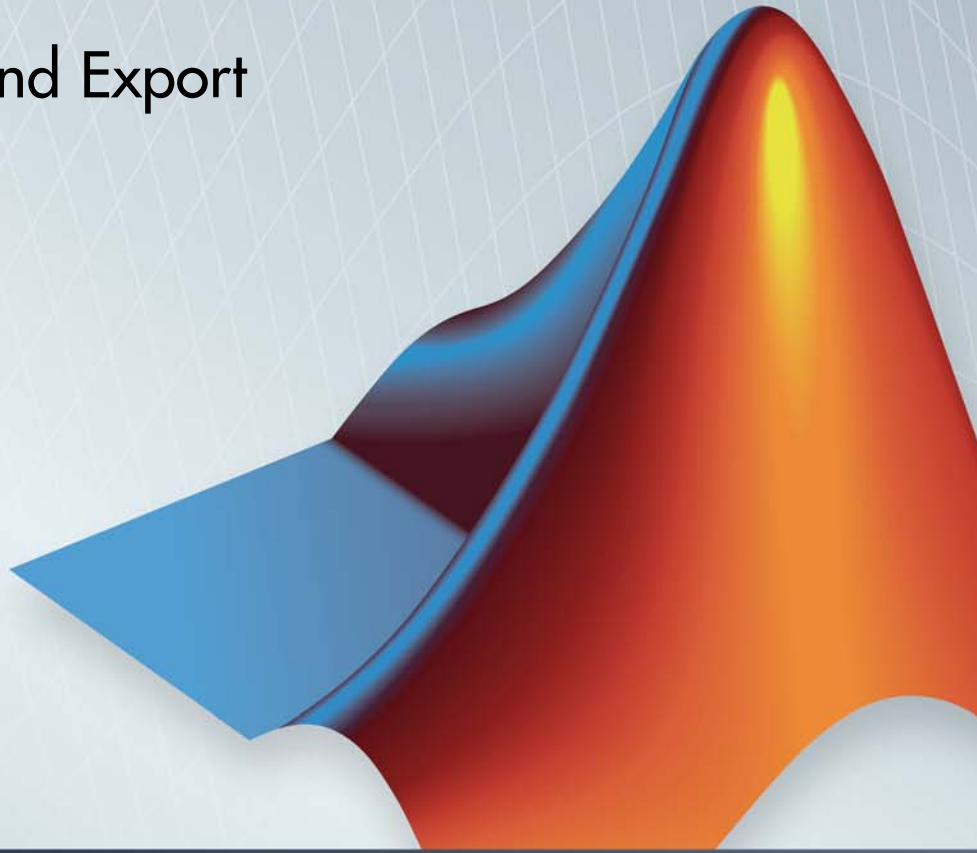


MATLAB®

Data Import and Export

R2014a



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 2009–2014 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)
April 2011	Online only	Revised for MATLAB 7.12 (Release 2011a)
September 2011	Online only	Revised for MATLAB 7.13 (Release 2011b)
March 2012	Online only	Revised for MATLAB 7.14 (Release 2012a)
September 2012	Online only	Revised for MATLAB 8.0 (Release 2012b)
March 2013	Online only	Revised for MATLAB 8.1 (Release 2013a)
September 2013	Online only	Revised for MATLAB 8.2 (Release 2013b)
March 2014	Online only	Revised for MATLAB 8.3 (Release 2014a)

File Opening, Loading, and Saving

1

Supported File Formats for Import and Export	1-2
Methods for Importing Data	1-7
Tools that Import Multiple File Formats	1-7
Importing Specific File Formats	1-8
Importing Data with Low-Level I/O	1-8
Import Images, Audio, and Video Interactively	1-9
Viewing the Contents of a File	1-9
Specifying Variables	1-10
Generating Reusable MATLAB Code	1-11
Import or Export a Sequence of Files	1-13
View the Contents of a MAT-File	1-14
Load Parts of Variables from MAT-Files	1-15
Load Using the matfile Function	1-15
Load from Variables with Unknown Names	1-16
Avoid Repeated File Access	1-17
Avoid Inadvertently Loading Entire Variables	1-18
Partial Loading Requires Version 7.3 MAT-Files	1-18
Save Parts of Variables to MAT-Files	1-20
Save Using the matfile Function	1-20
Partial Saving Requires Version 7.3 MAT-Files	1-22
Save Structure Fields as Separate Variables	1-23
MAT-File Versions	1-24
Default Version	1-24
Overriding the Default MAT-File Version	1-24

Speeding Up Save and Load Operations	1-25
File Size Increases Unexpectedly When Growing Array	1-27
Loading Variables within a Function	1-29
Create Temporary Files	1-30

Text Files

2

Ways to Import Text Files	2-2
Select Text File Data Using Import Tool	2-4
Select Data Interactively	2-4
Import Data from Multiple Text Files	2-7
Import Dates and Times from Text Files	2-9
Import Numeric Data from Text Files	2-10
Import Comma-Separated Data	2-10
Import Delimited Numeric Data	2-11
Import Mixed Text and Numeric Data from Text Files	2-13
Read File with Column Names	2-13
Read File Without Column Names	2-14
Import Large Text File Data in Blocks	2-16
Import Data from a Nonrectangular Text File	2-24
Write to Delimited Data Files	2-26
Export Numeric Array to ASCII File	2-26
Export Table to Text File	2-28

Export Cell Array to Text File	2-29
Write to a Diary File	2-32

Spreadsheets

3

Ways to Import Spreadsheets	3-2
Import Data from Spreadsheets	3-2
Paste Data from Clipboard	3-3
 Select Spreadsheet Data Using Import Tool	 3-4
Select Data Interactively	3-4
Import Data from Multiple Spreadsheets	3-6
 Import a Worksheet or Range	 3-8
Read Column-Oriented Data into Table	3-8
Read Numeric and Text Data into Arrays	3-9
Get Information about a Spreadsheet	3-11
 Import All Worksheets from a File	 3-12
Import Numeric Data from All Worksheets	3-12
Import Data and Headers from All Worksheets	3-12
 System Requirements for Importing Spreadsheets	 3-15
Importing Spreadsheets with Excel for Windows	3-15
Importing Spreadsheets Without Excel for Windows	3-15
 When to Convert Dates from Excel Files	 3-16
MATLAB and Excel Dates	3-16
Import an Excel File with Numeric Dates	3-17
Export to an Excel File with Numeric Dates	3-18
 Export to Excel Spreadsheets	 3-19
Write Tabular Data to Spreadsheet File	3-19
Write Numeric and Text Data to Spreadsheet File	3-20
Disable Warning When Adding New Worksheet	3-21

Supported Excel File Formats	3-21
Format Cells in Excel Files	3-21

Low-Level File I/O

4

Import Text Data Files with Low-Level I/O	4-2
Overview	4-2
Reading Data in a Formatted Pattern	4-3
Reading Data Line-by-Line	4-6
Testing for End of File (EOF)	4-7
Opening Files with Different Character Encodings	4-9
Import Binary Data with Low-Level I/O	4-11
Low-Level Functions for Importing Data	4-11
Reading Binary Data in a File	4-12
Reading Portions of a File	4-14
Reading Files Created on Other Systems	4-17
Opening Files with Different Character Encodings	4-18
Export to Text Data Files with Low-Level I/O	4-19
Writing to Text Files	4-19
Appending or Overwriting Existing Files	4-22
Opening Files with Different Character Encodings	4-25
Export Binary Data with Low-Level I/O	4-26
Low-Level Functions for Exporting Data	4-26
Writing Binary Data to a File	4-27
Overwriting or Appending to an Existing File	4-27
Creating a File for Use on a Different System	4-29
Opening Files with Different Character Encodings	4-30
Writing and Reading Complex Numbers	4-31

5

Importing Images	5-2
Getting Information about Image Files	5-2
Reading Image Data and Metadata from TIFF Files	5-3
Exporting to Images	5-6
Exporting Image Data and Metadata to TIFF Files	5-6

Scientific Data

6

Importing CDF Files	6-2
Overview	6-2
High-Level CDF Import Functions	6-2
Using the CDF Library Low-Level Functions to Import Data	6-6
Exporting to CDF Files	6-10
Importing NetCDF Files and OPeNDAP Data	6-12
Overview	6-12
Using the MATLAB High-Level NetCDF Functions to Import Data	6-12
Using the MATLAB Low-Level NetCDF Functions to Import Data	6-14
Troubleshooting OPeNDAP Connections	6-20
Exporting to NetCDF Files	6-21
Overview	6-21
Using the NetCDF High-Level Functions to Export Data	6-21
Using the NetCDF Low-Level Functions to Export Data ..	6-26
Importing Flexible Image Transport System (FITS) Files	6-30

Importing HDF5 Files	6-32
Overview	6-32
Using the High-Level HDF5 Functions to Import Data ...	6-32
Using the Low-Level HDF5 Functions to Import Data	6-39
Exporting to HDF5 Files	6-40
Overview	6-40
Using the MATLAB High-Level HDF5 Functions to Export Data	6-40
Using the MATLAB Low-Level HDF5 Functions to Export Data	6-41
Import HDF4 Files Programatically	6-52
Overview	6-52
Using the MATLAB HDF4 High-Level Functions	6-52
Map HDF4 to MATLAB Syntax	6-57
Import HDF4 Files Using Low-Level Functions	6-59
Import HDF4 Files Interactively	6-63
Step 1: Opening an HDF4 File in the HDF Import Tool ..	6-63
Step 2: Selecting a Data Set in an HDF File	6-65
Step 3: Specifying a Subset of the Data (Optional)	6-66
Step 4: Importing Data and Metadata	6-67
Step 5: Closing HDF Files and the HDF Import Tool	6-68
Using the HDF Import Tool Subsetting Options	6-68
About HDF4 and HDF-EOS	6-81
Export to HDF4 Files	6-82
Write MATLAB Data to HDF4 File	6-82
Manage HDF4 Identifiers	6-84

7

Read and Get Information About Audio Files 7-2

Record and Play Audio 7-3

 Record Audio 7-3

 Play Audio 7-6

 Record or Play Audio within a Function 7-7

Get Information about Video Files 7-9

Read Video Files 7-10

 Import Video Data from a File 7-10

 Display Video Frame with Colormap 7-10

 Process Frames of a Video File 7-10

 Read Variable Frame Rate Video 7-11

Supported Video File Formats 7-13

 What Are Video Files? 7-13

 Formats That VideoReader Supports 7-13

 View Codec Associated with Video File 7-14

 Troubleshooting: Errors Reading Video File 7-15

Convert Between Image Sequences and Video 7-16

Export to Audio and Video 7-20

 Export to Audio Files 7-20

 Export Video to AVI Files 7-20

Characteristics of Audio Files 7-22

8

Importing XML Documents 8-2

What Is an XML Document Object Model (DOM)?	8-2
Example — Finding Text in an XML File	8-3
Exporting to XML Documents	8-6
Creating an XML File	8-6
Updating an Existing XML File	8-8

Memory-Mapping Data Files

9

Overview of Memory-Mapping	9-2
What Is Memory-Mapping?	9-2
Benefits of Memory-Mapping	9-2
When to Use Memory-Mapping	9-4
Maximum Size of a Memory Map	9-5
Byte Ordering	9-6
Map File to Memory	9-7
Create a Simple Memory Map	9-7
Specify Format of Your Mapped Data	9-8
Map Multiple Data Types and Arrays	9-9
Select File to Map	9-11
Read Mapped File	9-12
Write to Mapped File	9-19
Write to Memory Mapped as Numeric Array	9-19
Write to Memory Mapped as Scalar Structure	9-21
Write to Memory Mapped as Nonscalar Structure	9-21
Syntaxes for Writing to Mapped File	9-23
Work with Copies of Your Mapped Data	9-24
Delete Memory Map	9-27
Ways to Delete a Memory Map	9-27
The Effect of Shared Data Copies On Performance	9-27
Share Memory Between Applications	9-28

10

Downloading Web Content and Files	10-2
Example — Using the urlread Function	10-2
Example — Using the urlwrite Function	10-3
Sending Email	10-4
Example — Using the sendmail Function	10-5
Performing FTP File Operations	10-7
Example — Retrieving a File from an FTP Server	10-7
Display Hyperlinks in the Command Window	10-9
Creating Hyperlinks to Web Pages	10-9
Transferring Files Using FTP	10-9

Install and Use Raspberry Pi Hardware

11

Install Support for Raspberry Pi Hardware	11-3
Install the Support Package	11-3
Complete Additional Setup Tasks	11-5
Guidelines for Entering Static IP Settings	11-17
Open Interactive Examples	11-18
Connecting to Raspberry Pi Hardware	11-20
Connect to Raspberry Pi Hardware	11-22
Create Connection to One Board	11-22
Create Connection to a Board That Has Different Settings	11-23
Troubleshoot Connecting to Raspberry Pi Hardware ..	11-25

Connection Timed Out	11-25
Host Does Not Exist	11-25
Active Connection Already Exists	11-26
Get the IP Address of the Raspberry Pi Hardware	11-27
Hear the Spoken IP Address	11-27
Show the IP Address on a Display	11-27
The Raspberry Pi LED	11-29
Turn the Raspberry Pi LED On and Off	11-31
Flash the Raspberry Pi LED in Response to an Input ..	11-34
The Raspberry Pi GPIO Pins	11-35
Use the Raspberry Pi GPIO Pins as Digital Inputs and Outputs	11-36
Troubleshoot Raspberry Pi GPIO Pins	11-40
Error Using raspi/writeDigitalPin	11-40
Error Using raspi/readDigitalPin	11-40
Unexpected Digital Pin Number	11-41
The Raspberry Pi Serial Port	11-42
Use the Raspberry Pi Serial Port to Connect to a Device	11-43
Troubleshoot the Raspberry Pi Serial Port	11-48
Missing or Garbled Data	11-48
The Raspberry Pi I2C Interface	11-49
Use the Raspberry Pi I2C Interface to Connect to a Device	11-50
Troubleshoot the Raspberry Pi I2C Interface	11-54

The Raspberry Pi SPI Interface	11-55
Use the Raspberry Pi SPI Interface to Connect to a Device	11-57
The Raspberry Pi Camera Board	11-61
Use the Raspberry Pi Camera Board to Capture Images and Video	11-63
Troubleshoot the Raspberry Pi Camera Board	11-65
The Raspberry Pi Linux Command Interface	11-66
Run Linux Commands on Raspberry Pi Hardware	11-67
Troubleshoot Running Linux Commands on Raspberry Pi Hardware	11-70
Management of Raspberry Pi Files	11-71
Manage Raspberry Pi Files	11-72
Troubleshoot Managing Raspberry Pi Files	11-73

Webcam Support in MATLAB

12

Webcam Acquisition Overview	12-2
Webcam Support	12-2
Supported Platforms	12-3
Connecting to Webcams	12-4
Acquiring Images from Webcams	12-6

Creating a Webcam Object	12-6
Acquiring Webcam Images	12-10
Acquiring Webcam Images in a Loop	12-14
Supported Functions for Webcam	12-16
Setting Properties for Webcam Acquisition	12-17
Installing the Webcam Support Package	12-22

File Opening, Loading, and Saving

- “Supported File Formats for Import and Export” on page 1-2
- “Methods for Importing Data” on page 1-7
- “Import Images, Audio, and Video Interactively” on page 1-9
- “Import or Export a Sequence of Files” on page 1-13
- “View the Contents of a MAT-File” on page 1-14
- “Load Parts of Variables from MAT-Files” on page 1-15
- “Save Parts of Variables to MAT-Files” on page 1-20
- “Save Structure Fields as Separate Variables” on page 1-23
- “MAT-File Versions” on page 1-24
- “File Size Increases Unexpectedly When Growing Array” on page 1-27
- “Loading Variables within a Function” on page 1-29
- “Create Temporary Files” on page 1-30

Supported File Formats for Import and Export

The following table shows the file formats that you can import and export from the MATLAB® application.

In addition to the functions in the table, you also can use the `importdata` function, or import these file formats interactively, with the following exceptions:

- `importdata` and interactive import do not support H5 and netCDF files.
- `importdata` does not support HDF files.

File Content	Extension	Description	Import Function	Export Function
MATLAB formatted data	MAT	Saved MATLAB workspace	<code>load</code>	<code>save</code>
		Partial access of variables in MATLAB workspace	<code>matfile</code>	<code>matfile</code>
Text	any, including: CSV TXT	Comma delimited numbers	<code>csvread</code>	<code>csvwrite</code>
		Delimited numbers	<code>dlmread</code>	<code>dlmwrite</code>
		Delimited numbers, or a mix of strings and numbers	<code>textscan</code>	<code>none</code>
		Column-oriented delimited numbers or a mix of strings and numbers	<code>readtable</code>	<code>writetable</code>

File Content	Extension	Description	Import Function	Export Function
Spreadsheet	XLS XLSX XLSM	Worksheet or range of spreadsheet	xlsread	xlswrite
	XLXB (Systems with Microsoft® Excel® for Windows® only) XLTM (import only) XLTX (import only) ODS (Systems with COM interface)	Column-oriented data in worksheet or range of spreadsheet	readtable	writetable
Extensible Markup Language	XML	XML-formatted text	xmlread	xmlwrite
Data Acquisition Toolbox™ file	DAQ	Data Acquisition Toolbox	daqread	none
Scientific data	CDF	Common Data Format	See <code>cdflib</code>	See <code>cdflib</code>
	FITS	Flexible Image Transport System	See “FITS Files”	See “FITS Files”
	HDF	Hierarchical Data Format, version 4, or HDF-EOS v. 2	See “HDF4 Files”	See “HDF4 Files”
	H5	HDF or HDF-EOS, version 5	See “HDF5 Files”	See “HDF5 Files”
	NC	Network Common Data Form (netCDF)	See <code>netcdf</code>	See <code>netcdf</code>

File Content	Extension	Description	Import Function	Export Function
Image	BMP	Windows Bitmap	imread	imwrite
	GIF	Graphics Interchange Format		
	HDF	Hierarchical Data Format		
	JPEG JPG	Joint Photographic Experts Group		
	JP2 JPF JPX J2C J2K	JPEG 2000		
	PBM	Portable Bitmap		
	PCX	Paintbrush		
	PGM	Portable Graymap		
	PNG	Portable Network Graphics		
	PNM	Portable Any Map		
	PPM	Portable Pixmap		
	RAS	Sun™ Raster		
	TIFF TIF	Tagged Image File Format		
	XWD	X Window Dump		
	CUR	Windows Cursor resources	imread	none
FITS FTS	Flexible Image Transport System			
ICO	Windows Icon resources			

File Content	Extension	Description	Import Function	Export Function
Audio (all platforms)	AU SND	NeXT/Sun sound	audioread	audiowrite
	FLAC	Free Lossless Audio Codec		
	OGG	Ogg Vorbis		
	WAV	Microsoft WAVE sound		
Audio (Windows)	M4A MP4	MPEG-4	audioread	audiowrite
	any	Formats supported by Microsoft Media Foundation	audioread	none
Audio (Mac)	M4A MP4	MPEG-4	audioread	audiowrite
Audio (Linux [®])	any	Formats supported by GStreamer	audioread	none
Video (all platforms)	AVI	Audio Video Interleave	VideoReader	VideoWriter
	MJ2	Motion JPEG 2000		
Video (Windows)	MPG	MPEG-1	VideoReader	none
	ASF ASX WMV	Windows Media [®]		
	any	Formats supported by Microsoft DirectShow [®]		
Video (Windows 7 or later)	MP4 M4V	MPEG-4	VideoReader	VideoWriter
	MOV	QuickTime	VideoReader	none
	any	Formats supported by Microsoft Media Foundation		

File Content	Extension	Description	Import Function	Export Function
Video (Mac)	MP4 M4V	MPEG-4	VideoReader	VideoWriter
	MPG	MPEG-1	VideoReader	none
	MOV	QuickTime		
	any	Formats supported by QuickTime, including .3gp, .3g2, and .dv		
Video (Linux)	any	Formats supported by your installed GStreamer plug-ins, including .ogg	VideoReader	none

Methods for Importing Data

In this section...

“Tools that Import Multiple File Formats” on page 1-7

“Importing Specific File Formats” on page 1-8


“Importing Data with Low-Level I/O” on page 1-8

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.


Tools that Import Multiple File Formats

You can import data into MATLAB from a disk file or the system clipboard interactively.

To import data from a file, do one of the following:

- On the **Home** tab, in the **Variable** section, select **Import Data** .
- Double-click a file name in the Current Folder browser.
- Call `uiimport`.

To import data from the clipboard, do one of the following:

- On the Workspace browser title bar, click , and then select **Paste**.
- Call `uiimport`.

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

For a complete list of the formats you can import interactively or with `importdata`, see “Supported File Formats for Import and Export” on page 1-2.

Importing Specific File Formats

MATLAB includes functions tailored to import specific file formats. Consider using format-specific functions instead of importing data interactively when 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. Some format-specific functions allow you to request multiple optional outputs. This option is not available when you import interactively.

For a complete list of the format-specific functions, see “Supported File Formats for Import and Export” on page 1-2.

For binary data files, consider “Overview of Memory-Mapping” on page 9-2. Memory-mapping enables you to access file data using standard MATLAB indexing operations.

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.

Importing Data with Low-Level I/O

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:

- “Import Text Data Files with Low-Level I/O” on page 4-2
- “Import Binary Data with Low-Level I/O” on page 4-11

Import Images, Audio, and Video Interactively

In this section...

“Viewing the Contents of a File” on page 1-9

“Specifying Variables” on page 1-10

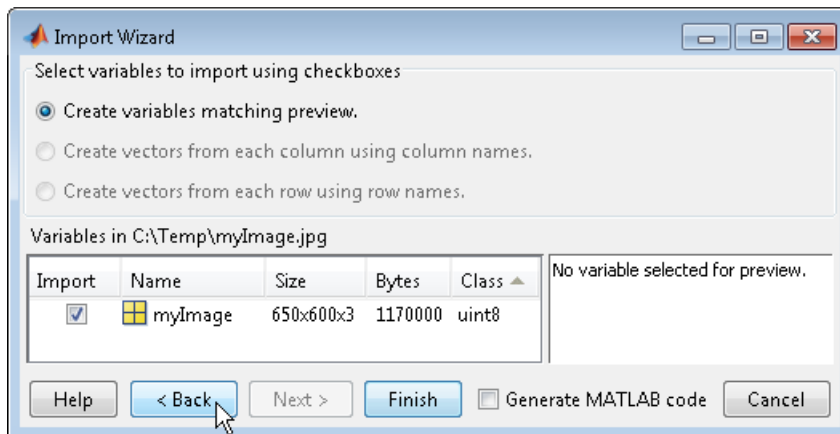
“Generating Reusable MATLAB Code” on page 1-11

Note For information on importing text files, see “Select Text File Data Using Import Tool” on page 2-4. For information on importing spreadsheets, see “Select Spreadsheet Data Using Import Tool” on page 3-4. For information on importing HDF4 files, see “Import HDF4 Files Interactively” on page 6-63.

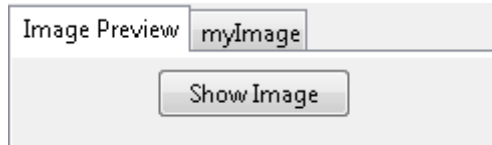
Viewing the Contents of a File

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

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:

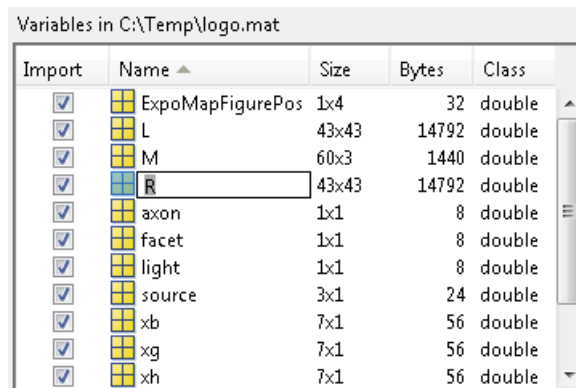
- “Renaming or Deselecting Variables” on page 1-10
- “Importing to a Structure Array” on page 1-11

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 name, select the name and type a new one.



