard

- importing binary data 1-5

importing

- ASCII data 1-18
- HDF4 data 1-91
 - from the command line 1-95
 - selecting HDF4 data sets 1-76
 - subsets of HDF4 data 1-77

Internet functions 4-1

L

large data sets

- reading 1-25

Mmapping memory. *See* memory mapping

memory mapping

- demonstration 3-40
- `memmapfile` class
 - class constructor 3-9
 - class methods 3-37

- class properties 3-7
- defined 3-7
- Filename property 3-12
- Format property 3-14
- Offset property 3-13
- Repeat property 3-20
- supported formats 3-19
- Writable property 3-21

overview 3-2

- benefits of 3-2
- byte ordering 3-5
- limitations of 3-4
- when to use 3-5

reading from file 3-23

removing map 3-39

selecting file to map 3-12

setting access privileges 3-21

setting extent of map 3-20

setting start of map 3-13

specifying classes in file 3-14

supported classes 3-19

writing to file 3-29

N

netCDF

- mapping netCDF syntax to MATLAB syntax 1-51
- MATLAB support 1-51
- reading data 1-51 to 1-52

Network Common Data Form

- see netCDF 1-51

O

opening

- files
 - HDF4 files 1-97
 - permissions 2-15 2-73

P

Paste Special option 1-2

permission strings 2-15 2-73

precision

- classes 1-118

R

reading

- HDF4 data 1-91
 - from the command line 1-95
- selecting HDF4 data sets 1-76
- subsets of HDF4 data 1-77

T

tempdir 2-79

tempname 2-79

temporary files

- creating 2-79

text files

- reading 1-18

V

value

- class 1-118

W

Web content access 4-2

writing

- ASCII data 2-8
- HDF4 data 2-47
- in HDF4 format 2-43
- in HDF5 format 2-33