Import	Name ^	Size	Bytes	Class
<input checked="" type="checkbox"/>	ExpoMapFigurePos	1x4	32	double
<input checked="" type="checkbox"/>	L	43x43	14792	double
<input checked="" type="checkbox"/>	M	60x3	1440	double
<input checked="" type="checkbox"/>	R	43x43	14792	double
<input checked="" type="checkbox"/>	axon	1x1	8	double
<input checked="" type="checkbox"/>	facet	1x1	8	double
<input checked="" type="checkbox"/>	light	1x1	8	double
<input checked="" type="checkbox"/>	source	3x1	24	double
<input checked="" type="checkbox"/>	xb	7x1	56	double
<input checked="" type="checkbox"/>	xg	7x1	56	double
<input checked="" type="checkbox"/>	xh	7x1	56	double

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 sample file `durer.mat` contains three variables: `X`, `map`, 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 Melancolia.  
Can you find the matrix?
```

For more information, see “Access Data in a Structure Array”.

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 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.

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

Import or Export 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”.)

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
```

View 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 example 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 1-24.

Load Parts of Variables from MAT-Files

In this section...

“Load Using the `matfile` Function” on page 1-15

“Load from Variables with Unknown Names” on page 1-16

“Avoid Repeated File Access” on page 1-17

“Avoid Inadvertently Loading Entire Variables” on page 1-18

“Partial Loading Requires Version 7.3 MAT-Files” on page 1-18

Load Using the `matfile` Function

This example shows how to load part of a variable from an existing MAT-file.

To run the code in this example, create a Version 7.3 MAT-file with two variables.

```
A = rand(5);  
B = magic(10);  
save example.mat A B -v7.3;  
clear A B
```

Construct a `matlab.io.MatFile` object that can load parts of variables from the file, `example.mat`.

```
example = matfile('example.mat')
```

```
example =
```

```
matlab.io.MatFile
```

```
Properties:
```

```
Properties.Source: C:\Documents\MATLAB\example.mat
```

```
Properties.Writable: false
```

```
A: [5x5 double]
```

```
B: [10x10 double]
```

The `matfile` function creates a `matlab.io.MatFile` object that corresponds to a MAT-file and displays the properties of the `matlab.io.MatFile` object.

Load the first column of `B` from `example.mat` into variable `firstColB`.

```
firstColB = example.B(:,1);
```

When you index into objects associated with Version 7.3 MAT-files, MATLAB loads only the part of the variable that you specify.

By default, `matfile` only allows loading from existing MAT-files. To enable saving, call `matfile` with the `Writable` parameter.

```
example = matfile('example.mat', 'Writable', true);
```

Alternatively, construct the object and set `Properties.Writable` in separate steps.

```
example = matfile('example.mat');  
example.Properties.Writable = true;
```

Load from Variables with Unknown Names

This example shows how to dynamically access variables, whose names are not always known. Consider the example MAT-file, `topography.mat`, that contains one or more arrays with unknown names.

Construct a `matlab.io.MatFile` object that corresponds to the file, `topography.mat`. Call `who` to get the variable names in the file.

```
matObj = matfile('topography.mat');  
varlist = who(matObj)
```

```
varlist =  
  
    'topo'  
    'topolegend'  
    'topomap1'  
    'topomap2'
```

`varlist` is a cell array containing the names of the four variables in `topography.mat`.

The third and fourth variables, `topomap1` and `topomap2`, are both arrays containing topography data. Load the elevation data from the third column of each variable into a field of the structure array, `S`. For each field, specify a field name that is the original variable name prefixed by `elevationOf_`. Access the data in each variable as properties of `matObj`. Because `varName` is a variable, enclose it in parentheses.

```
for index = 3:4
    varName = varlist{index};
    S(1).(['elevationOf_',varName]) = matObj.(varName)(:,3);
end
```

View the contents of the structure array, `S`.

`S`

`S =`

```
    elevationOf_topomap1: [64x1 double]
    elevationOf_topomap2: [128x1 double]
```

`S` has two fields, `elevationOf_topomap1` and `elevationOf_topomap2`, each containing a column vector.

Avoid Repeated File Access

The primary advantage of `matfile` over the `load` function is that you can process parts of very large data sets that are otherwise too large to fit in memory. When working with these large variables, read and write as much data into memory as possible at a time. Otherwise, repeated file access negatively impacts the performance of your code.

For example, suppose a variable in your file contains many rows and columns, and loading a single row requires most of the available memory. To calculate the mean of the entire data set, calculate the mean of each row, and then find the overall mean.

```
example = matfile('example.mat');
[nrows, ncols] = size(example,'B');

avgs = zeros(1, nrows);
```

```
for idx = 1:nrows
    avgs(idx) = mean(example.B(idx,:));
end
overallAvg = mean(avgs);
```

Avoid Inadvertently Loading Entire Variables

When you do not know the size of a large variable in a MAT-file, and want to load parts of that variable at a time, do not use the `end` keyword. Rather, call the `size` method for `matlab.io.MatFile` objects. For example, this code

```
[nrows,ncols] = size(example,'B');
lastColB = example.B(:,ncols);
```

requires less memory than

```
lastColB = example.B(:,end);
```

which temporarily loads the entire contents of `B`. For very large variables, loading takes a long time or generates Out of Memory errors.

Similarly, any time you refer to a variable with syntax of the form `matObj.varName`, such as `example.B`, MATLAB temporarily loads the entire variable into memory. Therefore, make sure to call the `size` method for `matlab.io.MatFile` objects with syntax such as

```
[nrows,ncols] = size(example,'B');
```

rather than passing the entire contents of `example.B` to the `size` function,

```
[nrows,ncols] = size(example.B);
```

The difference in syntax is subtle, but significant.

Partial Loading Requires Version 7.3 MAT-Files

Any load or save operation that uses a `matlab.io.MatFile` object associated with a Version 7 or earlier MAT-file temporarily loads the entire variable into memory.


The `matfile` function creates files in Version 7.3 format. For example, this code

```
newfile = matfile('newfile.mat');
```

creates a MAT-file that supports partial loading and saving.

However, by default, the save function creates Version 7 MAT-files. Convert existing MAT-files to Version 7.3 by calling the save function with the `-v7.3` option, such as

```
load('durer.mat');  
save('mycopy_durer.mat', '-v7.3');
```

To change your preferences to save new files in Version 7.3 format, access the **Environment** section on the **Home** tab, and click  **Preferences**. Select **MATLAB > General > MAT-Files**.

Save Parts of Variables to MAT-Files

In this section...
“Save Using the matfile Function” on page 1-20
“Partial Saving Requires Version 7.3 MAT-Files” on page 1-22

Save Using the matfile Function

This example shows how to change and save part of a variable in a MAT-file. To run the code in this example, create a Version 7.3 MAT-file with two variables.

```
A = rand(5);  
B = ones(4,8);  
save example.mat A B -v7.3;  
clear A B
```

Update the values in the first row of variable B in `example.mat`.

```
example = matfile('example.mat','Writable',true)  
example.B(1,:) = 2 * example.B(1,:);
```

The `matfile` function creates a `matlab.io.MatFile` object that corresponds to a MAT-file:

```
matlab.io.MatFile
```

Properties:

```
Properties.Source: C:\Documents\MATLAB\example.mat  
Properties.Writable: true  
A: [5x5 double]  
B: [4x8 double]
```

When you index into objects associated with Version 7.3 MAT-files, MATLAB loads and saves only the part of the variable that you specify. This partial loading or saving requires less memory than `load` or `save` commands, which always operate on entire variables.

For very large files, the best practice is to read and write as much data into memory as possible at a time. Otherwise, repeated file access negatively impacts the performance of your code. For example, suppose your file contains many rows and columns, and loading a single row requires most of the available memory. Rather than updating one element at a time, update each row.

```
example = matfile('example.mat','Writable',true);

[nrowsB,ncolsB] = size(example,'B');
for row = 1:nrowsB
    example.B(row,:) = row * example.B(row,:);
end
```

If memory is not a concern, you can update the entire contents of a variable at a time, such as

```
example = matfile('example.mat','Writable',true);
example.B = 10 * example.B;
```

Alternatively, update a variable by calling the save function with the `-append` option. The `-append` option requests that the save function replace only the specified variable, `B`, and leave other variables in the file intact:

```
load('example.mat','B');
B(1,:) = 2 * B(1,:);
save('example.mat','-append','B');
```

This method always requires that you load and save the entire variable.

Use either method to add a variable to the file. For example, this code

```
example = matfile('example.mat','Writable',true);
example.C = magic(8);
```

performs the same save operation as

```
C = magic(8);
save('example.mat','-append','C');
clear C
```

Partial Saving Requires Version 7.3 MAT-Files

Any load or save operation that uses a `matlab.io.MatFile` object associated with a Version 7 or earlier MAT-file temporarily loads the entire variable into memory.


The `matfile` function creates files in Version 7.3 format. For example, this code

```
newfile = matfile('newfile.mat');
```

creates a MAT-file that supports partial loading and saving.

However, by default, the `save` function creates Version 7 MAT-files. Convert existing MAT-files to Version 7.3 by calling the `save` function with the `-v7.3` option, such as

```
load('durer.mat');  
save('mycopy_durer.mat', '-v7.3');
```

To change your preferences to save new files in Version 7.3 format, access the **Environment** section on the **Home** tab, and click  **Preferences**. Select **MATLAB > General > MAT-Files**.

Save Structure Fields as Separate Variables

If any of the variables in your current workspace are structure arrays, the default behavior of the `save` function is to store the entire structure. To store fields of a scalar structure as individual variables, use the `-struct` option to the `save` function.

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')
```

MAT-File Versions

In this section...
“Default Version” on page 1-24
“Overriding the Default MAT-File Version” on page 1-24
“Speeding Up Save and Load Operations” on page 1-25

Default Version


By default, all save operations except new file creation with the `matfile` function create Version 7 MAT-files. Override the default to:

- Allow access to the file using earlier versions of MATLAB.
- Take advantage of Version 7.3 MAT-file features: data items larger than 2 GB on 64-bit systems, and saving or loading parts of variables.

Note Version 7.3 MAT-files use an HDF5 based 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.

- 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 1-25.

Overriding the Default MAT-File Version

To identify or change the default version, access the **Environment** section on the **Home** tab, and click  **Preferences**. Select **MATLAB > General > MAT-Files**. Alternatively, specify a MAT-file version as an argument 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 following table shows the differences between previous and current MAT-file versions.

Value of version	Can Load in MATLAB Versions	Supported Features
'-v7.3'	7.3 (R2006b) or later	Version 7.0 features, plus support for data items greater than or equal to 2 GB on 64-bit systems.
'-v7'	7.0 (R14) or later	Version 6 features, plus data compression and Unicode® character encoding. Unicode encoding enables file sharing between systems that use different default character encoding schemes.
'-v6'	5 (R8) or later	Version 4 features, plus N -dimensional arrays, cell arrays and structures, and variable names greater than 19 characters.
'-v4'	all	Two-dimensional double, character, and sparse arrays.

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 7.3 MAT-files use an HDF5-based format that stores data in compressed chunks. The time required to load part of a variable from a Version 7.3 MAT-file depends on how that data is stored across one or more chunks. Each chunk that contains any portion of the data you want to load must be fully uncompressed to access the data. Rechunking your data can improve the performance of the load operation. To rechunk data, use the HDF5 command line tools, which are part of the HDF5 distribution.

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 1-24.

File Size Increases Unexpectedly When Growing Array

This example shows how to prevent an array from growing when writing one million double-precision values to a file, by assigning initial values to the array.

Construct a `matlab.io.MatFile` object for writing.

```
fileName = 'matFileOfDoubles.mat';  
matObj = matfile(fileName);  
matObj.Properties.Writable = true;
```

Define parameters of the values to write. In this case, write one million values, fifty thousand at a time. The values should have a mean of 123.4, and a standard deviation of 56.7.

```
size = 1000000;  
chunk = 50000;  
mean = 123.4;  
std = 56.7;
```

Assign an initial value of zero to the last element in the array prior to populating it with data.

```
matObj.data(1,size) = 0;
```

View the size of the file.

- On Windows systems, use `dir`.

```
system('dir matFileOfDoubles.mat');
```

- On UNIX[®] systems, use `ls -ls`:

```
system('ls -ls matFileOfDoubles.mat');
```

`matFileOfDoubles.mat` is less than 5000 bytes. Assigning an initial value to the last element of the array does not create a large file.

Write data to the array, one chunk at a time.

```
nout = 0;
```

```
while(nout < size)
    fprintf('Writing %d of %d\n',nout,size);
    chunkSize = min(chunk,size-nout);
    data = mean + std * randn(1,chunkSize);
    matObj.data(1,(nout+1):(nout+chunkSize)) = data;
    nout = nout + chunkSize;
end
```

View the size of the file.

```
system('dir matFileOfDoubles.mat');
```

The file size is now larger now because the array is populated with data.

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`, and `path`.

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 MATLAB `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 the `load` function.
- Initialize variables (e.g., assign to an empty matrix or empty string) within the function before calling `load`.

To determine whether a particular name is associated with a MATLAB function, use the `exist` function.

Create Temporary Files

Use the `tempdir` function to return the name of the folder designated to hold temporary files on your system. For example, issuing `tempdir` on The Open Group UNIX systems returns the `/tmp` folder.

Use the `tempname` function to return a file name in the temporary folder. The returned file name 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:

```
fileID = fopen(tempname, 'w');
```

In most cases, `tempname` generates a universally unique identifier (UUID). However, if you run MATLAB without JVM™, then `tempname` generates a random string using the CPU counter and time, and this string is not guaranteed to be unique.

Some systems delete temporary files every time you reboot the system. On other systems, designating a file as temporary means only that the file is not backed up.

Text Files

- “Ways to Import Text Files” on page 2-2
- “Select Text File Data Using Import Tool” on page 2-4
- “Import Dates and Times from Text Files” on page 2-9
- “Import Numeric Data from Text Files” on page 2-10
- “Import Mixed Text and Numeric Data from Text Files” on page 2-13
- “Import Large Text File Data in Blocks” on page 2-16
- “Import Data from a Nonrectangular Text File” on page 2-24
- “Write to Delimited Data Files” on page 2-26
- “Write to a Diary File” on page 2-32

Ways to Import Text Files


You can import text files into MATLAB both interactively and programmatically.

To import data interactively, use the Import Tool. You can generate code to repeat the operation on multiple similar files. The Import Tool supports text files, including those with the extensions `.txt`, `.dat`, `.csv`, `.asc`, `.tab`, and `.dlm`. These text files can be nonrectangular, and can have row and column headers, as shown in the following figure. Data in these files can be a combination of numeric and nonnumeric text, and can be delimited by one or more characters.

To import data from text files programmatically, use an import function. Most of the import functions for text files require that each row of data has the same number of columns, and they allow you to specify a range of data to import.

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

This table compares the primary import options for text files.

Import Option	Description	For More Information, See...
Import Tool 	Import a file or range of data to column vectors, a matrix, a cell array, or a table. You can generate code to	“Select Text File Data Using Import Tool” on page 2-4

Import Option	Description	For More Information, See...
	repeat the operation on multiple similar files.	
<code>readtable</code>	Import column-oriented data into a table.	“Import Mixed Text and Numeric Data from Text Files” on page 2-13
<code>csvread</code>	Import a file or range of comma-separated numeric data to a matrix.	“Import Comma-Separated Data” on page 2-10
<code>dlmread</code>	Import a file or a range of numeric data separated by any single delimiter to a matrix.	“Import Delimited Numeric Data” on page 2-11
<code>textscan</code>	Import a nonrectangular or arbitrarily formatted text file to a cell array.	“Import Data from a Nonrectangular Text File” on page 2-24

For information on importing files with more complex formats, see “Import Text Data Files with Low-Level I/O” on page 4-2.

Select Text File Data Using Import Tool

In this section...


“Select Data Interactively” on page 2-4

“Import Data from Multiple Text Files” on page 2-7

Select Data Interactively

This example shows how to import data from a text file with column headers and numeric data using the Import Tool. The file in this example, `grades.txt`, contains the following data (to create the file, use any text editor, and copy and paste):

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

On the **Home** tab, in the **Variable** section, click **Import Data** . Alternatively, right-click the name of the file in the Current Folder browser and select **Import Data**. The Import Tool opens.

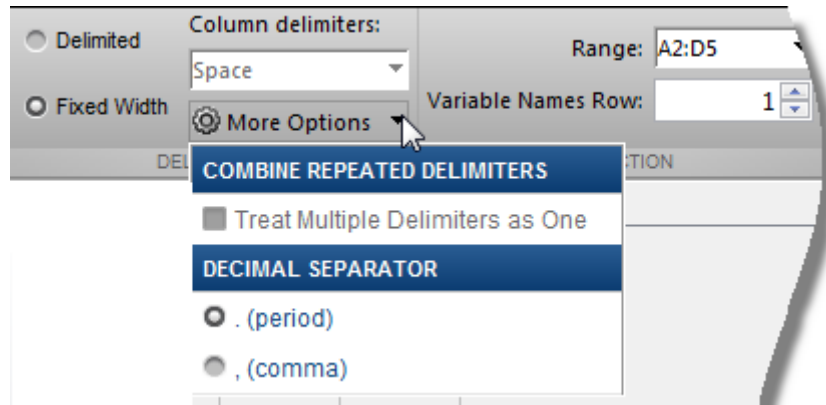
The screenshot shows the Import Tool interface for a file named 'grades.txt'. The 'Imported Data' section is active, showing four options: 'Column vectors', 'Matrix', 'Cell Array', and 'Table'. The 'Table' option is selected. The main window displays a table with columns A, B, C, and D, and rows 1 through 5. The data is imported as a table with headers 'John', 'Ann', 'Mark', and 'Rob'.

	A	B	C	D
	John	Ann	Mark	Rob
	NUM...	NUMBER	NUMBER	NUMBER
1	John	Ann	Mark	Rob
2	88.4	91.5	89.2	77.3
3	83.2	88.0	67.8	91.0
4	77.8	76.3		92.5
5	92.1	96.4	81.2	84.6

The Import Tool recognizes that `grades.txt` is a fixed width file. In the **Imported Data** section, select how you want the data to be imported. The following table indicates how data is imported depending on the option you select.

Option Selected	How Data is Imported
Column vectors	Import each column of the selected data as an individual m -by-1 vector.
Matrix	Import selected data as an m -by- n numeric array.
Cell Array	Import selected data as a cell array that can contain multiple data types, such as numeric data and text.
Table	Import selected data as a table.

Under **More Options**, you can specify whether the Import Tool should use a period or a comma as the decimal separator for numeric values.

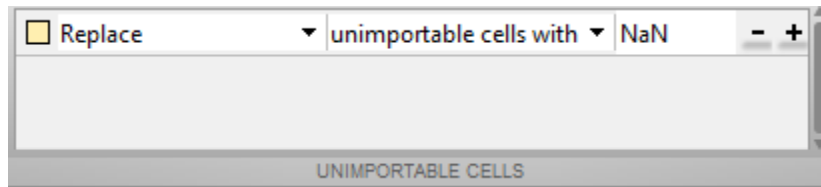


Double-click on a variable name to rename it.

	A	B	C	D
	John	Ann	Mark	Rob
	N...	NUMBER	NUMBER	NUMBER
1	John	Ann	Mark	Rob
2	88.4	91.5	89.2	77.3
3	83.2	88.0	67.8	91.0
4	77.8	76.3		92.5
5	92.1	96.4	81.2	84.6

You also can use the **Variable Names Row** box in the **Selection** section to select the row in the text file that the Import Tool uses for variable names.


The Import Tool highlights unimportable cells. Unimportable cells are cells that contain data that cannot be imported in the format specified for that column. In this example, the cell at row 3, column C, is considered unimportable because a blank cell is not numeric. Highlight colors correspond to proposed rules to make the data fit into a numeric array. You can add, remove, reorder, or edit rules, such as changing the replacement value from NaN to another value.



All rules apply to the imported data only, and do not change the data in the file. You must specify rules any time the range includes nonnumeric data and you are importing into a matrix or numeric column vectors.

You can see how your data will be imported when you place the cursor over individual cells.

	A	B	C	D
	John	Ann	Mark	Rob
	N...	NUMBER	NUMBER	NUMBER
1	John	Ann	Mark	Rob
2	88.4	91.5	89.2	77.3
3	83.2	88.0		
4	77.8	76.3	NaN	92.5
5	92.1	96.4	81.2	84.6

When you click the **Import Selection** button , the Import Tool creates variables in your workspace.

For more information on interacting with the Import Tool, watch this video.

Import Data from Multiple Text Files

This example shows how to perform the same import operation on multiple files using the Import Tool. You can generate code from the Import Tool, making it easier to repeat the operation. The Import Tool generates a program script that you can edit and run to import the files, or a function that you can call for each file.

Suppose you have a set of text files in the current folder named `myfile01.txt` through `myfile25.txt`, and you want to import the data from each file, starting from the second row. Generate code to import the entire set of files as follows:

- 1 Open one of the files in the Import Tool.
- 2 Click **Import Selection** ▾, and then select **Generate Function**. The Import Tool generates code similar to the following excerpt, and opens the code in the Editor.

```
function data = importfile(filename,startRow,endRow)
%IMPORTFILE Import numeric data from a text file as a matrix.
...
```

- 3 Save the function.
- 4 In a separate program file or at the command line, create a `for` loop to import data from each text file into a cell array named `myData`:

```
numFiles = 25;
startRow = 2;
endRow = inf;
myData = cell(1,numFiles);

for fileNum = 1:numFiles
    fileName = sprintf('myfile%02d.txt',fileNum);
    myData{fileNum} = importfile(fileName,startRow,endRow);
end
```

Each cell in `myData` contains an array of data from the corresponding text file. For example, `myData{1}` contains the data from the first file, `myfile01.txt`.

Import Dates and Times from Text Files

Formatted dates and times (such as '01/01/01' or '12:30:45') are *not* numeric fields. MATLAB interprets dates and times in files as text strings.

You can use the Import Tool to import formatted dates and times as serial date numbers. Specify the formats of dates and times, using the drop-down menu for each column. You can select from a predefined date format, or enter a custom format.

D	E	F	G	H
ReportDate	Rev1	Sales	Rev2	Total
DATE/TIME	TEXT	TEXT	TEXT	TEXT
TEXT (string)			TotalRevenue	TotalSales
TEXT			0.858,00	525,39
Text like "1.234" will convert to string '1.234'			2.623,22	57.736,18
NUMBER (double)			2.614,39	282,08
NUMBER			2.671,39	62.753,00
Text like "1.234" will convert to number 1.234			2.266,48	259,68
DATE/TIME (serial date number)			1.691,96	57.643,06
dd-mmm-yyyy			1.723,53	193,15
more date formats ...			2.008,51	50.290,65
2.262,12			160,39	
1-Oct-09	€901,89	9.924,71	€2.445,34	42.784,04
1-Nov-09	€1.221,79	11.562,98	€2.952,84	216,51
1-Dec-09	13.474,30	€3.241,38	47.660,53	
€1.555,78			266,00	

Import Numeric Data from Text Files

In this section...

“Import Comma-Separated Data” on page 2-10

“Import Delimited Numeric Data” on page 2-11

Import Comma-Separated Data

This example shows how to import comma-separated numeric data from a text file, using the `csvread` function.

Create a sample file named `ph.dat` that contains the following comma-separated data:

```
85.5, 54.0, 74.7, 34.2
63.0, 75.6, 46.8, 80.1
85.5, 39.6, 2.7, 38.7
```

```
A = 0.9*gallery('integerdata',99,[3,4],1);
dlmwrite('ph.dat',A,',');
```

The sample file, `ph.dat`, resides in your current folder.

Read the entire file using `csvread`. The file name is the only required input argument to the `csvread` function.

```
M = csvread('ph.dat')
```

```
M =
```

```
85.5000    54.0000    74.7000    34.2000
63.0000    75.6000    46.8000    80.1000
85.5000    39.6000     2.7000    38.7000
```

`M` is a 3-by-4 double array containing the data from the file.

Import only the rectangular portion of data starting from the first row and third column in the file. When using `csvread`, row and column indices are zero-based.

```
N = csvread('ph.dat',0,2)
```

```
N =
```

```
74.7000    34.2000
46.8000    80.1000
 2.7000    38.7000
```

Import Delimited Numeric Data

This example shows how to import numeric data delimited by any single character using the `dlmread` function.

Create a tab-delimited file named `num.txt` that contains the following data:

```
95    89    82    92
23    76    45    74
61    46    61    18
49     2    79    41
```

```
A = gallery('integerdata',99,[4,4],0);
dlmwrite('num.txt',A,'\t');
```

The sample file, `num.txt`, resides in your current folder.

Read the entire file. The file name is the only required input argument to the `dlmread` function. `dlmread` determines the delimiter from the formatting of the file.

```
M = dlmread('num.txt')
```

```
M =
```

```
95    89    82    92
```

```
23    76    45    74
61    46    61    18
49     2    79    41
```

M is a 4-by-4 double array containing the data from the file.

Read only the rectangular block of data beginning from the second row, third column, in the file. When using `d1mread`, row and column indices are zero-based. When you specify a specific range to read, you must also specify the delimiter. Use `'\t'` to indicate a tab delimiter.

```
N = d1mread('num.txt','\t',1,2)
```

```
N =
```

```
45    74
61    18
79    41
```

`d1mread` returns a 3-by-2 double array.

Read only the first two columns. You can use spreadsheet notation to indicate the range, in this case, `'A1..B4'`.

```
P = d1mread('num.txt','\t','A1..B4')
```

```
P =
```

```
95    89
23    76
61    46
49     2
```

See Also

`csvread` | `d1mread`

Import Mixed Text and Numeric Data from Text Files

In this section...

“Read File with Column Names” on page 2-13

“Read File Without Column Names” on page 2-14

Read File with Column Names

This example shows how to use the `readtable` function to import a text file with column and row headings.

Create a tab-delimited text file named `grades.txt` that contains the following (copy and paste into a text editor):

```
Class Grades for Spring Term
  Grade1 Grade2 Grade3
John 85 90 95
Ann 90 92 98
Martin 100 95 97
Rob 77 86 93
```

This file contains one header line followed by a row of column names, `Grade1`, `Grade2`, and `Grade3`.

Call `readtable` to read the file. Use the name-value pair argument, `HeaderLines`, to specify one header line to ignore. Use the name-value pair argument, `Delimiter`, to specify a tab delimiter.

```
T = readtable('grades.txt','HeaderLines',1,'Delimiter','\t')
```

T =

<u>Var1</u>	<u>Grade1</u>	<u>Grade2</u>	<u>Grade3</u>
'John'	85	90	95
'Ann'	90	92	98

```
'Martin' 100 95 97
'Rob'    77  86 93
```

`readtable` returns a 4-by-4 table. By default, `readtable` reads variable names from the first row of the file following the header lines. Because the first column of data in the file does not have text in the first row, `readtable` assigns that variable the name, `Var1`.

Read the file again, this time reading the first column of data as row names.

```
T = readtable('grades.txt','HeaderLines',1,...
'Delimiter','\t','ReadRowNames',true)
```

T =

	Grade1	Grade2	Grade3
John	85	90	95
Ann	90	92	98
Martin	100	95	97
Rob	77	86	93

`readtable` returns a 4-by-3 table with row names and variable names.

Read File Without Column Names

This example shows how to use the `readtable` function to import a text file with no column headings.

Create a text file named `results.dat` that contains the following (copy and paste into a text editor):

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

This file contains data without any column names.

Call `readtable` to read the file. By default, `readtable` treats the first row in a file as variable names. Use the name-value pair argument, `ReadVariableNames`, to tell `readtable` not to treat the first row in the file as variable names.

```
T = readtable('results.dat','Delimiter',' ','ReadVariableNames',false)
```

```
T =
```

Var1	Var2	Var3	Var4	Var5
'Sally'	'09/12/2005'	12.34	45	'Yes'
'Larry'	'10/12/2005'	34.56	54	'Yes'
'Tommy'	'11/12/2005'	67.89	23	'No'

`readtable` returns a 3-by-5 table and assigns default names to the table variables.

View the values in the table variable, `Var4`, using dot notation.

```
T.Var4
```

```
ans =
```

```
45
54
23
```

See Also

`readtable`

Concepts

- “Access Data in a Table”

Import Large Text File Data in Blocks

This example shows how to read small blocks of data from an arbitrarily large delimited text file using the `textscan` function and avoid memory errors. The first part of the example shows how to specify a constant block size. The second part of the example shows how to read and process each block of data in a loop.

Specify Block Size

Specify a constant block size, and then process each block of data within a loop.

Copy and paste the following text into a text editor to create a tab-delimited text file, `bigfile.txt`, in your current folder.

```
## A ID = 02476
## YKZ Timestamp Temp Humidity Wind Weather
06-Sep-2013 01:00:00 6.6 89 4 clear
06-Sep-2013 05:00:00 5.9 95 1 clear
06-Sep-2013 09:00:00 15.6 51 5 mainly clear
06-Sep-2013 13:00:00 19.6 37 10 mainly clear
06-Sep-2013 17:00:00 22.4 41 9 mostly cloudy
06-Sep-2013 21:00:00 17.3 67 7 mainly clear
## B ID = 02477
## YVR Timestamp Temp Humidity Wind Weather
09-Sep-2013 01:00:00 15.2 91 8 clear
09-Sep-2013 05:00:00 19.1 94 7 n/a
09-Sep-2013 09:00:00 18.5 94 4 fog
09-Sep-2013 13:00:00 20.1 81 15 mainly clear
09-Sep-2013 17:00:00 20.1 77 17 n/a
09-Sep-2013 18:00:00 20.0 75 17 n/a
09-Sep-2013 21:00:00 16.8 90 25 mainly clear
## C ID = 02478
## YYZ Timestamp Temp Humidity Wind Weather
```

This file has commented lines beginning with `##`, throughout the file. The data is arranged in five columns: The first column contains strings indicating timestamps. The second, third, and fourth columns contain numeric data indicating temperature, humidity and wind speed. The last column contains descriptive strings.

Define the size of each block to read from the text file. You do not need to know the total number of blocks in advance, and the number of rows of data in the file do not have to divide evenly into the block size.

Specify a block size of 5.

```
N = 5;
```

Open the file to read using the `fopen` function.

```
fileID = fopen('bigfile.txt');
```

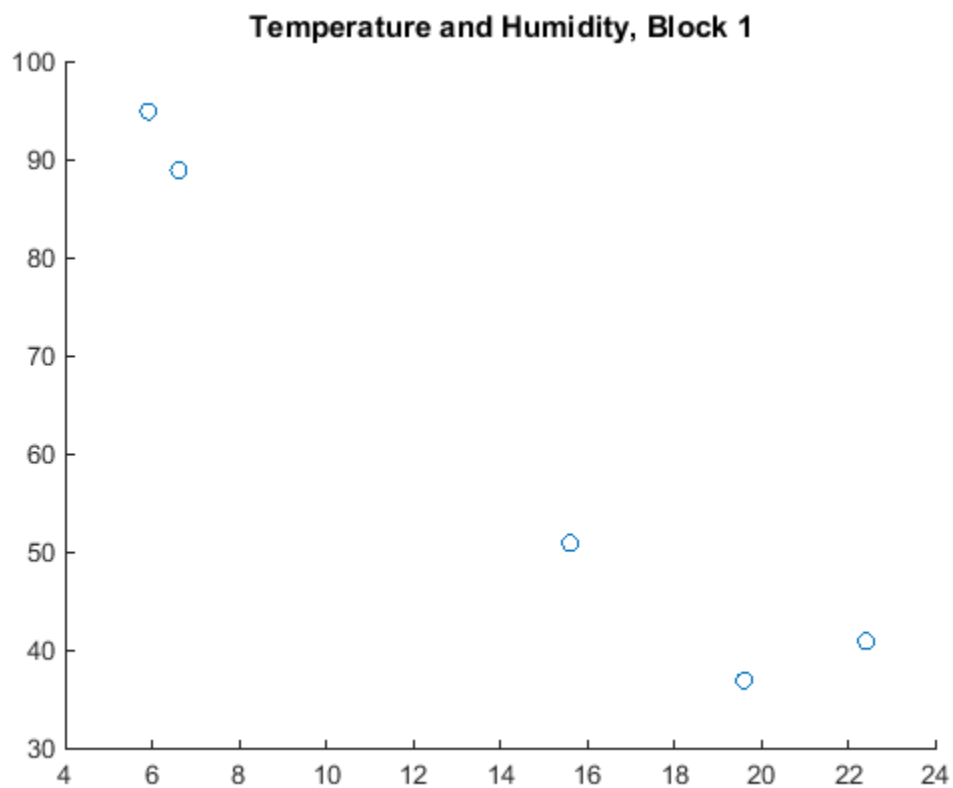
`fopen` returns a file identifier, `fileID`, that the `textscan` function calls to read from the file. `fopen` positions a pointer at the beginning of the file, and each read operation changes the location of that pointer.

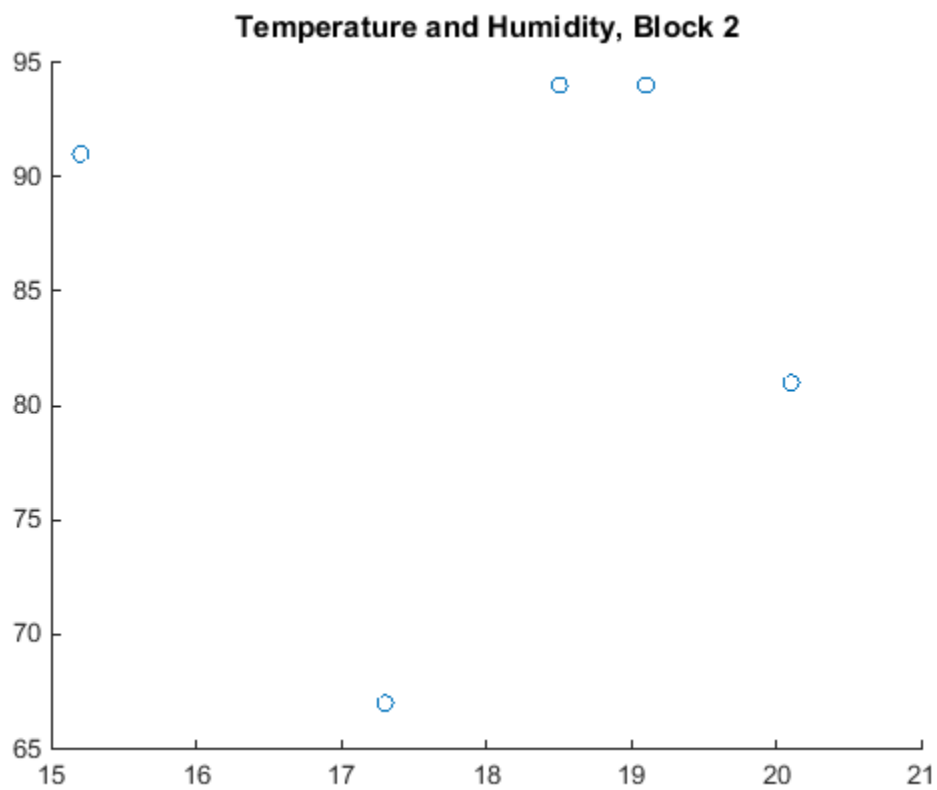
Describe each data field using format specifiers, such as `'%s'` for a string, `'%d'` for an integer, or `'%f'` for a floating-point number.

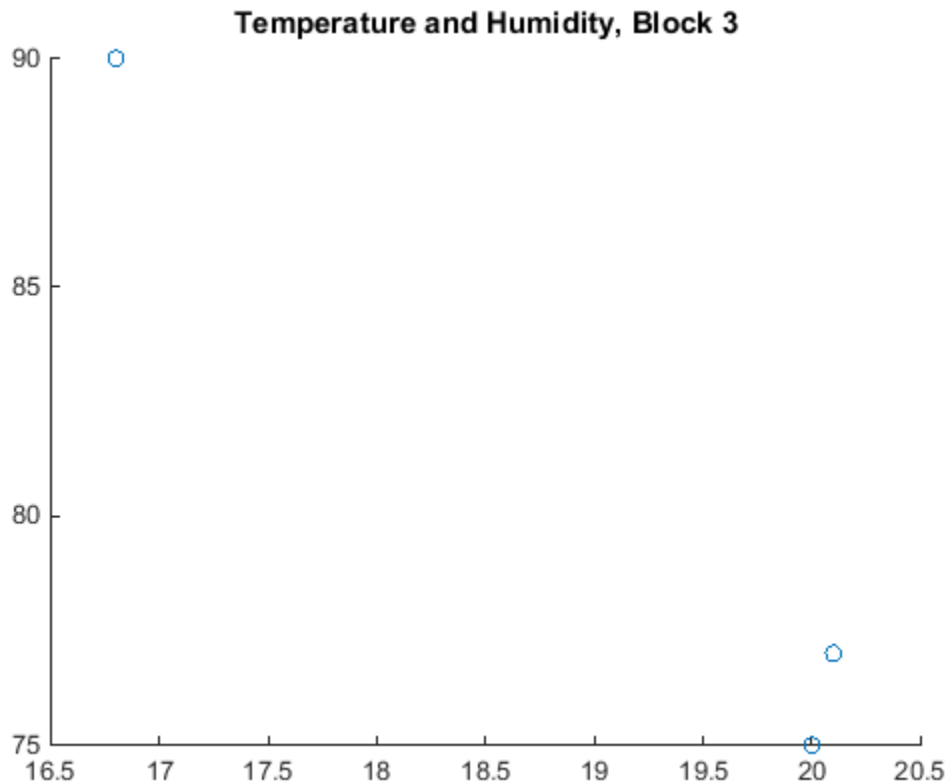
```
formatSpec = '%s %f %f %f %s';
```

In a `while` loop, call `textscan` to read each block of data. The file identifier, format specifier string, and the segment size (`N`), are the first three inputs to `textscan`. Ignore the commented lines using the `CommentStyle` name-value pair argument. Specify the tab delimiter using the `Delimiter` name-value pair argument. Then, process the data in the block. In this example, call `scatter` to display a scatter plot of temperature and humidity values in the block. The commands within the loop execute while the file pointer is not at the end of the file.

```
k = 0;
while ~feof(fileID)
    k = k+1;
    C = textscan(fileID,formatSpec,N,'CommentStyle','##','Delimiter','\t');
    figure, scatter(C{2},C{3}), title(['Temperature and Humidity, Block ',num2str(k)]);
end
```







`textscan` reads data from `bigfile.txt` indefinitely, until it reaches the end of the file or until it cannot read a block of data in the format specified by `formatSpec`. For each complete block, `textscan` returns a 1-by-5 cell array. Because the sample file, `bigfile.txt`, contains 13 rows of data, `textscan` returns only 3 rows in the last block.

View the temperature values in the last block returned by `textscan`.

```
C{2}
```

```
ans =
```

```
20.1000
```

```

20.0000
16.8000

```

Close the file.

```
fclose(fileID);
```

Read Data with Arbitrary Block Sizes

Read and process separately each block of data between commented lines in the file, `bigfile.txt`. The length of each block can be arbitrary. However, you must specify the number of lines to skip between blocks of data. In `bigfile.txt`, each block of data is preceded by two lines of comments.

Open the file for reading.

```
fileID = fopen('bigfile.txt');
```

Specify the format of the data you want to read. Tell `textscan` to ignore certain data fields by including `.*` in the format specifier string, `formatSpec`. In this example, skip the third and fourth columns of floating-point data using `.*f`.

```
formatSpec = '%s %f .*f .*f %s';
```

Read a block of data in the file. Use the `HeaderLines` name-value pair argument to instruct `textscan` to skip two lines before reading data.

```
D = textscan(fileID,formatSpec,'HeaderLines',2,'Delimiter','\t')
```

```
D =
```

```

    {7x1 cell}    [6x1 double]    {6x1 cell}

```

`textscan` returns a 1-by-3 cell array, `D`.

View the contents of the first cell in `D`.

```
D{1,1}
```

```
ans =
```

```
'06-Sep-2013 01:00:00'  
'06-Sep-2013 05:00:00'  
'06-Sep-2013 09:00:00'  
'06-Sep-2013 13:00:00'  
'06-Sep-2013 17:00:00'  
'06-Sep-2013 21:00:00'  
'## B'
```

`textscan` stops reading after the text, '## B', because it cannot read the subsequent text as a number, as specified by `formatSpec`. The file pointer remains at the position where `textscan` terminated.

Process the first block of data. In this example, find the maximum temperature value in the second cell of `D`.

```
maxTemp1 = max(D{1,2})
```

```
maxTemp1 =
```

```
22.4000
```

Repeat the call to `textscan` to read the next block of data.

```
D = textscan(fileID,formatSpec,'HeaderLines',2,'Delimiter','\t')
```

```
D =
```

```
{8x1 cell} [7x1 double] {7x1 cell}
```

Again, `textscan` returns a 1-by-3 cell array.

Find the maximum temperature value in this block of data.

```
maxTemp2 = max(D{1,2})
```

```
maxTemp2 =
```

```
20.1000
```

Close the file.

```
fclose(fileID);
```

See Also

textscan | fopen

Concepts

- “Access Data in a Cell Array”
- “Moving within a File” on page 4-15

Import Data from a Nonrectangular Text File

This example shows how to import data from a nonrectangular file using the `textscan` function. When using `textscan`, your data does not have to be in a regular pattern of columns and rows, but it must be in a repeated pattern.

Create a file named `nonrect.dat` that contains the following (copy and paste into a text editor):

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

Open the file to read using the `fopen` function.

```
fileID = fopen('nonrect.dat');
```

`fopen` returns a file identifier, `fileID`, that `textscan` calls to read from the file.

Describe the pattern of the file 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. 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.

```
formatSpec = '%*s v1=%f v2=%f v3=%f %*s';
```

Import the data using `textscan`. Pass the file identifier and `formatSpec` as inputs. Since each data 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` name-value pair argument to `true`.

```
C = textscan(fileID,formatSpec,...  
            'Delimiter', '\n', ...  
            'CollectOutput', true)
```

```
C =
```

```
    [2x3 double]
```

textscan returns the cell array, C.

Close the file.

```
fclose(fileID);
```

View the contents of C.

```
celldisp(C)
```

```
C{1} =
```

```
    12.6700    3.1400    6.7780  
    21.7800    5.2400    9.8380
```

See Also

textscan

Concepts

- “Access Data in a Cell Array”

Write to Delimited Data Files

In this section...

“Export Numeric Array to ASCII File” on page 2-26

“Export Table to Text File” on page 2-28

“Export Cell Array to Text File” on page 2-29

Export Numeric Array to ASCII File

- “Export Numeric Array to ASCII File Using `save`” on page 2-26
- “Export Numeric Array to ASCII File Using `dlmwrite`” on page 2-27

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, you can first convert the cell array to a matrix using the `cell2mat` function, and 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 Numeric Array to 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.

Export Numeric Array to 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, `dmlwrite` 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 ('').

Export Table to Text File

This example shows how to export a table to a text file, using the `writetable` function.

Create a sample table, `T`, for exporting.

```
Name = {'M4'; 'M5'; 'M6'; 'M8'; 'M10'};  
Pitch = [0.7; 0.8; 1; 1.25; 1.5];  
Shape = {'Pan'; 'Round'; 'Button'; 'Pan'; 'Round'};  
Price = [10.0; 13.59; 10.50; 12.00; 16.69];  
Stock = [376; 502; 465; 1091; 562];  
T = table(Pitch, Shape, Price, Stock, 'RowNames', Name)
```

`T =`

	Pitch	Shape	Price	Stock
	-----	-----	-----	-----
M4	0.7	'Pan'	10	376
M5	0.8	'Round'	13.59	502
M6	1	'Button'	10.5	465
M8	1.25	'Pan'	12	1091
M10	1.5	'Round'	16.69	562

The table has both column headings and row names.

Export the table, `T`, to a text file named `tabledata.txt`.

```
writetable(T, 'tabledata.txt')
```

View the file.

```
type tabledata.txt
```

```
Pitch, Shape, Price, Stock  
0.7, Pan, 10, 376  
0.8, Round, 13.59, 502
```

```
1,Button,10.5,465
1.25,Pan,12,1091
1.5,Round,16.69,562
```

By default, `writetable` writes comma-separated data, includes table variable names as column headings, and does not write row names.

Export table `T` to a tab-delimited text file named `tabledata2.txt` and write the row names in the first column of the output. Use the `Delimiter` name-value pair argument to specify a tab delimiter, and the `WriteRowNames` name-value pair argument to include row names.

```
writetable(T,'tabledata2.txt','Delimiter','\t','WriteRowNames',true)
```

View the file.

```
type tabledata2.txt
```

```
Row Pitch Shape Price Stock
M4 0.7 Pan 10 376
M5 0.8 Round 13.59 502
M6 1 Button 10.5 465
M8 1.25 Pan 12 1091
M10 1.5 Round 16.69 562
```

Export Cell Array to Text File

Export Cell Array Using `fprintf`

This example shows how to export a cell array to a text file, using the `fprintf` function.

Create a sample cell array, `C`, for exporting.

```
C = {'Atkins',32,77.3,'M';'Cheng',30,99.8,'F';'Lam',31,80.2,'M'}
```

```
C =
```

```
    'Atkins'    [32]    [77.3000]    'M'
```

```
'Cheng'    [30]    [99.8000]    'F'  
'Lam'      [31]    [80.2000]    'M'
```

Open a file named `celldata.dat` for writing.

```
fileID = fopen('celldata.dat','w');
```

`fopen` returns a file identifier, `fileID`, that `fprintf` calls to write to the file.

Describe the pattern of the file data using format specifiers. Typical format specifiers include `'%s'` for a string, `'%d'` for an integer, or `'%f'` for a floating-point number. Separate each format specifier with a space to indicate a space delimiter for the output file. Include a newline character at the end of each row of data (`'\n'`).

```
formatSpec = '%s %d %2.1f %s\n';
```

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.

Determine the size of `C`. Then, export one row of data at a time using the `fprintf` function.

```
[nrows,ncols] = size(C);  
for row = 1:nrows  
    fprintf(fileID,formatSpec,C{row,:});  
end
```

`fprintf` writes a space-delimited file.

Close the file.

```
fclose(fileID);
```

View the file.

```
type celldata.dat
```

```
Atkins 32 77.3 M
Cheng 30 99.8 F
Lam 31 80.2 M
```

Convert Cell Array to Table for Export

This example shows how to convert a cell array of mixed text and numeric data to a table before writing the data to a text file. Tables are suitable for column-oriented or tabular data. You then can write the table to a text file using the `writetable` function.

Convert the cell array, `C`, from the previous example, to a table using the `cell2table` function. Add variable names to each column of data using the `VariableNames` name-value pair argument.

```
T = cell2table(C,'VariableNames',{'Name','Age','Result','Gender'});
```

Write table `T` to a text file.

```
writetable(T,'tabledata.dat');
```

View the file.

```
type tabledata.dat
```

```
Name, Age, Result, Gender
Atkins, 32, 77.3, M
Cheng, 30, 99.8, F
Lam, 31, 80.2, M
```

See Also

`fprintf` | `type` | `writetable` | `save` | `dlmwrite`

Write 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.

Spreadsheets

- “Ways to Import Spreadsheets” on page 3-2
- “Select Spreadsheet Data Using Import Tool” on page 3-4
- “Import a Worksheet or Range” on page 3-8
- “Import All Worksheets from a File” on page 3-12
- “System Requirements for Importing Spreadsheets” on page 3-15
- “When to Convert Dates from Excel Files” on page 3-16
- “Export to Excel Spreadsheets” on page 3-19


Ways to Import Spreadsheets

In this section...
“Import Data from Spreadsheets” on page 3-2
“Paste Data from Clipboard” on page 3-3

Import Data from Spreadsheets

You can import data from spreadsheet files into MATLAB interactively, using the Import Tool, or programmatically, using an import function.


This table compares the primary import options for spreadsheet files.

Import Option	Description	For More Information, See...
Import Tool 	Import a worksheet or range to column vectors, a matrix, a cell array, or a table. You can generate code to repeat the operation on multiple similar files.	“Select Spreadsheet Data Using Import Tool” on page 3-4
readtable	Import a worksheet or range to a table.	“Import a Worksheet or Range” on page 3-8
xlsread	Import a worksheet or range to numeric and cell arrays.	
importdata	Import one or more worksheets in a file to a structure array.	“Import All Worksheets from a File” on page 3-12

Some import options require that your system includes Excel for Windows. For more information, see “System Requirements for Importing Spreadsheets” on page 3-15.

Paste Data from Clipboard

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

- On the Workspace browser title bar, click , and then select **Paste**.
- Open an existing variable in the Variables editor, right-click, and then select **Paste Excel Data**.
- Call `uiimport -pastespecial`.

Select Spreadsheet Data Using Import Tool

In this section...

“Select Data Interactively” on page 3-4

“Import Data from Multiple Spreadsheets” on page 3-6

Select Data Interactively

This example shows how to import data from a spreadsheet into the workspace with the Import Tool. The worksheet in this example includes three columns of data labeled Station, Temp, and Date:

Station	Temp	Date
12	98	9/22/2010
13	x	10/23/2010
14	97	12/1/2010

On the **Home** tab, in the **Variable** section, click **Import Data** .

Alternatively, in the Current Folder browser, double-click the name of a file with an extension of .xls, .xlsx, .xlsb, or .xlsm. The Import Tool opens.

Select the data you want to import. In the **Imported Data** section, select how you want the data to be imported. The following table indicates how data is imported depending on the option you select.

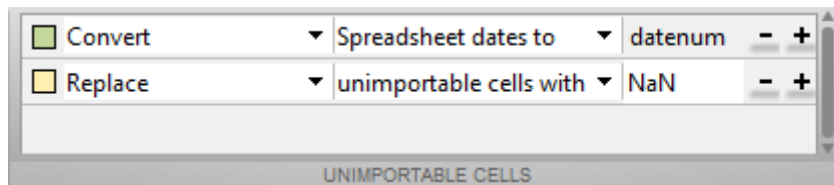
Option Selected	How Data is Imported
Column vectors	Import each column of the selected data as an individual m-by-1 vector.
Matrix	Import selected data as an m-by-n numeric array.
Cell Array	Import selected data as a cell array that can contain multiple data types, such as numeric data and text.
Table	Import selected data as a table.

For example, the data in the following figure corresponds to a 3-by-3 matrix named `untitled`. You can edit the variable name within the tab, and you can select noncontiguous sections of data for the same variable.

	A	B	C
	untitled		
1	Station	Temp	Date
2	12	98	9/22/2013
3	13	x	10/2
4	14	97	735569


Note: A tooltip is visible over the cell containing '735569', displaying: 12/1/2013 Converted To[Type:Number, Value:735569.0]

If you choose to import as a matrix or numeric column vectors, the tool highlights any nonnumeric data in the worksheet. Each highlight color corresponds to a proposed rule to make the data fit into a numeric array. For example, you can convert spreadsheet dates to MATLAB serial date numbers. You can see how your data will be imported when you place the cursor over individual cells.



You can add, remove, reorder, or edit rules, such as changing the replacement value from `NaN` to another value. All rules apply to the imported data only, and do not change the data in the file. You must specify rules any time the range includes nonnumeric data and you are importing into a matrix or numeric column vectors.

Any cells that contain `#Error?` correspond to formula errors in your spreadsheet file, such as division by zero. The Import Tool regards these cells as nonnumeric.

When you click the **Import Selection** button , the Import Tool creates variables in your workspace.

For more information on interacting with the Import Tool, watch this video.

Import Data from Multiple Spreadsheets

If you plan to perform the same import operation on multiple files, you can generate code from the Import Tool to make it easier to repeat the operation. On all platforms, the Import Tool can generate a program script that you can edit and run to import the files. On Microsoft Windows systems with Excel software, the Import Tool can generate a function that you can call for each file.

For example, suppose you have a set of spreadsheets in the current folder named `myfile01.xlsx` through `myfile25.xlsx`, and you want to import the same range of data, `A2:G100`, from the first worksheet in each file. Generate code to import the entire set of files as follows:

- 1 Open one of the files in the Import Tool.
- 2 From the **Import** button, select **Generate Function**. The Import Tool generates code similar to the following excerpt, and opens the code in the Editor.

```
function data = importfile(workbookFile, sheetName, range)
%IMPORTFILE    Import numeric data from a spreadsheet
...
```

- 3 Save the function.
- 4 In a separate program file or at the command line, create a `for` loop to import data from each spreadsheet into a cell array named `myData`:

```
numFiles = 25;
range = 'A2:G100';
sheet = 1;
myData = cell(1,numFiles);

for fileNum = 1:numFiles
    fileName = sprintf('myfile%02d.xlsx',fileNum);
    myData{fileNum} = importfile(fileName,sheet,range);
end
```

Each cell in `myData` contains an array of data from the corresponding worksheet. For example, `myData{1}` contains the data from the first file, `myfile01.xlsx`.

Import a Worksheet or Range

In this section...

“Read Column-Oriented Data into Table” on page 3-8

“Read Numeric and Text Data into Arrays” on page 3-9

“Get Information about a Spreadsheet” on page 3-11

Read Column-Oriented Data into Table

This example shows how to import mixed numeric and text data from a spreadsheet into a table, using the `readtable` function. Tables are suitable for column-oriented or tabular data. You can store variable names or row names along with the data in a single container.

This example uses a sample spreadsheet file, `climate.xlsx`, that contains the following numeric and text data in a worksheet called `Temperatures`.

```
Time Temp Visibility
12 98 clear
13 99 clear
14 97 partly cloudy
```

Create the sample file for reading.

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

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

`xlswrite` warns that it has added a worksheet.

Call `readtable` to read all the data in the worksheet called `Temperatures`. Specify the worksheet name using the `Sheet` name-value pair argument. If your data is on the first worksheet in the file, you do not need to specify `Sheet`.

```
T = readtable('climate.xlsx','Sheet','Temperatures')
```

T =

Time	Temp	Visibility
12	98	'clear'
13	99	'clear'
14	97	'partly cloudy'

`readtable` returns a 3-by-3 table. By default, `readtable` reads the first row of the worksheet as variable names for the table.

Read only the first two columns of data by specifying a range, 'A1:B4'.

```
cols = readtable('climate.xlsx','Sheet','Temperatures','Range','A1:B4')
```

cols =

Time	Temp
12	98
13	99
14	97

`readtable` returns a 3-by-2 table.

Read Numeric and Text Data into Arrays

This example shows how to import mixed numeric and text data into separate arrays in MATLAB, using the `xlsread` function.

This example uses a sample spreadsheet file, `climate.xlsx`, that contains the following data in a worksheet called `Temperatures`.

Time	Temp
12	98
13	99
14	97

Create the sample file for reading.

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

```
xlswrite('climate2.xlsx',d,'Temperatures');
```

xlswrite warns that it has added a worksheet.

Import only the numeric data into a matrix, using `xlsread` with a single output argument. `xlsread` ignores any leading row or column of text in the numeric result.

```
num = xlsread('climate2.xlsx','Temperatures')
```

```
num =  
     12     98  
     13     99  
     14     97
```

`xlsread` returns the numeric array, `num`.

Alternatively, import both numeric data and text data, by specifying two output arguments in the call to `xlsread`.

```
[num,headertext] = xlsread('climate2.xlsx','Temperatures')
```

```
num =  
     12     98  
     13     99  
     14     97
```

```
headertext =  
     'Time'     'Temp'
```

`xlsread` returns the numeric data in the array, `num`, and the text data in the cell array, `headertext`.

Read only the first row of data by specifying a range, 'A2:B2'.

```
row1 = xlsread('climate2.xlsx','Temperatures','A2:B2')
```



```
row1 =  
    12    98
```

Get Information about a Spreadsheet

To determine whether a file contains a readable Excel spreadsheet, use the `xlsfinfo` function. For readable files, `xlsfinfo` returns a nonempty string, such as '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 about the spreadsheet file, `climate2.xlsx`, created in the previous example:

```
[type, sheets] = xlsfinfo('climate2.xlsx')  
  
type =  
Microsoft Excel Spreadsheet  
sheets =  
    'Sheet1'    'Sheet2'    'Sheet3'    'Temperatures'
```

See Also

`xlsfinfo` | `xlsread` | `readtable`

Concepts

- “When to Convert Dates from Excel Files” on page 3-16
- “Access Data in a Table”

Import All Worksheets from a File

In this section...

“Import Numeric Data from All Worksheets” on page 3-12

“Import Data and Headers from All Worksheets” on page 3-12

Import Numeric Data from All Worksheets

This example shows how to import worksheets in an Excel file that contains only numeric data (no row or column headers, and no inner cells with text) into a structure array, using the `importdata` function.

Create a sample spreadsheet file for importing by writing an array of numeric data to the first and second worksheets in a file called `numdata.xlsx`.

```
xlswrite('numdata.xlsx',rand(5,5),1);
xlswrite('numdata.xlsx',rand(5,6),2);
```

Import the data from all worksheets in `numdata.xlsx`.

```
S = importdata('numdata.xlsx')
```

```
S =
```

```
Sheet1: [5x5 double]
Sheet2: [5x6 double]
```

`importdata` returns a structure array, `S`, with one field for each worksheet with data.

Import Data and Headers from All Worksheets

This example shows how to read numeric data and text headers from all worksheets in an Excel file into a nested structure array, using the `importdata` function.

This example uses a sample spreadsheet file, `testdata.xlsx`, that contains the following data in the first worksheet, and similar data in the second worksheet.

Time	Temp
12	98
13	99
14	97

Write a sample file, `testdata.xlsx`, for reading. Write an array of sample data, `d1`, to the first worksheet in the file and a second array, `d2`, to the second worksheet.

```
d1 = {'Time','Temp';
      12 98;
      13 99;
      14 97};
```

```
d2 = {'Time','Temp';
      12 78;
      13 77;
      14 78};
```

```
xlswrite('testdata.xlsx',d1,1);
xlswrite('testdata.xlsx',d2,2);
```

Read the data from all worksheets in `testdata.xlsx`.

```
climate = importdata('testdata.xlsx')

climate =

    data: [1x1 struct]
  txtdata: [1x1 struct]
 colheaders: [1x1 struct]
```

`importdata` returns a nested structure array, `climate`, with three fields, `data`, `txtdata`, and `colheaders`. Structures created from Excel files with row headers include the field `rowheaders` instead of `colheaders`.

View the contents of the structure array named `data`.

```
climate.data
```

```
ans =
```

```
Sheet1: [3x2 double]
Sheet2: [3x2 double]
```

`climate.data` contains one field for each worksheet with numeric data.

View the data in the worksheet named `Sheet1`.

```
climate.data.Sheet1
```

```
ans =
```

```
12    98
13    99
14    97
```

The field, `Sheet1`, contains the numeric data from the first worksheet in the file.

View the column headers in each sheet.

```
headers = climate.colheaders
```

```
headers =
```

```
Sheet1: {'Time' 'Temp'}
Sheet2: {'Time' 'Temp'}
```

Both the worksheets named `Sheet1` and `Sheet2` have the column headers, `Time` and `Temp`.

See Also

```
importdata
```

Concepts

- “When to Convert Dates from Excel Files” on page 3-16

System Requirements for Importing Spreadsheets

In this section...

“Importing Spreadsheets with Excel for Windows” on page 3-15

“Importing Spreadsheets Without Excel for Windows” on page 3-15

Importing Spreadsheets with Excel for Windows

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, XLSM, XLTM, and XLTX formats.
- `xlsread` also imports HTML-based formats.
- `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)
```

- 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.
- If you have Excel 2010, all MATLAB import options support ODS files.

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

Importing Spreadsheets Without Excel for Windows

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

- All MATLAB import options read XLS, XLSX, XLSM, XLTM, and XLTX files.

When to Convert Dates from Excel Files

In this section...
“MATLAB and Excel Dates” on page 3-16
“Import an Excel File with Numeric Dates” on page 3-17
“Export to an Excel File with Numeric Dates” on page 3-18

MATLAB and Excel Dates

Both MATLAB and Excel applications can represent dates 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 easily convert between string and numeric representations.

You must convert dates when:

- Importing any Excel file on a system without Excel for Windows
- Importing a spreadsheet with dates stored as numbers using `xlsread` or `importdata`
- Exporting an array or table with dates stored as numbers

You do not need to convert dates when:

- Importing a spreadsheet using the Import Tool
- Importing a spreadsheet with dates stored as strings on a system with Excel for Windows
- Exporting an array or table with dates stored as strings

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

Import 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');
```

Export 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);
```


Export to Excel Spreadsheets

In this section...

“Write Tabular Data to Spreadsheet File” on page 3-19

“Write Numeric and Text Data to Spreadsheet File” on page 3-20

“Disable Warning When Adding New Worksheet” on page 3-21

“Supported Excel File Formats” on page 3-21

“Format Cells in Excel Files” on page 3-21

Write Tabular Data to Spreadsheet File

This example shows how to export a table in the workspace to a Microsoft Excel spreadsheet file, using the `writetable` function. You can export data from the workspace to any worksheet in the file, and to any location within that worksheet. By default, `writetable` writes your table data to the first worksheet in the file, starting at cell A1.

Create a sample table of column-oriented data and display the first five rows.

```
load patients.mat
T = table(LastName, Age, Weight, Smoker);
T(1:5, :)
```

ans =

LastName	Age	Weight	Smoker
'Smith'	38	176	true
'Johnson'	43	163	false
'Williams'	38	131	false
'Jones'	40	133	false
'Brown'	49	119	false

Write table `T` to the first sheet in a new spreadsheet file named `patientdata.xlsx`, starting at cell D1. Specify the portion of the worksheet to write to, using the `Range` name-value pair argument.

```
filename = 'patientdata.xlsx';  
writetable(T,filename,'Sheet',1,'Range','D1')
```

By default, `writetable` writes the table variable names as column headings in the spreadsheet file.

Write table `T` to the second sheet in the file, but do not write the table variable names.

```
writetable(T,filename,'Sheet',2,'WriteVariableNames',false)
```

Write Numeric and Text Data to Spreadsheet File

This example shows how to export a numeric array and a cell array to a Microsoft Excel spreadsheet file, using the `xlswrite` function. You can export data in individual numeric and text workspace variables 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.

Create a sample array of numeric data, `A`, and a sample cell array of text and numeric data, `C`.

```
A = magic(5)  
C = {'Time', 'Temp'; 12 98; 13 'x'; 14 97}
```

A =

```
    17    24     1     8    15  
    23     5     7    14    16  
     4     6    13    20    22  
    10    12    19    21     3  
    11    18    25     2     9
```

C =

```
    'Time'    'Temp'  
    [ 12]    [ 98]  
    [ 13]    'x'  
    [ 14]    [ 97]
```

Write array *A* to the 5-by-5 rectangular region, E1:I5, on the first sheet in a new spreadsheet file named `testdata.xlsx`.

```
filename = 'testdata.xlsx';  
xlswrite(filename,A,1,'E1:I5')
```

Write the cell array, *C*, to a rectangular region that starts at cell B2 on a worksheet named `Temperatures` in the file. When you specify the sheet, you can specify a range using only the first cell.

```
xlswrite(filename,C,'Temperatures','B2');
```

`xlswrite` displays a warning because the worksheet, `Temperatures`, does not already exist in the file.

Disable Warning When Adding New Worksheet

If the target worksheet does not already exist in the file, the `writetable` and `xlswrite` functions display the following warning:

```
Warning: Added specified worksheet.
```

You can disable these warnings with this command:

```
warning('off','MATLAB:xlswrite:AddSheet')
```

Supported Excel File Formats

`writetable` and `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.

Note If you are using a system that does not have Excel for Windows installed, `writetable` and `xlswrite` write data to a comma-separated value (CSV) file.

Format 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 `writetable` or `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 “Getting Started with COM”.

See Also

`xlswrite` | `writetable`

Concepts

- “When to Convert Dates from Excel Files” on page 3-16

Low-Level File I/O

- “Import Text Data Files with Low-Level I/O” on page 4-2
- “Import Binary Data with Low-Level I/O” on page 4-11
- “Export to Text Data Files with Low-Level I/O” on page 4-19
- “Export Binary Data with Low-Level I/O” on page 4-26

Import Text Data Files with Low-Level I/O

In this section...
“Overview” on page 4-2
“Reading Data in a Formatted Pattern” on page 4-3
“Reading Data Line-by-Line” on page 4-6
“Testing for End of File (EOF)” on page 4-7
“Opening Files with Different Character Encodings” on page 4-9

Overview

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 “Ways to Import Text Files” on page 2-2.

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 4-3.
- `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 4-6.
- `fread`, which reads a stream of data at the byte or bit level. For more information, see “Import Binary Data with Low-Level I/O” on page 4-11.

For additional information, see:

- “Testing for End of File (EOF)” on page 4-7
- “Opening Files with Different Character Encodings” on page 4-9

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', 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 `my meas.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 4-6 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.

Import Binary Data with Low-Level I/O

In this section...

“Low-Level Functions for Importing Data” on page 4-11

“Reading Binary Data in a File” on page 4-12

“Reading Portions of a File” on page 4-14

“Reading Files Created on Other Systems” on page 4-17

“Opening Files with Different Character Encodings” on page 4-18

Low-Level Functions for Importing Data

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 “Supported File Formats for Import and Export” on page 1-2.

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 4-3.
- `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 4-6.
- `fread`, which reads a stream of data at the byte or bit level. For more information, see “Reading Binary Data in a File” on page 4-12.

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 4-12. Consider combining this method with “Testing for End of File” on page 4-14.
- Move to a specific location in a file to begin reading. For more information, see “Moving within a File” on page 4-15.
- Skip a certain number of bytes or bits after each element read. For an example, see “Writing and Reading Complex Numbers” on page 4-31.

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: fscanff, 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.

Export to Text Data Files with Low-Level I/O

In this section...
“Writing to Text Files” on page 4-19
“Appending or Overwriting Existing Files” on page 4-22
“Opening Files with Different Character Encodings” on page 4-25

Writing to Text Files

To create rectangular, delimited ASCII files (such as CSV files) from numeric arrays, use high-level functions such as `dlmwrite`. For more information, see “Write to Delimited Data Files” on page 2-26.

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.

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 — Append 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, call 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 — Overwrite an Existing Text File

This example shows two ways to replace characters in a text file.

A text file consists of a contiguous string of characters, including newline characters. To replace a line of the file with a different number of characters, you must rewrite the line that you want to change *and* all subsequent lines in the file.

For example, replace the first line of `changing.txt` (created in the previous example) with longer, descriptive text. Because the change applies to the first line, rewrite the entire file:

```
replaceLine = 1;
numLines = 5;
newText = 'This file originally contained a magic square';

fid = fopen('changing.txt','r');
mydata = cell(1, numLines);
for k = 1:numLines
    mydata{k} = fgetl(fid);
end
fclose(fid);

mydata{replaceLine} = newText;

fid = fopen('changing.txt','w');
fprintf(fid, '%s\n', mydata{:});
fclose(fid);
```

The file now contains:

```
This file originally contained a magic square
  2   11    7   14
  3   10    6   15
 13    8   12    1
 55   55   55   55
```

If you want to replace a portion of a text file with *exactly* the same number of characters, you do not need to rewrite any other lines in the file. For example, replace the third line of `changing.txt` with `[33 33 33 33]`:

```
replaceLine = 3;
myformat = '%5d %5d %5d %5d\n';
newData = [33 33 33 33];
```

```

% move the file position marker to the correct 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');

fprintf(fid, myformat, newData);
fclose(fid);

```

The file now contains:

This file originally contained a magic square

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.

Export Binary Data with Low-Level I/O

In this section...

“Low-Level Functions for Exporting Data” on page 4-26

“Writing Binary Data to a File” on page 4-27

“Overwriting or Appending to an Existing File” on page 4-27

“Creating a File for Use on a Different System” on page 4-29

“Opening Files with Different Character Encodings” on page 4-30

“Writing and Reading Complex Numbers” on page 4-31

Low-Level Functions for Exporting Data

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 “Supported File Formats for Import and Export” on page 1-2.

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 “Export to Text Data Files with Low-Level I/O” on page 4-19.
- `fwrite`, which writes a stream of binary data to a file. For more information, see “Writing Binary Data to a File” on page 4-27.

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

Use the `fwrite` function to export a stream of binary data to a file. As with any of the low-level I/O functions, before writing, open or create a file with `fopen`, and obtain a file identifier. When you finish processing a file, close it with `fclose`.

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 4-15.

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: fscanff, 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);
```


Images

- “Importing Images” on page 5-2
- “Exporting to Images” on page 5-6

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 5-3 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: 'matlabroot\toolbox\matlab\demos\ngc6543a.jpg'  
      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: {'CREATOR: XV Version 3.00b  Rev: 6/15/94  Quality =..
```

Reading Image Data and Metadata from TIFF Files

While you can use `imread` to import image data and metadata from 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 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 5-11, 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();
```

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 5-6 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 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 5-13. 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 5-7. 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 5-7. 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 5-13. 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 CIE Lab: 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	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.

Scientific Data

- “Importing CDF Files” on page 6-2
- “Exporting to CDF Files” on page 6-10
- “Importing NetCDF Files and OPeNDAP Data” on page 6-12
- “Exporting to NetCDF Files” on page 6-21
- “Importing Flexible Image Transport System (FITS) Files” on page 6-30
- “Importing HDF5 Files” on page 6-32
- “Exporting to HDF5 Files” on page 6-40
- “Import HDF4 Files Programatically” on page 6-52
- “Map HDF4 to MATLAB Syntax” on page 6-57
- “Import HDF4 Files Using Low-Level Functions” on page 6-59
- “Import HDF4 Files Interactively” on page 6-63
- “About HDF4 and HDF-EOS” on page 6-81
- “Export to HDF4 Files” on page 6-82

Importing CDF Files

In this section...
“Overview” on page 6-2
“High-Level CDF Import Functions” on page 6-2
“Using the CDF Library Low-Level Functions to Import Data” on page 6-6

Overview

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 6-2
- “Using the CDF Library Low-Level Functions to Import Data” on page 6-6

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 6-3
- “Reading Data from a CDF File” on page 6-4
- “Speeding Up Read Operations” on page 6-4
- “Representing CDF Time Values” on page 6-6

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 6-6.

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

```
whos
      Name                Size          Bytes  Class  Attributes
```

<code>data</code>	24x6	16512	cell
<code>data_combined</code>	1x6	2544	cell
<code>data_datenums</code>	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 CDF Files” on page 6-2). 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 “Exporting to CDF Files” on page 6-10.

```
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  
% cdfplib.getAttrgEntry function for global attributes.  
value = cdfplib.getAttrgEntry(cdfid,0,0)
```

```
value =
```

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

6 Close the CDF file.

```
cdfplib.close(cdfid);
```

Exporting to CDF Files

The Common Data Format (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 (write) data from a Common Data Format (CDF) file, 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 6-6.

```
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 };
cdflib.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]);
cdflib.hyperPutVarData(cdfid,image_id,recspec,dimspec,data);

```

4 Create a global attribute in the CDF file and write data to the attribute..

```

titleAttrNum = cdflib.createAttr(cdfid,'TITLE','global_scope');

% Write the global attribute entries
cdflib.putAttrEntry(cdfid,titleAttrNum,0,'CDF_CHAR','cdf Title');
cdflib.putAttrEntry(cdfid,titleAttrNum,1,'CDF_CHAR','Author');

```

5 Create attributes associated with variables in the CDF file and write data to the attribute.

```

fieldAttrNum = cdflib.createAttr(cdfid,'FIELDNAM','variable_scope');
unitsAttrNum = cdflib.createAttr(cdfid,'UNITS','variable_scope');

% Write the time variable attributes
cdflib.putAttrEntry(cdfid,fieldAttrNum,time_id,'CDF_CHAR','Time of observation');
cdflib.putAttrEntry(cdfid,unitsAttrNum,time_id,'CDF_CHAR','Hours');

```

6 Close the CDF file.

```

cdflib.close(cdfid);

```

Importing NetCDF Files and OPeNDAP Data

In this section...

“Overview” on page 6-12

“Using the MATLAB High-Level NetCDF Functions to Import Data” on page 6-12

“Using the MATLAB Low-Level NetCDF Functions to Import Data” on page 6-14

“Troubleshooting OPeNDAP Connections” on page 6-20

Overview

Network Common Data Form (NetCDF) is a set of software libraries and machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data. NetCDF 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, read the NetCDF documentation available at the Unidata Web site.

MATLAB provides two methods to import data from a NetCDF file or from an OPeNDAP source:

- High-level functions that simplify the process of importing data
- Low-level functions that enable more complete control over the importing process, by providing access to the routines in the NetCDF C library

Note For information about importing to Common Data Format (CDF) files, which have a completely separate, incompatible format, see “Importing CDF Files” on page 6-2.

Using the MATLAB High-Level NetCDF Functions to Import Data

MATLAB includes several functions that you can use to examine the contents of a NetCDF file and import data from the file into the MATLAB workspace.

- `ncdisp` — View the contents of a NetCDF file or OPeNDAP URL
- `ncinfo` — Create a structure that contains all the metadata that defines a NetCDF file
- `ncread` — Read data from a variable in a NetCDF file or OPeNDAP URL
- `ncreadatt` — Read data from an attribute associated with a variable in a NetCDF file or with the file itself (a global attribute).

For details about how to use these functions, see their reference pages, which include examples. The following section illustrates how to use these functions to perform a common task: finding all the unlimited dimensions in a NetCDF file.

Finding All Unlimited Dimensions in a NetCDF File

This example shows how to find all unlimited dimensions in an existing NetCDF file, visually and programmatically.

- 1 To determine which dimensions in a NetCDF file are unlimited, display the contents of the example NetCDF file, using `ncdisp`. The `ncdisp` function identifies unlimited dimensions with the label UNLIMITED.

```
Source:
        \\matlabroot\toolbox\matlab\demos\example.nc
Format:
        netcdf4
Global Attributes:
        creation_date = '29-Mar-2010'
Dimensions:
        x = 50
        y = 50
        z = 5
.
.
.
Groups:
        /grid2/
        Attributes:
                description = 'This is another group attribute.'
```

```

Dimensions:
    x    = 360
    y    = 180
    time = 0      (UNLIMITED)
Variables:
    temp
        Size:      []
        Dimensions: x,y,time
        Datatype:  int16

```

- 2** To determine all unlimited dimensions programmatically, first get information about the file using `ncinfo`. This example gets information about a particular group in the file.

```
ginfo = ncinfo('example.nc','/grid2/');
```

- 3** Get a vector of the Boolean values that indicate, for this group, which dimension is unlimited.

```
unlimDims = [ginfo.Dimensions.Unlimited]
```

```
unlimDims =
```

```
    0    0    1
```

- 4** Use this vector to display the unlimited dimension.

```
disp(ginfo.Dimensions(unlimDims))
    Name: 'time'
    Length: 0
    Unlimited: 1

```

Using the MATLAB Low-Level NetCDF Functions to Import Data

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`.

This section does not 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.

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 6-19

- 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 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 6-16. 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 “Exporting (Writing) Data to a NetCDF File” on page 6-26.)

```
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.

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	

Troubleshooting OPeNDAP Connections

If you have trouble reading OPeNDAP data, consider the following:

- OPeNDAP data is being pulled over the network from a server on the Internet. Pulling large data could be slow. Speed and reliability depends on their network connection
- OPeNDAP capability does not support proxy servers or any kind of authentication
- Failure to open an OPeNDAP link could have multiple causes:
 - Invalid URL
 - Local machine firewall/network firewall does not allow any external connections.
 - Local machine firewall/network firewall does not allow external connections on the OPeNDAP protocol.
 - Remote server is down.
 - Remote server will not serve the amount of data being requested. In this case, you can read data in subsets or chunks.
 - Remote server is incorrectly configured.

Exporting to NetCDF Files

In this section...
“Overview” on page 6-21
“Using the NetCDF High-Level Functions to Export Data” on page 6-21
“Using the NetCDF Low-Level Functions to Export Data” on page 6-26

Overview

Network Common Data Form (NetCDF) is a set of software libraries and machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data. NetCDF 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, read the NetCDF documentation available at the Unidata Web site.

MATLAB provides two methods to export data from the workspace into a NetCDF file:

- High-level functions that make it easy to export data
- Low-level functions that provide access to routines in the NetCDF C library

Note For information about exporting to Common Data Format (CDF) files, which have a completely separate and incompatible format, see “Exporting to NetCDF Files” on page 6-21.

Using the NetCDF High-Level Functions to Export Data

MATLAB includes several functions that you can use to export data from the file into the MATLAB workspace.

- `nccreate` — Create a variable in a NetCDF file. If the file does not exist, `nccreate` creates it.
- `ncwrite` — Write data to a NetCDF file

- `ncwriteatt` — Write data to an attribute associated with a variable in a NetCDF file or with the file itself (global attribute)
- `ncwriteschema` — Add a NetCDF schema to a NetCDF file, or create a new file using the schema as a template.

For details about how to use these functions, see their reference pages. These pages include examples. For information about importing (reading) data from a NetCDF file, see “Using the MATLAB High-Level NetCDF Functions to Import Data” on page 6-12. The following examples illustrate how to use these functions to perform several common scenarios:

- “Creating a New NetCDF File from an Existing File or Template” on page 6-22
- “Converting Between NetCDF File Formats” on page 6-23
- “Merging Two NetCDF Files” on page 6-24

Creating a New NetCDF File from an Existing File or Template

This example describes how to create a new file based on an existing file (or template).

- 1** Read the variable, dimension, and group definitions from the file using `ncinfo`. This information defines the file’s *schema*.

```
finfo = ncinfo('example.nc');
```

- 2** Create a new NetCDF file that uses this schema, using `ncwriteschema`.

```
ncwriteschema('mynewfile.nc',finfo);
```

- 3** View the existing file and the new file, using `ncdisp`. You can see how the new file contains the same set of dimensions, variables, and groups as the existing file.

Note A schema defines the structure of the file but does not contain any of the data that was in the original file.

```
ncdisp('example.nc')
ncdisp('mynewfile.nc')
```

Converting Between NetCDF File Formats

This example shows how to convert an existing file from one format to another.

Note When you convert a file's format using `ncwritschema`, you might get a warning message, if the original file format includes fields that are not supported by the new format. For example, the `netcdf4` format supports fill values but the NetCDF classic format does not. In these cases, `ncwritschema` still creates the file, but leaves out the field that is undefined in the new format.

- 1 Create a new file containing one variable, using the `nccreate` function.

```
nccreate('ex1.nc', 'myvar');
```

- 2 Determine the format of the new file, using `ncinfo`.

```
finfo = ncinfo('ex1.nc');
file_fmt = finfo.Format
```

```
file_fmt =
```

```
netcdf4_classic
```

- 3 Change the value of the `Format` field in the `finfo` structure to another supported NetCDF format. You use the `finfo` structure to specify the new format.

```
finfo.Format = 'netcdf4';
```

- 4 Create a new version of the file that uses the new format, using the `ncwritschema` function.

```
finfo = ncwritschema('newfile.nc', finfo);
finfo = ncinfo('newfile.nc');
new_fmt = finfo.Format
```

```
file_fmt =  
  
netcdf4
```

Note The new file contains the variable and dimension definitions of the original file, but does not contain the data. You must write the data to the file.

Merging Two NetCDF Files

This example shows how to merge two NetCDF files.

Note The combined file contains the variable and dimension definitions of the files that are combined, but does not contain the data in these original files.

- 1 Create a file, define a variable in the file, and write data to the variable.

```
nccreate('ex1.nc', 'myvar');  
ncwrite('ex1.nc', 'myvar', 55)  
ncdisp('ex1.nc')
```

- 2 Create a second file, with another variable, and write data to it.

```
nccreate('ex2.nc', 'myvar2');  
ncwrite('ex2.nc', 'myvar2', 99)  
ncdisp('ex2.nc')
```

- 3 Get the schema of each of the newly created files, using `ncinfo`.

```
finfo1 = ncinfo('ex1.nc')  
  
finfo1 =  
  
    Filename: 'H:\file1.nc'  
      Name: '/'  
Dimensions: []  
Variables: [1x1 struct]
```

```

    Attributes: []
      Groups: []
      Format: 'netcdf4_classic'

finfo2 = ncinfo('file2.nc')

finfo2 =

    Filename: 'H:\file2.nc'
      Name: '/'
    Dimensions: []
    Variables: [1x1 struct]
    Attributes: []
      Groups: []
      Format: 'netcdf4_classic'

```

- 4** Create a new NetCDF file that uses the schema of the first example file, using `ncwritschema`.

```

ncwritschema('combined_file.nc',finfo1);

ncdisp('combined_file.nc')
Source:
    H:\combined_file.nc
Format:
    netcdf4_classic
Variables:
    myvar1
        Size:      1x1
        Dimensions:
        Datatype:  double
        Attributes:
            _FillValue = 9.97e+036

```

- 5** Add the schema from the second example file to the newly created file, using `ncwritschema`. When you view the contents, notice how the file now contains the variable defined in the first example file and the variable defined in the second file.

```

ncwritschema('combined_file.nc',finfo2);

```

```
ncdisp('combined_file.nc')
Source:
      H:\combined_file.nc
Format:
      netcdf4_classic
Variables:
  myvar1
      Size:          1x1
      Dimensions:
      Datatype:     double
      Attributes:
                  _FillValue = 9.97e+036
  myvar2
      Size:          1x1
      Dimensions:
      Datatype:     double
      Attributes:
                  _FillValue = 9.97e+036
```

Using the NetCDF Low-Level Functions to Export Data

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`.

This section does not 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.

Exporting (Writing) Data to 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 folder.

- 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 `NO_CLOBBER` 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','NO_CLOBBER');
```

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
int8	NC_BYTE ¹
uint8	NC_BYTE ²
char	NC_CHAR
int16	NC_SHORT
uint16	No equivalent
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.

1. NetCDF interprets byte data as either signed or unsigned.
2. NetCDF interprets byte data as either signed or unsigned.

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  
.  
.  
.
```

Because you stored the data in the file as `NC_BYTE`, MATLAB reads the data from the variable into the workspace as class `int8`.

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 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. For a complete list of extensions, see `fitsread`.

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 primary data in the file or you can import the data in any of the extensions in the file, such as the Image extension, as shown in this example.

- 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 19:37:46'
    FileSize: 109440
  Contents: {'Primary' 'Binary Table' 'Unknown' 'Image' 'ASCII Table'}
  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 Binary Table, ASCII Table, and Image extensions.

2 Read data from the file.

To read the Primary data 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 Binary Table extension from the FITS file:

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

Importing HDF5 Files

In this section...
“Overview” on page 6-32
“Using the High-Level HDF5 Functions to Import Data” on page 6-32
“Using the Low-Level HDF5 Functions to Import Data” on page 6-39

Overview

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>).

MATLAB provides two methods to import data from an HDF5 file:

- High-level functions that make it easy to import data, when working with numeric datasets
- Low-level functions that enable more complete control over the importing process, by providing access to the routines in the HDF5 C library

Note For information about importing to HDF4 files, which have a completely separate, incompatible format, see “Import HDF4 Files Programatically” on page 6-52.

Using the High-Level HDF5 Functions to Import Data

MATLAB includes several functions that you can use to examine the contents of an HDF5 file and import data from the file into the MATLAB workspace.

Note You can only use the high-level functions to read numeric datasets or attributes. To read non-numeric datasets or attributes, you must use the low-level interface.

- `h5disp` — View the contents of an HDF5 file
- `h5info` — Create a structure that contains all the metadata defining an HDF5 file
- `h5read` — Read data from a variable in an HDF5 file
- `h5readatt` — Read data from an attribute associated with a variable in an HDF5 file or with the file itself (a global attribute).

For details about how to use these functions, see their reference pages, which include examples. The following sections illustrate some common usage scenarios.

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 objects without having to make a copy of the object.

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.

To get a quick view into the contents of an HDF5 file, use the `h5disp` function.

```
h5disp('example.h5')
```

```
HDF5 example.h5
Group '/'
```

```
Attributes:
  'attr1': 97 98 99 100 101 102 103 104 105 0
  'attr2': 2x2 H5T_INTEGER
Group '/g1'
  Group '/g1/g1.1'
    Dataset 'dset1.1.1'
      Size: 10x10
      MaxSize: 10x10
      Datatype: H5T_STD_I32BE (int32)
      ChunkSize: []
      Filters: none
      Attributes:
        'attr1': 49 115 116 32 97 116 116 114 105 ...
        'attr2': 50 110 100 32 97 116 116 114 105 ...
    Dataset 'dset1.1.2'
      Size: 20
      MaxSize: 20
      Datatype: H5T_STD_I32BE (int32)
      ChunkSize: []
      Filters: none
  Group '/g1/g1.2'
    Group '/g1/g1.2/g1.2.1'
      Link 'slink'
      Type: soft link
Group '/g2'
  Dataset 'dset2.1'
    Size: 10
    MaxSize: 10
    Datatype: H5T_IEEE_F32BE (single)
    ChunkSize: []
    Filters: none
  Dataset 'dset2.2'
    Size: 5x3
    MaxSize: 5x3
    Datatype: H5T_IEEE_F32BE (single)
    ChunkSize: []
    Filters: none
.
.
.
```

To explore the hierarchical organization of an HDF5 file, use the `h5info` function. `h5info` returns a structure that contains various information about the HDF5 file, including the name of the file.

```
info = h5info('example.h5')
info =

    Filename: 'matlabroot\matlab\toolbox\matlab\demos\example.h5'
      Name: '/'
   Groups: [4x1 struct]
  Datasets: []
 Datatypes: []
    Links: []
 Attributes: [2x1 struct]
```

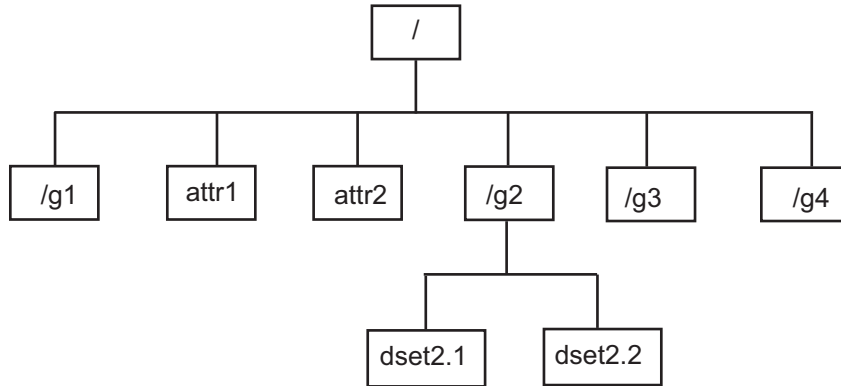
By looking at the `Groups` and `Attributes` fields, you can see that the file contains four groups and two attributes. The `Datasets`, `Datatypes`, and `Links` fields are all empty, indicating that the root group does not contain any data sets, data types, or links. To explore the contents of the sample HDF5 file further, examine one of the structures in `Groups`. The following example shows the contents of the second structure in this field.

```
level2 = info.Groups(2)

level2 =

      Name: '/g2'
   Groups: []
  Datasets: [2x1 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.



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

```
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: []
```

Importing Data from an HDF5 File

To read data or metadata from an HDF5 file, use the `h5read` function. As arguments, specify the name of the HDF5 file and the name of the data set. (To read the value of an attribute, you must use `h5readatt`.)

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

```
data = h5read('example.h5', '/g2/dset2.1')
```

```
data =
```

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

Mapping HDF5 Datatypes to MATLAB Datatypes

When the `h5read` function reads data from an HDF5 file into the MATLAB workspace, it maps HDF5 data types to MATLAB data types, as shown in the table below.

HDF5 Data Type	h5read Returns
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	Returns the actual data pointed to by the reference, not the value of the reference.
Strings, fixed-length and variable length	Cell array of strings

HDF5 Data Type	h5read Returns
Enums	Cell array of strings, where each enumerated value is replaced by the corresponding member name
Compound	1-by-1 struct array; the dimensions of the dataset are expressed in the fields of the structure.
Arrays	Array of values using the same datatype as the HDF5 array. For example, if the array is of signed 32-bit integers, the MATLAB array will be of type <code>int32</code> .

The example HDF5 file included with MATLAB includes examples of all these datatypes.

For example, the data set `/g3/string` is a string.

```
h5disp('example.h5','/g3/string')
HDF5 example.h5
Dataset 'string'
  Size: 2
  MaxSize: 2
  Datatype: H5T_STRING
    String Length: 3
    Padding: H5T_STR_NULLTERM
    Character Set: H5T_CSET_ASCII
    Character Type: H5T_C_S1
  ChunkSize: []
  Filters: none
  FillValue: ''
```

Now read the data from the file, MATLAB returns it as a cell array of strings.

```
s = h5read('example.h5','/g3/string')

s =

    'ab '
```

```

    'de '
>> whos s
    Name      Size      Bytes  Class  Attributes
    s         2x1       236    cell

```

The compound data types are always returned as a 1-by-1 struct. The dimensions of the data set are expressed in the fields of the struct. For example, the data set `/g3/compound2D` is a compound datatype.

```

h5disp('example.h5', '/g3/compound2D')
HDF5 example.h5
Dataset 'compound2D'
  Size: 2x3
  MaxSize: 2x3
  Datatype: H5T_COMPOUND
    Member 'a': H5T_STD_I8LE (int8)
    Member 'b': H5T_IEEE_F64LE (double)
  ChunkSize: []
  Filters: none
  FillValue: H5T_COMPOUND

```

Now read the data from the file, MATLAB returns it as a 1-by-1 struct.

```

data = h5read('example.h5', '/g3/compound2D')

data =

    a: [2x3 int8]
    b: [2x3 double]

```

Using the Low-Level HDF5 Functions to Import Data

MATLAB provides direct access to dozens of 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 to Export Data” on page 6-41.

Exporting to HDF5 Files

In this section...

“Overview” on page 6-40

“Using the MATLAB High-Level HDF5 Functions to Export Data” on page 6-40

“Using the MATLAB Low-Level HDF5 Functions to Export Data” on page 6-41

Overview

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>).

MATLAB provides two methods to export data to an HDF5 file:

- High-level functions that simplify the process of exporting data, when working with numeric datasets
- Low-level functions that provide a MATLAB interface to routines in the HDF5 C library

Note For information about exporting to HDF4 files, which have a completely separate and incompatible format, see “Export to HDF4 Files” on page 6-82.

Using the MATLAB High-Level HDF5 Functions to Export Data

The easiest way to write data or metadata from the MATLAB workspace to an HDF5 file is to use these MATLAB high-level functions.

Note You can use the high-level functions only with numeric data. To write nonnumeric data, you must use the low-level interface.

- `h5create` — Create an HDF5 dataset
- `h5write` — Write data to an HDF5 dataset
- `h5writeatt` — Write data to an HDF5 attribute

For details about how to use these functions, see their reference pages, which include examples. The following sections illustrate some common usage scenarios.

Writing a Numeric Array to an HDF5 Dataset

This example creates an array and then writes the array to an HDF5 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 Create the HDF5 file and the dataset, using `h5create`.

```
h5create('my_example_file.h5', '/dataset1', size(testdata))
```

- 3 Write the data to the HDF5 file.

```
h5write('my_example_file.h5', '/dataset1', testdata)
```

Using the MATLAB Low-Level HDF5 Functions to Export Data

MATLAB provides direct access to dozens of 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 to Export Data” on page 6-41.

The HDF5 library organizes the library functions into collections, 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 folder.

The following sections provide more detail about how to use the MATLAB HDF5 low-level functions.

- “Mapping HDF5 Function Syntax to MATLAB Function Syntax” on page 6-42
- “Mapping Between HDF5 Data Types and MATLAB Data Types” on page 6-45
- “Reporting Data Set Dimensions” on page 6-46
- “Writing Data to an HDF5 Data Set Using the MATLAB Low-Level Functions” on page 6-47
- “Preserving the Correct Layout of Your Data” on page 6-50

Note This section does not describe all features of the HDF5 library or explain basic HDF5 programming concepts. To use the MATLAB HDF5 low-level functions effectively, refer to the official HDF5 documentation available at <http://www.hdfgroup.org>.

Mapping HDF5 Function Syntax to MATLAB Function Syntax

In most cases, the syntax of the MATLAB low-level HDF5 functions matches 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 folder 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 describes some general differences that you should keep in mind when using the MATLAB low-level HDF5 functions.

- **HDF5 output parameters become MATLAB return values** — Some HDF5 library functions use function parameters to return data. Because MATLAB functions can return multiple values, these output parameters become return values. To illustrate, the HDF5 `H5Dread` 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. Also, in the MATLAB function, 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 are unnecessary** — The length parameter, used by some HDF5 library functions to specify the length of a string parameter, is 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.

```
buf = H5A.get_name(attr_id)
```

- **Use an empty array to specify NULL** — Wherever HDF5 library functions accept the value `NULL`, the corresponding MATLAB function uses empty arrays (`[]`). 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`.

- **Use cell arrays to specify multiple constants** — Some functions in the HDF5 library require you to specify an array of constants. For example, in the `H5Screate_simple` function, to specify that a dimension in the data space can be unlimited, you use the constant `H5S_UNLIMITED` for the 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 6-45 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 6-46 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.

Writing Data to an HDF5 Data Set Using the MATLAB 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 example 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 folder.

```
filename = fullfile(tempdir,'my_file.h5');

fileID = H5F.create(filename,'H5F_ACC_TRUNC','H5P_DEFAULT','H5P_DEFAULT');
```

In HDF5, 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, 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, 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 `H5S.create_simple` to preserve the layout of the data. You can use `flip1r` for this purpose. For a list of other HDF5 functions that require dimension flipping, see “Preserving the Correct Layout of Your Data” on page 6-50.

```
dims = size(testdata);  
dataspaceID = H5S.create_simple(2, flip1r(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, 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 “Writing Data to an HDF5 Data Set Using the MATLAB Low-Level Functions” on page 6-47.

- `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`

Import HDF4 Files Programatically

In this section...
“Overview” on page 6-52
“Using the MATLAB HDF4 High-Level Functions” on page 6-52

Overview

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.

Note For information about importing HDF5 data, which is a separate, incompatible format, see “Importing HDF5 Files” on page 6-32.

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 “Import HDF4 Files Using Low-Level Functions” on page 6-59.

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

- “Using `hdfinfo` to Get Information About an HDF4 File” on page 6-53
- “Using `hdfread` to Import Data from an HDF4 File” on page 6-53

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.

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 6-55.

- 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.

```

dssets = info.SDS

dssets =

    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

Map 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`). For the SD API and the HDF-EOS GD and SW APIs, MATLAB provides functions that map to individual C routines in the HDF4 library. These functions are implemented in the `matlab.io.hdf4.sd`, `matlab.io.hdfeos.gd`, and `matlab.io.hdfeos.sw` packages. For example, the SD API includes the C routine `SDendaccess` to close an HDF4 data set:

```
status = SDendaccess(sds_id); /* C code */
```

To call this routine from MATLAB, use the MATLAB function, `matlab.io.hdf4.sd.endAccess`. The syntax is similar:

```
sd.endAccess(sdsID)
```

For the remaining supported HDF4 APIs, MATLAB provides a single function that serves as a gateway to all the routines in the particular HDF4 API. For example, the HDF Annotations (AN) API includes the C routine `ANend` to terminate access to an AN interface:

```
status = ANend(an_id); /* C code */
```

To call this routine from MATLAB, use the MATLAB function associated with the AN API, `hdfan`. 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 = hdfan('end',an_id);
```

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 `ANget_tagref` routine returns the tag and reference number of an annotation in two output arguments, `ann_tag` and `ann_ref`. Here is the C code:

```
status = ANget_tagref(an_id,index,annot_type,ann_tag,ann_ref);
```

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

```
[tag,ref,status] = hdfan('get_tagref',AN_id,index,annot_type);
```

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.

Import HDF4 Files Using Low-Level Functions

This example shows how to read data from a Scientific Data Set in an HDF4 file, using the functions in the `matlab.io.hdf4.sd` package. In HDF4 terminology, the numeric arrays stored in HDF4 files are called data sets.

Add Package to Import List

Add the `matlab.io.hdf4.*` path to the import list.

```
import matlab.io.hdf4.*
```

Subsequent calls to functions in the `matlab.io.hdf4.sd` package need only be prefixed with `sd`, rather than the entire package path.

Open HDF4 File

Open the example HDF4 file, `sd.hdf`, and specify read access, using the `matlab.io.hdf4.sd.start` function. This function corresponds to the SD API routine, `SDstart`.

```
sdID = sd.start('sd.hdf','read');
```

`sd.start` returns an HDF4 SD file identifier, `sdID`.

Get Information About HDF4 File

Get the number of data sets and global attributes in the file, using the `matlab.io.hdf4.sd.fileInfo` function. This function corresponds to the SD API routine, `SDfileinfo`.

```
[ndatasets,ngatts] = sd.fileInfo(sdID)
```

```
ndatasets =
```

```
4
```

```
ngatts =
```

```
1
```

The file, `sd.hdf`, contains four data sets and one global attribute,

Get Attributes from HDF4 File

Get the contents of the first global attribute. HDF4 uses zero-based indexing, so an index value of 0 specifies the first index.

HDF4 files can optionally include information, called *attributes*, that describes the data that the file contains. Attributes associated with an entire HDF4 file are *global* attributes. Attributes associated with a data set are *local* attributes.

```
attr = sd.readAttr(sdID,0)
```

```
attr =
```

```
02-Sep-2010 11:13:16
```

Select Data Sets to Import

Determine the index number of the data set named `temperature`. Then, get the identifier of that data set.

```
idx = sd.nameToIndex(sdID, 'temperature');  
sdsID = sd.select(sdID, idx);
```

`sd.select` returns an HDF4 SD data set identifier, `sdsID`.

Get Information About Data Set

Get information about the data set identified by `sdsID` using the `matlab.io.hdf4.sd.getInfo` function. This function corresponds to the SD API routine, `SDgetinfo`.

```
[name,dims,datatype,nattrs] = sd.getInfo(sdsID)
```

```
name =
```

```
temperature
```

```
dims =
```



```

    20    10

datatype =

double

nattrs =

    11

```

`sd.getInfo` returns information about the name, size, data type, and number of attributes of the data set.

Read Entire Data Set

Read the entire contents of the data set specified by the data set identifier, `sdsID`.

```
data = sd.readData(sdsID);
```

Read Portion of Data Set

Read a 2-by-4 portion of the data set, starting from the first column in the second row. Use the `matlab.io.hdf4.sd.readData` function, which corresponds to the SD API routine, `SDreaddata`. The `start` input is a vector of index values specifying the location in the data set where you want to start reading data. The `count` input is a vector specifying the number of elements to read along each data set dimension.

```
start = [0 1];
count = [2 4];
data2 = sd.readData(sdsID,start,count)
```

```
data2 =

    21    41    61    81
    22    42    62    82
```

Close HDF4 Data Set

Close access to the data set, using the `matlab.io.hdf4.sd.endAccess` function. This function corresponds to the SD API routine, `SDendaccess`. You must close access to all the data sets in and HDF4 file before closing the file.

```
sd.endAccess(sdsID)
```

Close HDF4 File

Close the HDF4 file using the `matlab.io.hdf4.sd.close` function. This function corresponds to the SD API routine, `SDend`.

```
sd.close(sdID)
```

See Also

`sd.getInfo` | `sd.readData` | `sd.endAccess` | `sd.close` | `sd.start` | `sd.fileInfo`

Concepts

- “Map HDF4 to MATLAB Syntax” on page 6-57

Import HDF4 Files Interactively

Note The HDF Import Tool will be removed in a future release.

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:

In this section...
“Step 1: Opening an HDF4 File in the HDF Import Tool” on page 6-63
“Step 2: Selecting a Data Set in an HDF File” on page 6-65
“Step 3: Specifying a Subset of the Data (Optional)” on page 6-66
“Step 4: Importing Data and Metadata” on page 6-67
“Step 5: Closing HDF Files and the HDF Import Tool” on page 6-68
“Using the HDF Import Tool Subsetting Options” on page 6-68

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:

- On the **Home** tab, in the **Variable** section, click **Import Data**. 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
```

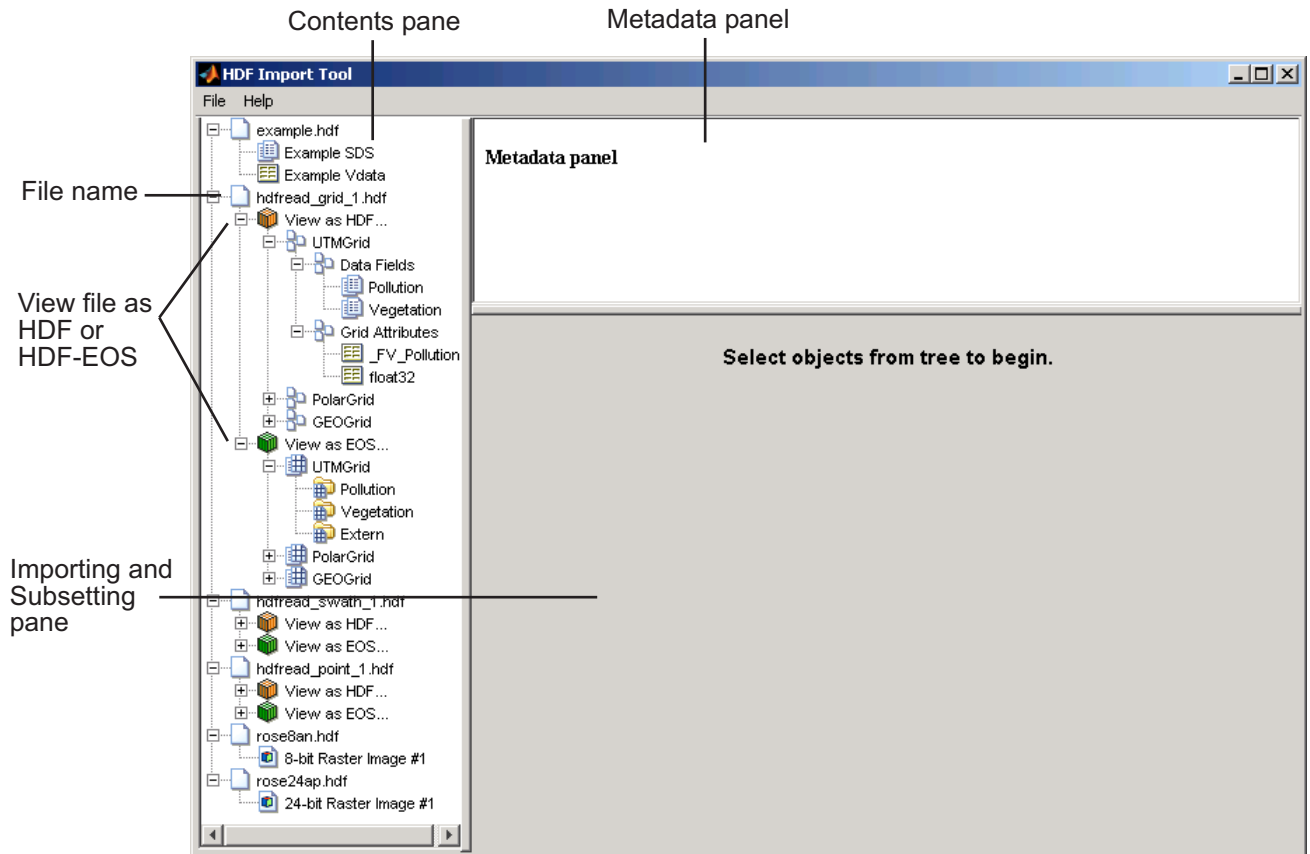
This opens an empty HDF Import Tool. To open a file, click the **Open** option on the HDFTool **File** menu and select the file you want to open. You can open multiple files in the HDF Import Tool.

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

```
hdftool('example.hdf')
```

Viewing a File in the HDF Import Tool

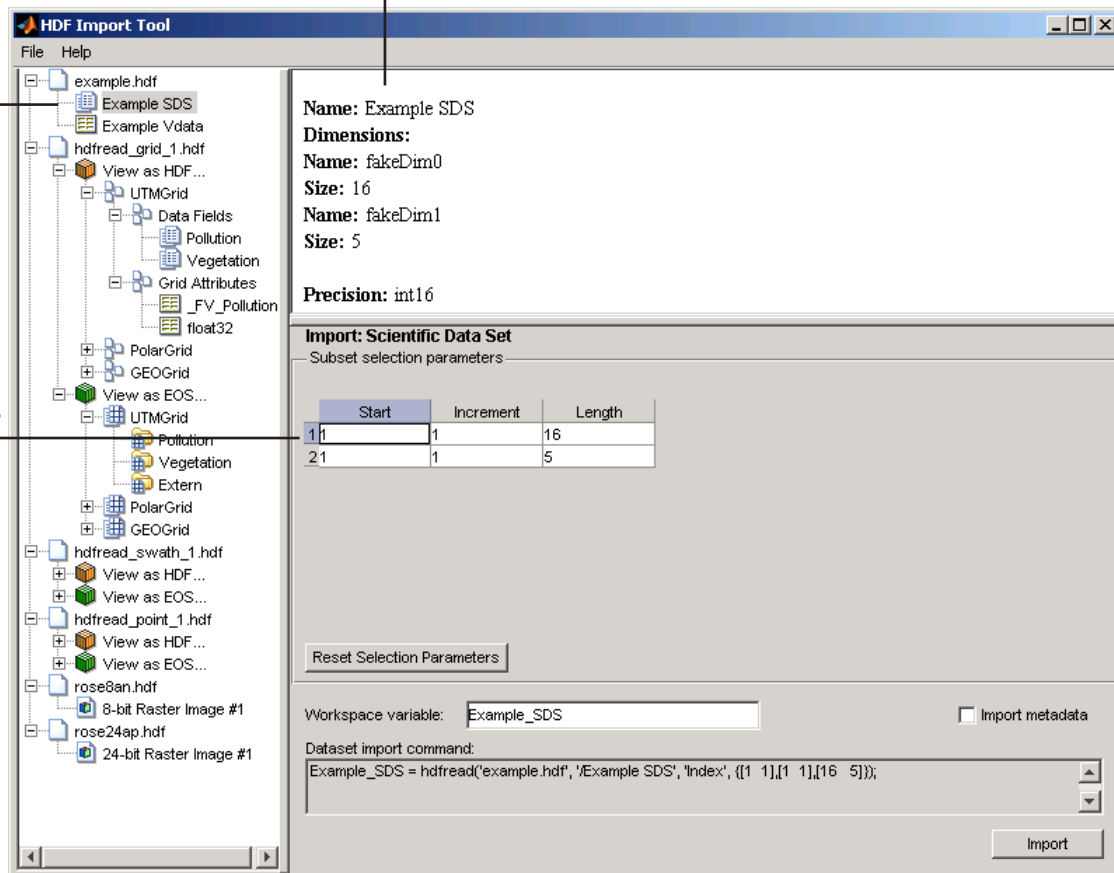
When you open an HDF4 or HDF-EOS file in the HDF Import Tool, the tool displays the contents of the file in the Contents pane. You can use this pane to navigate within the file to see what data sets it contains. You can view the contents of HDF-EOS files as HDF data sets or as HDF-EOS files. The icon 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
metadataSelected
data setSubsetting
options for this
HDF object

Step 3: Specifying a Subset of the Data (Optional)

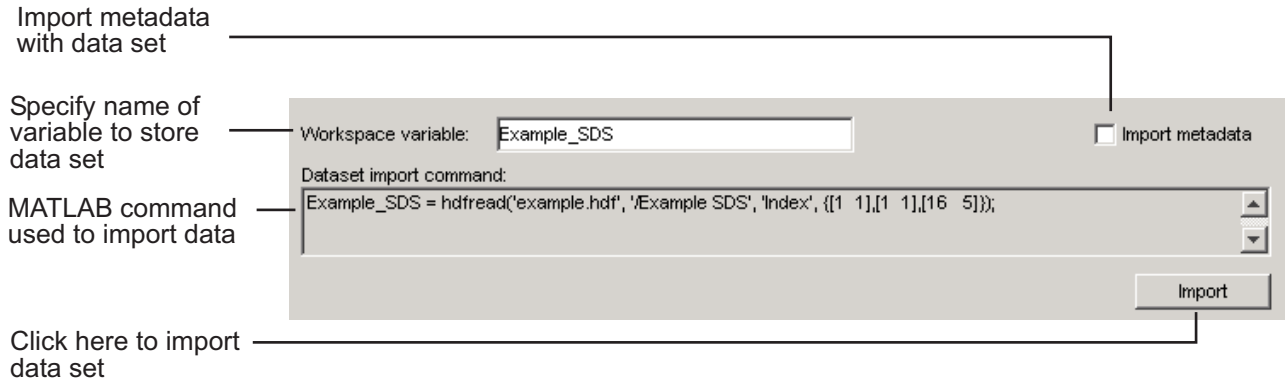
When you select a data set in the contents pane, the importing and subsetting pane displays the subsetting options available for that type of HDF object. The subsetting options displayed vary depending on the type of HDF object. For more information, see “Using the HDF Import Tool Subsetting Options” on page 6-68.

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

Note The HDF Import Tool will be removed in a future release.

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 “Import HDF4 Files Interactively” on page 6-63.

- “HDF Scientific Data Sets (SD)” on page 6-69
- “HDF Vdata” on page 6-70
- “HDF-EOS Grid Data” on page 6-71
- “HDF-EOS Point Data” on page 6-76
- “HDF-EOS Swath Data” on page 6-76
- “HDF Raster Image Data” on page 6-80

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

	Start	Increment	Length
Dimension 1	1	1	16
Dimension 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.

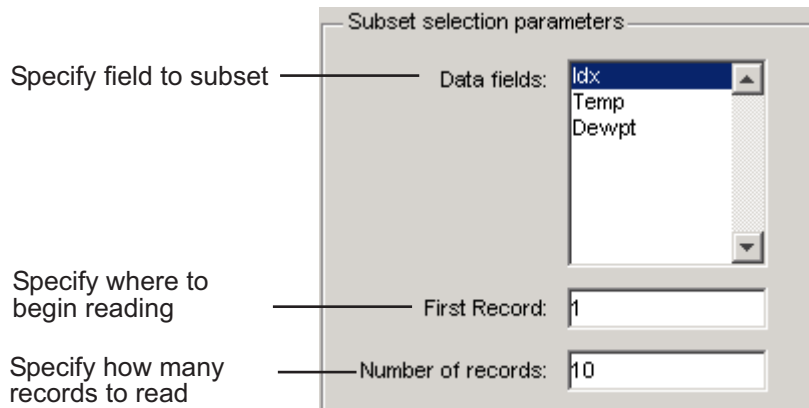
Fieldnames	idx	Temp	Dewpt
Records	1	0	5
	2	12	5
	3	3	7

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.



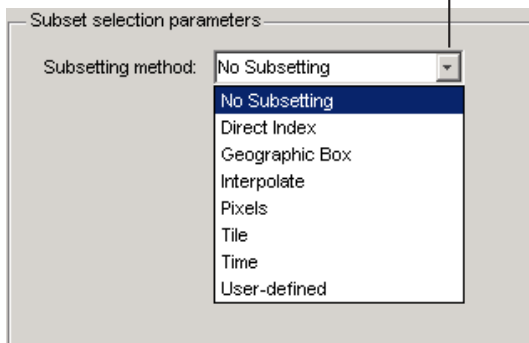
HDF-EOS Grid Data

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:

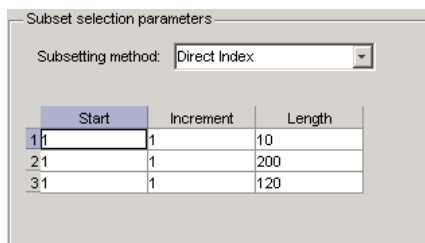
- “Direct Index” on page 6-72
- “Geographic Box” on page 6-73
- “Interpolation” on page 6-73
- “Pixels” on page 6-74
- “Tile” on page 6-74
- “Time” on page 6-75
- “User-Defined” on page 6-75

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

Click here to see options



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)

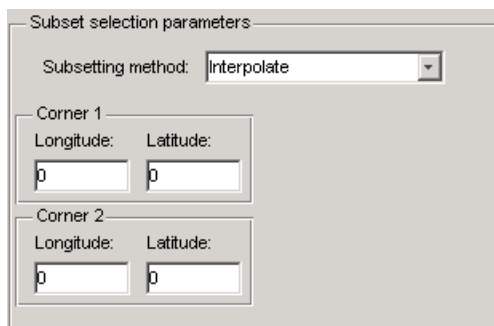
Dimension or Field Name:	Min:	Max:
DIM:Time		
DIM:Time		
DIM:Time		

Optionally, you can further define the subset of data you are interested in by using Time parameters (see “Time” on page 6-75) or by specifying other User-Defined subsetting parameters (see “User-Defined” on page 6-75).

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:

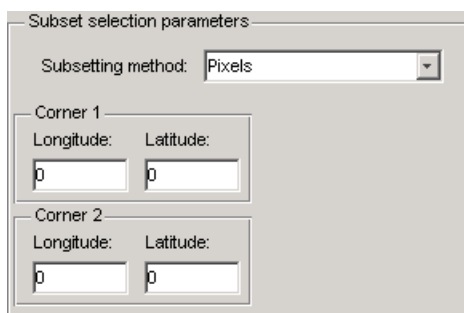
- **Corner 1** – Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box



The screenshot shows a 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: "Corner 1" and "Corner 2". Each section contains two input fields: "Longitude:" and "Latitude:". In the "Corner 1" section, both fields contain the number "0". In the "Corner 2" section, both fields also contain 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



The screenshot shows a dialog box titled "Subset selection parameters". At the top, there is a dropdown menu labeled "Subsetting method:" with "Pixels" selected. Below this, there are two sections: "Corner 1" and "Corner 2". Each section contains two input fields: "Longitude:" and "Latitude:". In the "Corner 1" section, both fields contain the number "0". In the "Corner 2" section, both fields also contain the number "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 Coordinates:

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

Start: Stop:

User-defined (optional)

Dimension or Field Name:	Min:	Max:
<input type="text" value="DIM:Time"/>	<input type="text"/>	<input type="text"/>
<input type="text" value="DIM:Time"/>	<input type="text"/>	<input type="text"/>
<input type="text" value="DIM:Time"/>	<input type="text"/>	<input type="text"/>

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.

Subset selection parameters

Subsetting method:

User-defined

Dimension or Field Name:	Min:	Max:
<input type="text" value="DIM:Time"/>	<input type="text"/>	<input type="text"/>
<input type="text" value="DIM:Time"/>	<input type="text"/>	<input type="text"/>
<input type="text" value="DIM:Time"/>	<input type="text"/>	<input type="text"/>

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 6-73. For information about subsetting by time, see “Time” on page 6-75.

The image shows a dialog box titled "Subset selection parameters". On the left, there is a "Data fields:" label followed by a list box containing "Time", "Concentration", and "Species". Below the list box is a "Level:" label with a text input field containing the number "1". At the bottom left is a "Record (optional):" label with an empty text input field. On the right side, there are three vertically stacked sections, each with a title and two input fields. The first section is titled "Corner 1 (optional)" and has "Longitude:" and "Latitude:" labels. The second section is titled "Corner 2 (optional)" and also has "Longitude:" and "Latitude:" labels. The third section is titled "Time (optional)" and has "Start:" and "Stop:" labels. All input fields are currently empty.

HDF-EOS Swath Data

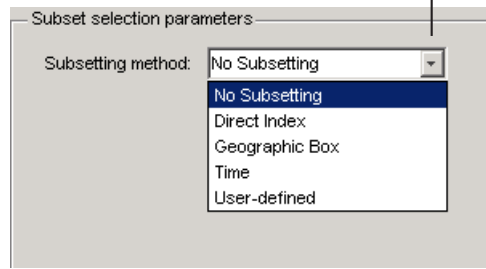
HDF-EOS Swath data is data that is produced by a satellite as it traces a path over the earth. This path is called its ground track. The sensor aboard the satellite takes a series of scans perpendicular to the ground track. Swath data can also include a vertical measure as a third dimension. For example,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

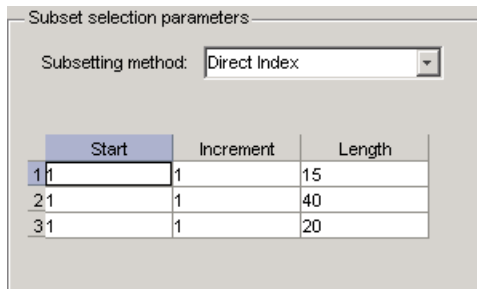
- “Direct Index” on page 6-77
- “Geographic Box” on page 6-78
- “Time” on page 6-79
- “User-Defined” on page 6-80

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.



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.

Subset selection parameters

Subsetting method: Geographic Box

Corner 1
Longitude: 0 Latitude: 0

Corner 2
Longitude: 0 Latitude: 0

Selection mode
Cross Track Inclusion: AnyPoint
Geolocation Mode: Internal

Time (optional)
Start: Stop:

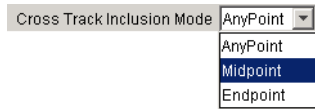
User-defined (optional)

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

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.

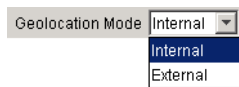
You specify the selection mode by choosing the type of **Cross Track Inclusion** and the **Geolocation mode**. The **Cross Track Inclusion** value determines how much of the area of the geographic box that you define must fall within the boundaries of the swath.



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.

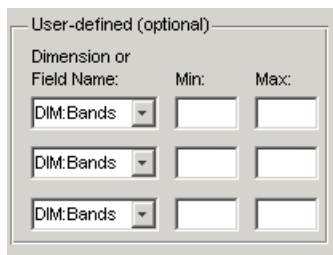


Select from these values:

- **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.



The image shows a dialog box titled "User-defined (optional)". It contains a table with three rows and three columns. The first column is labeled "Dimension or Field Name:", the second is "Min:", and the third is "Max:". Each row has a dropdown menu in the first column, and two empty text input fields in the second and third columns. All three dropdown menus are currently set to "DIM:Bands".

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

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.

About HDF4 and HDF-EOS

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).

HDF4 Application Programming Interfaces (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. MATLAB functions allow you to access specific HDF4 APIs.

To use the MATLAB HDF4 functions effectively, you must be familiar with the HDF library. For detailed information about HDF4 features and routines, refer to the documentation at the HDF Web site.

Export to HDF4 Files

In this section...

“Write MATLAB Data to HDF4 File” on page 6-82

“Manage HDF4 Identifiers” on page 6-84

Write MATLAB Data to HDF4 File

This example shows how to write MATLAB arrays to a Scientific Data Set in an HDF4 file.

Add Package to Import List

Add the `matlab.io.hdf4.*` path to the import list.

```
import matlab.io.hdf4.*
```

Prefix subsequent calls to functions in the `matlab.io.hdf4.sd` package with `sd`, rather than the entire package path.

Create HDF4 File

Create a new HDF4 file using the `matlab.io.hdf4.sd.start` function. This function corresponds to the SD API routine, `SDstart`.

```
sdID = sd.start('mydata.hdf', 'create');
```

`sd.start` creates the file and returns a file identifier named `sdID`.

To open an existing file instead of creating a new one, call `sd.start` with `'write'` access instead of `'create'`.

Create HDF4 Data Set

Create a data set in the file for each MATLAB array you want to export. If you are writing to an existing data set, you can skip ahead to the next step. In this example, create one data set for the array of sample data, `A`, using the `matlab.io.hdf4.sd.create` function. This function corresponds to the SD API routine, `SDcreate`. The `ds_type` argument is a string specifying the MATLAB data type of the data set.

```
A = [1 2 3 4 5 ; 6 7 8 9 10 ; 11 12 13 14 15];
ds_name = 'A';
ds_type = 'double';
ds_dims = size(A);
sdsID = sd.create(sdsID,ds_name,ds_type,ds_dims);
```

`sd.create` returns an HDF4 SD data set identifier, `sdsID`.

Write MATLAB Data to HDF4 File

Write data in `A` to the data set in the file using the `matlab.io.hdf4.sd.writedata` function. This function corresponds to the SD API routine, `SDwritedata`. The `start` argument specifies the zero-based starting index.

```
start = [0 0];
sd.writeData(sdsID,start,A);
```

`sd.writeData` queues the write operation. Queued operations execute when you close the HDF4 file.

Write MATLAB Data to Portion of Data Set

Replace the second row of the data set with the vector `B`. Use a `start` input value of `[1 0]` to begin writing at the second row, first column. `start` uses zero-based indexing.

```
B = [9 9 9 9 9];
start = [1 0];
sd.writeData(sdsID,start,B);
```

Write Metadata to HDF4 File

Create a global attribute named `creation_date`, with a value that is the current date and time. Use the `matlab.io.hdf4.sd.setAttr` function, which corresponds to the SD API routine, `SDsetattr`.

```
sd.setAttr(sdsID,'creation_date',datestr(now));
```

`sd.Attr` creates a file attribute, also called a global attribute, associated with the HDF4 file identified by `sdsID`.

Associate a predefined attribute, `cordsys`, to the data set identified by `sdsID`. Possible values of this attribute include the text strings `'cartesian'`, `'polar'`, and `'spherical'`.

```
attr_name = 'cordsys';  
attr_value = 'polar';  
sd.setAttr(sdsID,attr_name,attr_value);
```

Close HDF4 Data Set

Close access to the data set, using the `matlab.io.hdf4.sd.endAccess` function. This function corresponds to the SD API routine, `SDendaccess`. You must close access to all the data sets in and HDF4 file before closing the file.

```
sd.endAccess(sdsID);
```

Close HDF4 File

Close the HDF4 file using the `matlab.io.hdf4.sd.close` function. This function corresponds to the SD API routine, `SDend`.

```
sd.close(sdID);
```

Closing an HDF4 file executes all the write operations that have been queued using `SDwritedata`.

Manage HDF4 Identifiers

MATLAB supports utility functions that make it easier to use HDF4 in the MATLAB environment.

- “View All Open HDF4 Identifiers” on page 6-84
- “Close All Open HDF4 Identifiers” on page 6-85

View All Open HDF4 Identifiers

Use the gateway function to the MATLAB HDF4 utility API, `hdfml`, and specify the name of the `listinfo` routine as an argument to view all the currently open HDF4 identifiers. MATLAB updates this list whenever HDF4 identifiers are created or closed. In this 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
```

Close All Open HDF4 Identifiers

Close all the currently open HDF4 identifiers in a single call using the gateway function to the MATLAB HDF4 utility API, `hdfml`. Specify the name of the `closeall` routine as an argument:

```
hdfml('closeall')
```

See Also

`sd.start` | `sd.create` | `sd.writeData` | `sd.setAttr` | `sd.close` |
`sd.endAccess` | `hdfml`

Concepts


- “Map HDF4 to MATLAB Syntax” on page 6-57

Audio and Video

- “Read and Get Information About Audio Files” on page 7-2
- “Record and Play Audio” on page 7-3
- “Get Information about Video Files” on page 7-9
- “Read Video Files” on page 7-10
- “Supported Video File Formats” on page 7-13
- “Convert Between Image Sequences and Video” on page 7-16
- “Export to Audio and Video” on page 7-20
- “Characteristics of Audio Files” on page 7-22

Read and Get Information About Audio Files

Use the `audioread` function to read audio data from a file. `audioread` can support WAVE, OGG, FLAC, AU, MP3, and MPEG-4 AAC files.

You also can read WAV, AU, or SND files interactively. Select  **Import Data** or double-click the file name in the Current Folder browser.

To get information about audio files, use `audioinfo`. The `audioinfo` function returns information such as the number of audio channels, sample rate, duration, bits per sample, bit rate, and metadata, as applicable.

Record and Play Audio

In this section...

“Record Audio” on page 7-3

“Play Audio” on page 7-6

“Record or Play Audio within a Function” on page 7-7

Record 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. The recording is performed asynchronously.
 - `recordblocking` retains control until the recording is complete. Specify the length of the recording in seconds. The recording is performed synchronously.
- 3 Create a numeric array corresponding to the signal data using the `getaudiodata` method.

The following examples show how to use the `recordblocking` and `record` methods.

Record Microphone Input

This example shows how to record microphone input, play back the recording, and store the recorded audio signal in a numeric array. You must first connect a microphone to your system.

Create an `audiorecorder` object named `recObj` for recording audio input.

```
recObj = audiorecorder
```

```
recObj =  
  
    audiorecorder with properties:  
  
        SampleRate: 8000  
        BitsPerSample: 8  
        NumberOfChannels: 1  
        DeviceID: -1  
        CurrentSample: 1  
        TotalSamples: 0  
        Running: 'off'  
        StartFcn: []  
        StopFcn: []  
        TimerFcn: []  
        TimerPeriod: 0.0500  
        Tag: ''  
        UserData: []  
        Type: 'audiorecorder'
```

audiorecorder creates an 8000 Hz, 8-bit, 1-channel audiorecorder object.

Record your voice for 5 seconds.

```
disp('Start speaking.')
```

```
recordblocking(recObj, 5);
```

```
disp('End of Recording.');
```

Play back the recording.

```
play(recObj);
```

Store data in double-precision array, y.

```
y = getaudiodata(recObj);
```

Plot the audio samples.

```
plot(y);
```

Record Two Channels from Different Sound Cards

To record audio independently from two different sound cards, with a microphone connected to each:

- 1 Call `audiodevinfo` to list the available sounds cards. For example, this code returns a structure array containing all input and output audio devices on your system:

```
info = audiodevinfo;
```

Identify the sound cards you want to use by name, and note their ID values.

- 2 Create two `audiorecorder` objects. For example, this code creates the `audiorecorder` object, `recorder1`, for recording a single channel from device 3 at 44.1 kHz and 16 bits per sample. The `audiorecorder` object, `recorder2`, is for recording a single channel from device 4 at 48 kHz:

```
recorder1 = audiorecorder(44100,16,1,3);  
recorder2 = audiorecorder(48000,16,1,4);
```

- 3 Record each audio channel separately.

```
record(recorder1);  
record(recorder2);  
pause(5);
```

The recordings occur simultaneously as the first call to `record` does not block.

- 4 Stop the recordings.

```
stop(recorder1);  
stop(recorder2);
```

Specify 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.

Play 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 sample 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 sample file:

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

For an additional example, see “Record or Play Audio within a Function” on page 7-7.

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.

Record or Play Audio within a Function

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 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)
```

Get 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 file `xylophone.mp4`:

```
xyloObj = VideoReader('xylophone.mp4');  
info = get(xyloObj)
```

The `get` function returns:

```
info =  
  
    Duration: 4.7000  
    Name: 'xylophone.mp4'  
    Path: [1x88 char]  
    Tag: ''  
    Type: 'VideoReader'  
    UserData: []  
    BitsPerPixel: 24  
    FrameRate: 30  
    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 “Read Variable Frame Rate Video” on page 7-11.

See Also `get`

Read Video Files

In this section...

“Import Video Data from a File” on page 7-10

“Display Video Frame with Colormap” on page 7-10

“Process Frames of a Video File” on page 7-10

“Read Variable Frame Rate Video” on page 7-11

Import Video Data from a File

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 file `xylophone.mp4`:

```
xyloObj = VideoReader('xylophone.mp4');  
vidFrames = read(xyloObj);
```

It is not necessary to close the multimedia object.

Display Video Frame with Colormap

Use the `read` function to specify a video frame to read. For example, read the second frame in a hypothetical file, `indexedImg.avi`.

```
obj = VideoReader('indexedImg.avi');  
vidFrame = read(obj,2);
```

Use the `image` function to display the video frame. For indexed video, you can apply a colormap. In this case, `gray(256)` applies a linear grayscale colormap of length 256.

```
image(vidFrame)  
colormap(gray(256))
```

Process Frames of a Video File

This example shows how to save memory by processing a video one frame at a time. For faster processing, preallocate memory to store the video data.

Convert the file `xylophone.mp4` to a MATLAB movie, and play it with the `movie` function

Create a `VideoReader` object for `xylophone.mp4`. Determine the number of frames, and the height and width of the frames.

```
xyloObj = VideoReader('xylophone.mp4');
```

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

Preallocate the 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);
```

Read 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.

See Also

`VideoReader` | `mmfileinfo` | `movie`

Concepts

- “Supported Video File Formats” on page 7-13

Supported Video File Formats

In this section...

“What Are Video Files?” on page 7-13

“Formats That VideoReader Supports” on page 7-13

“View Codec Associated with Video File” on page 7-14

“Troubleshooting: Errors Reading Video File” on page 7-15

What Are Video Files?

For video data, the term “file format” often refers to either the *container format* or the *codec*. A container format describes the layout of the file, while a codec describes how to encode/decode the video data. Many container formats can hold data encoded with different codecs.

To read a video file, any application must:

- Recognize the container format (such as AVI).
- Have access to the codec that can decode the video data stored in the file. Some codecs are part of standard Windows and Macintosh system installations, and allow you to play video in Windows Media Player or QuickTime. In MATLAB, VideoReader can access most, but not all, of these codecs.
- Properly use the codec to decode the video data in the file. VideoReader cannot always read files associated with codecs that were not part of your original system installation.

Formats That VideoReader Supports

Use VideoReader to read video files in MATLAB. The file formats that VideoReader supports vary by platform, and have no restrictions on file extensions.

All Platforms	AVI, including uncompressed, indexed, grayscale, and Motion JPEG-encoded video (.avi) Motion JPEG 2000 (.mj2)
All Windows	MPEG-1 (.mpg) Windows Media Video (.wmv, .asf, .asx) Any format supported by Microsoft DirectShow
Windows 7 or later	MPEG-4, including H.264 encoded video (.mp4, .m4v) Apple QuickTime Movie (.mov) Any format supported by Microsoft Media Foundation
Macintosh	Most formats supported by QuickTime Player, including: MPEG-1 (.mpg) MPEG-4, including H.264 encoded video (.mp4, .m4v) Apple QuickTime Movie (.mov) 3GPP 3GPP2 AVCHD DV
Linux	Any format supported by your installed plug-ins for GStreamer 0.10 or above, as listed on http://gstreamer.freedesktop.org/documentation/plugins.html , including Ogg Theora (.ogg).

View Codec Associated with Video File

This example shows how to view the codec associated with a video file, using the `mmfileinfo` function.

Store information about the sample video file, `shuttle.avi`, in a structure array named `info`.

```
info = mmfileinfo('shuttle.avi');
```

View the `Format` field of the structure.

```
info.Video.Format
```

```
ans =
```


MJPG

The file, `shuttle.avi`, uses the Motion JPEG codec.

Troubleshooting: Errors Reading Video File

You might be unable to read a video file if MATLAB cannot access the appropriate codec. 64-bit applications use 64-bit codec libraries, while 32-bit applications use 32-bit codec libraries. For example, when working with 64-bit MATLAB, you cannot read video files that require access to a 32-bit codec installed on your system. You might need to install a 32-bit version of the codec, or create a video file with a codec whose 64-bit version is available and installed on your computer.

Sometimes, `VideoReader` cannot open a video file for reading on Windows platforms. This might occur if you have installed a third-party codec that overrides your system settings. Uninstall the codec and try opening the video file in MATLAB again.

Convert Between Image Sequences and Video

This example shows how to convert between video files and sequences of image files using `VideoReader` and `VideoWriter`.

The sample file named `shuttle.avi` contains 121 frames. Convert the frames to image files using `VideoReader` and the `imwrite` function. Then, convert the image files to an AVI file using `VideoWriter`.

Setup

Create a temporary working folder to store the image sequence.

```
workingDir = tempname;  
mkdir(workingDir);  
mkdir(workingDir, 'images');
```

Construct a VideoReader Object

Create a `VideoReader` object to use for reading frames from the file.

```
shuttleVideo = VideoReader('shuttle.avi');
```

Create the Image Sequence

Loop through the video, reading each frame into a width-by-height-by-3 array named `img`. Write out each image to a JPEG file with a name in the form `imgN.jpg`, where `N` is the frame number:

```
    img1.jpg  
    img2.jpg  
    ...  
    img121.jpg  
  
for ii = 1:shuttleVideo.NumberOfFrames  
    img = read(shuttleVideo,ii);  
  
    % Write out to a JPEG file (img1.jpg, img2.jpg, etc.)  
    imwrite(img,fullfile(workingDir, 'images',sprintf('img%d.jpg',ii)));  
end
```

Read and Sort the Image Sequence into MATLAB®

Find all the JPEG file names in the `images` folder. Convert the set of image names to a cell array.

```
imageNames = dir(fullfile(workingDir, 'images', '*.jpg'));  
imageNames = {imageNames.name}';
```

Notice that the image file names are not in numeric order.

```
disp(imageNames(1:10));
```

```
'img1.jpg'  
'img10.jpg'  
'img100.jpg'  
'img101.jpg'  
'img102.jpg'  
'img103.jpg'  
'img104.jpg'  
'img105.jpg'  
'img106.jpg'  
'img107.jpg'
```

To sort the file names, extract the frame numbers from the file names and use them to sort the array.

First, match any file names that contain a sequence of numeric digits. Convert the strings to doubles.

```
imageStrings = regexp([imageNames{:}], '(\\d*)', 'match');  
imageNumbers = str2double(imageStrings);
```

Sort the frame numbers from lowest to highest. The `sort` function returns an index matrix that indicates how to order the associated files.

```
[~,sortedIndices] = sort(imageNumbers);  
sortedImageNames = imageNames(sortedIndices);
```

The sequence file names are now sorted.

```
disp(sortedImageNames(1:10));
```

```
'img1.jpg'  
'img2.jpg'  
'img3.jpg'  
'img4.jpg'  
'img5.jpg'  
'img6.jpg'  
'img7.jpg'  
'img8.jpg'  
'img9.jpg'  
'img10.jpg'
```

Create a New Video with the Image Sequence

Construct a `VideoWriter` object, which creates a Motion-JPEG AVI file by default.

```
outputVideo = VideoWriter(fullfile(workingDir, 'shuttle_out.avi'));  
outputVideo.FrameRate = shuttleVideo.FrameRate;  
open(outputVideo);
```

Loop through the image sequence, load each image, and then write it to the video.

```
for ii = 1:length(sortedImageNames)  
    img = imread(fullfile(workingDir, 'images', sortedImageNames{ii}));  
  
    writeVideo(outputVideo, img);  
end
```

Finalize the video file.

```
close(outputVideo);
```

View the Final Video

Construct a reader object.

```
shuttleAvi = VideoReader(fullfile(workingDir, 'shuttle_out.avi'));
```

Create a MATLAB movie struct from the video frames.

```
mov(shuttleAvi.NumberOfFrames) = struct('cdata',[],'colormap',[]);
for ii = 1:shuttleAvi.NumberOfFrames
    mov(ii) = im2frame(read(shuttleAvi,ii));
end
```

Resize the current figure and axes based on the video's width and height, and view the first frame of the movie.

```
set(gcf,'position',[150 150 shuttleAvi.Width shuttleAvi.Height])
set(gca,'units','pixels');
set(gca,'position',[0 0 shuttleAvi.Width shuttleAvi.Height])

image(mov(1).cdata,'Parent',gca);
axis off;
```

Play back the movie once at the video's frame rate.

```
movie(mov,1,shuttleAvi.FrameRate);
```

Credits

Video of the Space Shuttle courtesy of NASA.

Export to Audio and Video

In this section...
“Export to Audio Files” on page 7-20
“Export Video to AVI Files” on page 7-20

Export to Audio Files

In MATLAB, audio data is simply numeric data that you can export using standard MATLAB data export functions, such as `save`.

You also can export audio data to files in specific file formats using the `audiowrite` function.

Export 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:

```
uncompressedVideo = 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);
```

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.

XML Documents

- “Importing XML Documents” on page 8-2
- “Exporting to XML Documents” on page 8-6

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. For more information, see:

- “What Is an XML Document Object Model (DOM)?” on page 8-2
- “Example — Finding Text in an XML File” on page 8-3

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, see the `org.w3c.dom` package description at <http://download.oracle.com/javase/6/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.

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 8-2

For more information, see:

- “Creating an XML File” on page 8-6
- “Updating an Existing XML File” on page 8-8

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, see the `org.w3c.dom` package description at <http://download.oracle.com/javase/6/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 “Display Custom Documentation”), 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, see the `org.w3c.dom` package description at <http://download.oracle.com/javase/6/docs/api/>.

For examples that use these methods, see:

- “Example — Finding Text in an XML File” on page 8-3
- “Example — Creating an XML File with `xmlwrite`” on page 8-7
- `xmlread` and `xmlwrite`

Memory-Mapping Data Files

- “Overview of Memory-Mapping” on page 9-2
- “Map File to Memory” on page 9-7
- “Read Mapped File” on page 9-12
- “Write to Mapped File” on page 9-19
- “Delete Memory Map” on page 9-27
- “Share Memory Between Applications” on page 9-28

Overview of Memory-Mapping

In this section...
“What Is Memory-Mapping?” on page 9-2
“Benefits of Memory-Mapping” on page 9-2
“When to Use Memory-Mapping” on page 9-4
“Maximum Size of a Memory Map” on page 9-5
“Byte Ordering” on page 9-6

What Is 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`.

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

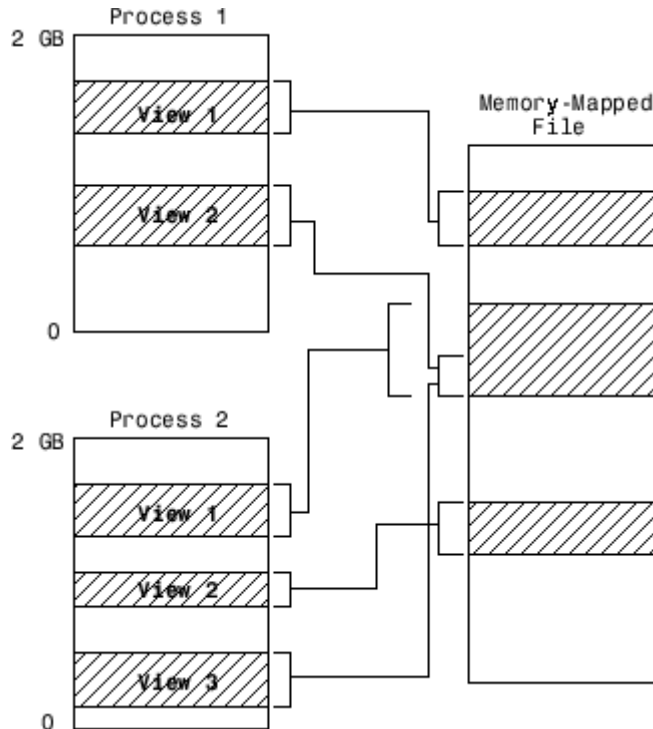
Memory-mapping in your MATLAB application enables you to access file data using standard MATLAB indexing operations. Once you have mapped a file to memory, you can read the contents of that file using the same type of MATLAB statements used to read variables from the MATLAB workspace. The contents of 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. Therefore, you do not need 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.

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.

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.

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 Torvalds' 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.

Map File to Memory

In this section...

“Create a Simple Memory Map” on page 9-7

“Specify Format of Your Mapped Data” on page 9-8

“Map Multiple Data Types and Arrays” on page 9-9

“Select File to Map” on page 9-11

Create a Simple Memory Map

Suppose you want to create a memory map for a file named `records.dat`, using the `memmapfile` function.

Create a sample file named `records.dat`, containing 5000 values.

```
myData = gallery('uniformdata', [5000,1], 0);
```

```
fileID = fopen('records.dat','w');
fwrite(fileID, myData, 'double');
fclose(fileID);
```

Next, create the memory map. Use the `Format` name-value pair argument to specify that the values are of type `double`. Use the `Writable` name-value pair argument to allow write access to the mapped region.

```
m = memmapfile('records.dat', ...
    'Format', 'double', ...
    'Writable', true)

m =

    Filename: 'd:\matlab\records.dat'
    Writable: true
    Offset: 0
    Format: 'double'
    Repeat: Inf
    Data: 5000x1 double array
```

MATLAB creates a `memmapfile` object, `m`. The `Format` property indicates that read and write operations to the mapped region treat the data in the file as a sequence of double-precision numbers. The `Data` property contains the 5000 values from the file, `records.dat`. You can change the value of any of the properties, except for `Data`, after you create the memory map, `m`.

For example, change the starting position of the memory map, `m`. Begin the mapped region 1024 bytes from the start of the file by changing the value of the `Offset` property.

```
m.Offset = 1024
```

```
m =
```

```
Filename: 'd:\matlab\records.dat'  
Writable: true  
Offset: 1024  
Format: 'double'  
Repeat: Inf  
Data: 4872x1 double array
```

Whenever you change the value of a memory map property, MATLAB remaps the file to memory. The `Data` property now contains only 4872 values.

Specify Format of Your Mapped Data

By default, MATLAB considers all the data in a mapped file to be a sequence of unsigned 8-bit integers. However, your data might be of a different data type. When you call the `memmapfile` function, use the `Format` name-value pair argument to indicate another data type. The value of `Format` 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.

Suppose you map a file that is 12 kilobytes in length. Data read from this file can 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 few possibilities. You also could read this data 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 can determine how MATLAB will interpret the mapped data by setting the `Format` property of the memory map when you call the `memmapfile` function.

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.

Map Multiple Data Types and Arrays

If the region you are mapping comprises segments of varying data types or array shapes, you can specify an individual format for each segment. Specify the value of the `Format` name-value pair argument as an `n`-by-3 cell array, where `n` is the number of segments. Each row in the cell array corresponds to a segment. The first cell in the row contains a string identifying the data type to apply to the mapped segment. The second cell contains the array dimensions to apply to the segment. The third cell contains a string specifying the field name for referencing that segment. For a memory map, `m`, use the following syntax:

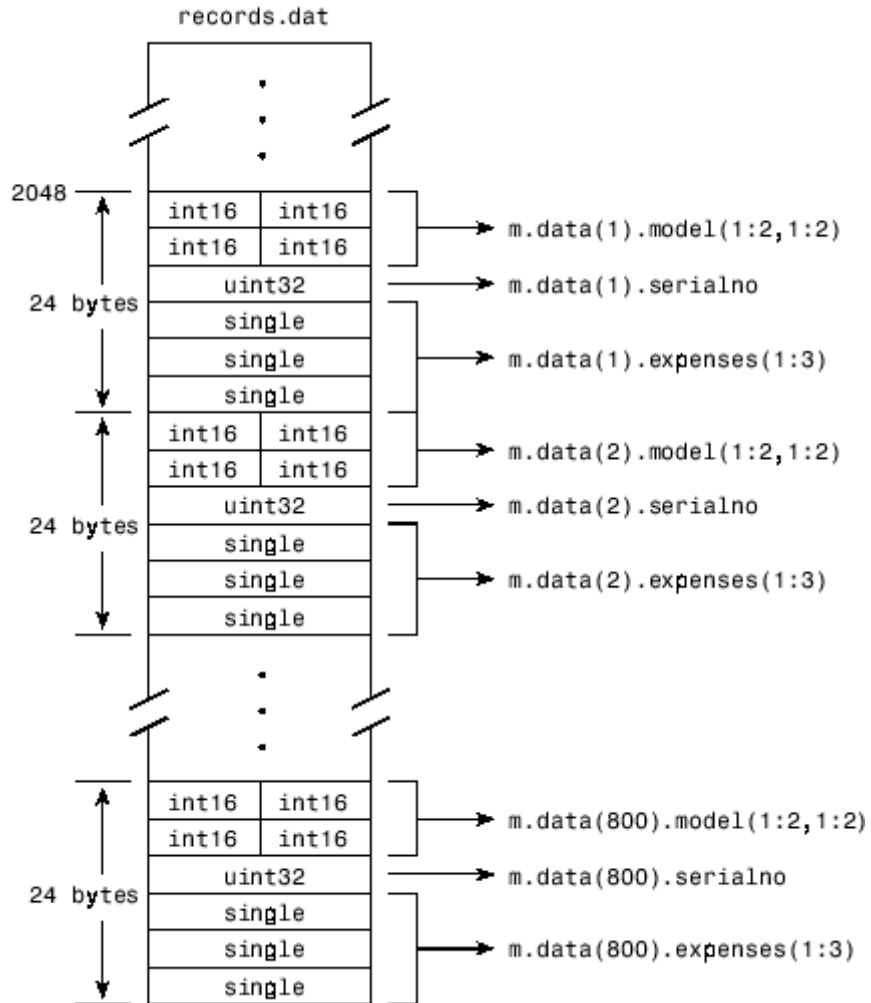
```
m = memmapfile(filename, ...
               'Format', { ...
                       datatype1, dimensions1, fieldname1; ...
                       datatype2, dimensions2, fieldname2; ...
                       :           :           :           ...
                       datatypeN, dimensionsN, fieldnameN})
```

Suppose you have a file that is 40,000 bytes in length. The following code maps the data beginning at the 2048th byte. The `Format` value is a 3-by-3 cell array that maps the file data to three different classes: `int16`, `uint32`, and `single`.

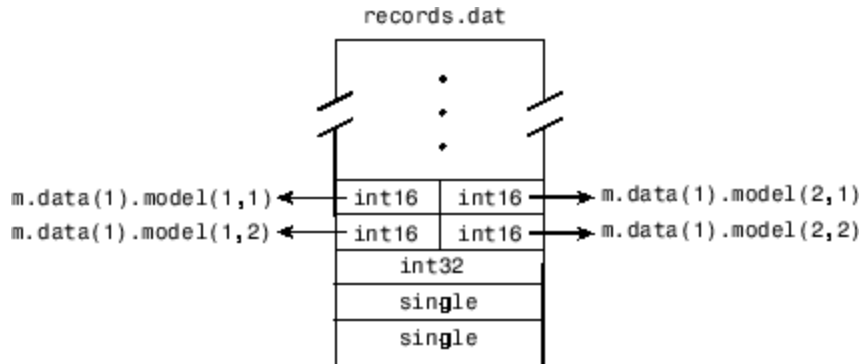
```
m = memmapfile('records.dat', ...
               'Offset', 2048, ...
               'Format', { ...
                       'int16' [2 2] 'model'; ...
                       'uint32' [1 1] 'serialno'; ...
                       'single' [1 3] 'expenses'});
```

In this case, `memmapfile` maps the `int16` data as a 2-by-2 matrix that you can access using the field name, `model`. The `uint32` data is a scalar value accessed using the field name, `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`.

This figure shows the mapping of the example file.



The next figure 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.

Select File to Map

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.

Update the path in the `Filename` property for a memory map using dot notation. For example, to specify a new path, `f:\testfiles\records.dat` for a memory map, `m`, type:

```
m.Filename = 'f:\testfiles\records.dat'
```

See Also

`memmapfile`

Concepts

- “Read Mapped File” on page 9-12
- “Write to Mapped File” on page 9-19

Read Mapped File

This example shows how to create two different memory maps, and then read from each of the maps using the appropriate syntax. Then, it shows how to modify map properties and analyze your data.

You can read the contents of a file that you mapped to memory using the same MATLAB® commands you use to read variables from the MATLAB workspace. By accessing the `Data` property of the memory map, the contents of the mapped file appear as an array in the currently active workspace. To read the data you want from the file, simply index into the array. For better performance, copy the `Data` field to a variable, and then read the mapped file using this variable:

```
dataRef = m.Data;
for k = 1 : N
    y(k) = dataRef(k);
end
```

By contrast, reading directly from the `memmapfile` object is slower:

```
for k = 1 : N
    y(k) = m.Data(k);
end
```

Read from Memory Mapped as Numeric Array

First, create a sample data file named `records.dat` that contains a 5000-by-1 matrix of double-precision floating-point numbers.

```
randData = gallery('uniformdata',[5000,1],0);

fileID = fopen('records.dat','w');
fwrite(fileID,randData,'double');
fclose(fileID);
```

Map 100 double-precision floating-point numbers from the file to memory, and then read a portion of the mapped data. Create the memory map, `m`. Specify an `Offset` value of 1024 to begin the map 1024 bytes from the start of the file. Specify a `Repeat` value of 100 to map 100 values.

```
m = memmapfile('records.dat','Format','double', ...
              'Offset',1024,'Repeat',100);
```

Copy the Data property to a variable, d. Then, show the format of d.

```
d = m.Data;
```

```
whos d
```

Name	Size	Bytes	Class	Attributes
d	100x1	800	double	

The mapped data is an 800-byte array because there are 100 double values, each requiring 8 bytes.

Read a selected set of numbers from the file by indexing into the vector, d.

```
d(15:20)
```

```
ans =
```

```
0.8392
0.6288
0.1338
0.2071
0.6072
0.6299
```

Read from Memory Mapped as Nonscalar Structure

Map portions of data in the file, records.dat, as a sequence of multiple data types.

Call the memmapfile function to create a memory map, m.

```
m = memmapfile('records.dat', ...
```

```
'Format', {  
    'uint16' [5 8] 'x'; ...  
    'double' [4 5] 'y' });
```

The `Format` parameter tells `memmapfile` to treat the first 80 bytes of the file as a 5-by-8 matrix of `uint16` values, and the 160 bytes after that as a 4-by-5 matrix of `double` values. This pattern repeats until the end of the file is reached.

Copy the `Data` property to a variable, `d`.

```
d = m.Data
```

```
d =
```

```
166x1 struct array with fields:
```

```
    x  
    y
```

`d` is a 166-element structure array with two fields. `d` is a nonscalar structure array because the file is mapped as a repeating sequence of multiple data types.

Examine one structure in the array to show the format of each field.

```
d(3)
```

```
ans =
```

```
    x: [5x8 uint16]  
    y: [4x5 double]
```

Read the `x` field of that structure from the file.

```
d(3).x
```



```

ans =

    19972    47529    19145    16356    46507    47978    35550    16341
    60686    51944    16362    58647    35418    58072    16338    62509
    51075    16364    54226    34395     8341    16341    33787    57669
    16351    35598     6686    11480    16357    28709    36239     5932
    44292    15577    41755    16362    30311    31712    54813    16353

```

MATLAB formats the block of data as a 5-by-8 matrix of `uint16` values, as specified by the `Format` property.

Read the `y` field of that structure from the file.

```
d(3).y
```

```

ans =

    0.7271    0.3704    0.6946    0.5226    0.2714
    0.3093    0.7027    0.6213    0.8801    0.2523
    0.8385    0.5466    0.7948    0.1730    0.8757
    0.5681    0.4449    0.9568    0.9797    0.7373

```

MATLAB formats the block of data as a 4-by-5 matrix of `double` values.

Modify Map Properties and Analyze Data

This part of the example shows how to plot the Fourier transform of data read from a file via a memory map. It then modifies several properties of the existing map, reads from a different part of the data file, and plots a histogram from that data.

Create a sample file named `double.dat`.

```

randData = gallery('uniformdata',[5000,1],0);
fileID = fopen('double.dat','w');

```

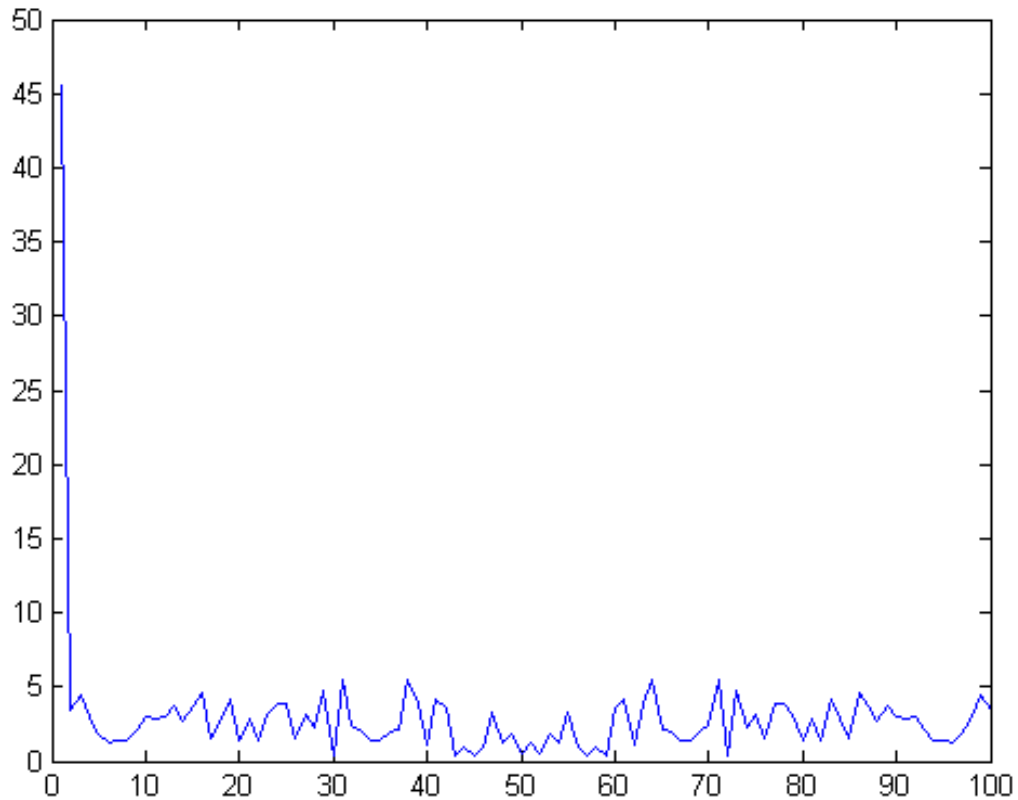
```
fwrite(fileID,randData,'double');  
fclose(fileID);
```

Create a `memmapfile` object of 1,000 elements of type `double`, starting at the 1025th byte.

```
m = memmapfile('double.dat','Offset',1024, ...  
              'Format','double','Repeat',1000);
```

Copy the `Data` property to a variable, `k`. Then, get data associated with the map and plot the FFT of the first 100 values of the map.

```
k = m.Data;  
plot(abs(fft(k(1:100))))
```



This is the first time that data is referenced and is when the actual mapping of the file to the MATLAB address space takes place.

Change the map properties, but continue using the same file. Whenever you change the value of a memory map property, MATLAB remaps the file to memory.

```
m.Offset = 4096;  
m.Format = 'single';  
m.Repeat = 800;
```

`m` is now a `memmapfile` object of 800 elements of type `single`. The map now begins at the 4096th byte in the file, `records.dat`.

Read from the portion of the file that begins at the 4096th byte, and calculate the maximum value of the data. This command maps a new region and unmaps the previous region.

```
X = max(m.Data)
```

```
X =
```

```
7.5449e+37
```

See Also

`memmapfile`

Concepts

- “Map File to Memory” on page 9-7
- “Write to Mapped File” on page 9-19

Write to Mapped File

This example shows how to create three different memory maps, and then write to each of the maps using the appropriate syntax. Then, it shows how to work with copies of your mapped data.

You can write to a file using the same MATLAB commands you use to access variables in the MATLAB workspace. By accessing the `Data` property of the memory map, the contents of the mapped file appear as an array in the currently active workspace. Simply index into this array to write data to the file. The syntax to use when writing to mapped memory depends on the format of the `Data` property of the memory map.

In this section...

“Write to Memory Mapped as Numeric Array” on page 9-19

“Write to Memory Mapped as Scalar Structure” on page 9-21

“Write to Memory Mapped as Nonscalar Structure” on page 9-21

“Syntaxes for Writing to Mapped File” on page 9-23

“Work with Copies of Your Mapped Data” on page 9-24

Write to Memory Mapped as Numeric Array

First, create a sample file named `records.dat`, in your current folder.

```
myData = gallery('uniformdata', [5000,1], 0);
```

```
fileID = fopen('records.dat','w');  
fwrite(fileID, myData, 'double');  
fclose(fileID);
```

Map the file as a sequence of 16-bit-unsigned integers. Use the `Format` name-value pair argument to specify that the values are of type `uint16`.

```
m = memmapfile('records.dat', ...  
    'Offset',20, ...  
    'Format','uint16', ...  
    'Repeat',15);
```

Because the file is mapped as a sequence of a single class (`uint16`), `Data` is a numeric array.

Ensure that you have write permission to the mapped file. Set the `Writable` property of the memory map, `m`, to `true`.

```
m.Writable = true;
```

Create a matrix `X` that is the same size as the `Data` property, and write it to the mapped part of the file. 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.

```
X = uint16(1:1:15);  
m.Data = X;
```

`X` is a 1-by-15 vector of integer values ranging from 1 to 15.

Verify that new values were written to the file. Specify an `Offset` value of 0 to begin reading from the beginning of the file. Specify a `Repeat` value of 35 to view a total of 35 values. Use the `reshape` function to display the values as a 7-by-5 matrix.

```
m.Offset = 0;  
m.Repeat = 35;  
reshape(m.Data,5,7)'
```

```
ans =
```

```
47662  34773  26485  16366  58664  
25170  38386  16333  14934   9028  
     1     2     3     4     5  
     6     7     8     9    10  
    11    12    13    14    15  
10085  14020  16349  37120  31342  
62110  16274   9357  44395  18679
```

The values in `X` have been written to the file, `records.dat`.

Write to Memory Mapped as Scalar Structure

Map a region of the file, `records.dat`, as a 300-by-8 matrix of type `uint16` that can be referenced by the field name, `x`, followed by a 200-by-5 matrix of type `double` that can be referenced by the field name, `y`. Specify write permission to the mapped file using the `Writable` name-value pair argument.

```
m = memmapfile('records.dat', ...
    'Format', {
        'uint16' [300 8] 'x'; ...
        'double' [200 5] 'y' }, ...
    'Repeat', 1, 'Writable', true);
```

View the `Data` property

```
m.Data
```

```
ans =
    x: [300x8 uint16]
    y: [200x5 double]
```

`Data` is a scalar structure array. This is because the file, `records.dat`, is mapped as containing multiple data types that do not repeat.

Replace the matrix in the field, `x`, with a matrix of all ones.

```
m.Data.x = ones(300,8,'uint16');
```

Write to Memory Mapped as Nonscalar Structure

Map the file, `records.dat`, as a 25-by-8 matrix of type `uint16` followed by a 15-by-5 matrix of type `double`. Repeat the pattern 20 times.

```
m = memmapfile('records.dat', ...
    'Format', {
        'uint16' [5 4] 'x'; ...
```

```
'double' [15 5] 'y' }, ...  
'Repeat', 20, 'Writable', true);
```

View the Data property

```
m.Data
```

```
ans =
```

```
20x1 struct array with fields:
```

```
  x  
  y
```

Data is a nonscalar structure array, because the file is mapped as a repeating sequence of multiple data types.

Write an array of all ones to the field named x in the 12th element of Data.

```
m.Data(12).x = ones(5,4,'uint16');
```

For the 12th element of Data, write the value, 50, to all elements in rows 3 to 5 of the field, x.

```
m.Data(12).x(3:5,1:end) = 50;
```

View the field, x, of the 12th element of Data.

```
m.Data(12).x
```

```
ans =
```

```
  1     1     1     1  
  1     1     1     1  
 50    50    50    50  
 50    50    50    50  
 50    50    50    50
```


Syntaxes for Writing to Mapped File

The syntax to use when writing to mapped memory depends on the format of the `Data` property of the memory map. View the properties of the memory map by typing the name of the `memmapfile` object.

This table shows the syntaxes for writing a matrix, `X`, to a memory map, `m`.

Format of the Data Property	Syntax for Writing to Mapped File
Numeric array Example: 15x1 uint16 array	<code>m.Data = X;</code>
Scalar (1-by-1) structure array Example: 1x1 struct array with fields: x y	<code>m.Data.fieldname = X;</code> <i>fieldname</i> is the name of a field.
Nonscalar (n-by-1) structure array Example: 20x1 struct array with fields: x y	<code>m.Data(k).fieldname = X;</code> k is a scalar index and <i>fieldname</i> is the name of a field.

The class of `X` and the number of elements in `X` must match those of the `Data` property or the field of the `Data` property being accessed. You cannot change the dimensions of the `Data` property after you have created the memory map using the `memmapfile` function. For example, you cannot diminish or expand the size of an array by removing or adding a row from the mapped array, `m.Data`.

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 memory map, `m`, you must either modify the `Format` or `Repeat` properties for `m`, or recreate `m` using the `memmapfile` function.

Note To successfully modify a mapped file, you must have write permission for that file. If you do not have write permission, attempting to write to the file generates an error, even if the `Writable` property is `true`.

Work with Copies of Your Mapped Data

This part of the example shows how to work with copies of your mapped data. 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.

Create a sample file named `double.dat`.

```
myData = gallery('uniformdata',[5000,1],0) * 100;
fileID = fopen('double.dat','w');
fwrite(fileID,myData,'double');
fclose(fileID);
```

Map the file as a series of double matrices.

```
m = memmapfile('double.dat', ...
    'Format', { ...
        'double' [5 5] 'x'; ...
        'double' [4 5] 'y' });
```

View the values in `m.Data(2).x`.

```
m.Data(2).x
```

```
ans =
```

```
50.2813    19.3431    69.7898    49.6552    66.0228
```

```
70.9471  68.2223  37.8373  89.9769  34.1971
42.8892  30.2764  86.0012  82.1629  28.9726
30.4617  54.1674  85.3655  64.4910  34.1194
18.9654  15.0873  59.3563  81.7974  53.4079
```

Copy the contents of `m.Data` to the variable, `d`.

```
d = m.Data;
```

Write all zeros to the field named `x` in the copy.

```
d(2).x(1:5,1:5) = 0;
```

Verify that zeros are written to `d(2).x`

```
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
```

Verify that the data in the mapped file is not changed.

```
m.Data(2).x
```

```
ans =
```

```
50.2813  19.3431  69.7898  49.6552  66.0228
70.9471  68.2223  37.8373  89.9769  34.1971
42.8892  30.2764  86.0012  82.1629  28.9726
30.4617  54.1674  85.3655  64.4910  34.1194
18.9654  15.0873  59.3563  81.7974  53.4079
```

See Also

memmapfile

Concepts

- “Map File to Memory” on page 9-7
- “Read Mapped File” on page 9-12

Delete Memory Map

In this section...
“Ways to Delete a Memory Map” on page 9-27
“The Effect of Shared Data Copies On Performance” on page 9-27

Ways to Delete a Memory Map

To clear a `memmapfile` object from memory, do any of the following:

- Reassign another value to the `memmapfile` object’s variable
- Clear the `memmapfile` object’s variable from memory
- Exit the function scope in which the `memmapfile` 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 because no memory actually gets copied. In the following statement, `d` is a shared data copy of the data mapped from the file:

```
d = m.Data;
```

When you finish using the mapped data, make sure to clear any variables that share data with the mapped file before clearing the `memmapfile` 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 cleared. If access to the mapped file was over a network, then copying this data to local memory can take considerable time. Therefore, if you assign `m.Data` to the variable, `d`, you should be sure to clear `d` before clearing `m` when you are finished with the memory map.

Share Memory Between Applications

This example shows how to implement two separate MATLAB processes that 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.

Prior to running the example, copy the `send` and `answer` functions to files `send.m` and `answer.m` in your current working directory.

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.

```
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
```

```
% 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

    % Warn if the message is longer than 255 characters.
    if len > 255
        warning('ml:ml', 'SEND input will be truncated to 255 characters. ');
    end
    str = str(1:min(len,255)); % Limit message to 255 characters.
    len = length(str); % Update len if str has been truncated.

    % 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
```

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 until the 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+double(m.Data(1)))));

    % Display the message.
```



```

disp('Received message from SEND:')
disp(msg)

% Transform the message to all uppercase.
m.Data(2:1+double(m.Data(1))) = upper(msg);

% Signal to SEND that the response is ready.
m.Data(1) = 0;
end

```

Running the Example

To see what the example looks like when it is run, first, start two separate MATLAB sessions on the same computer system. Call the `send` function with no inputs in one MATLAB session. Call the `answer` function in the other session, 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?

Enter a second message at the prompt display by the **send** function. **send** writes the second message to the file.

I received your reply.

response from ANSWER is:
I RECEIVED YOUR REPLY.
Enter send string (or RETURN to end):

answer reads the second message, put it into uppercase, and then writes the message to the file.

Received message from SEND:
I received your reply.

In the first instance of MATLAB, press **Enter** to exit the example.

Terminating SEND function.

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 10-2
- “Sending Email” on page 10-4
- “Performing FTP File Operations” on page 10-7
- “Display Hyperlinks in the Command Window” on page 10-9

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, <http://www.mathworks.com/matlabcentral/fileexchange/>. 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`.

Sending Email

To send an email from MATLAB, use the `sendmail` function. You can also attach files to an email, which lets you mail files directly from MATLAB. To use `sendmail`, you must first set up your email address and your SMTP server information with the `setpref` function.

The `setpref` function defines two mail-related preferences:

- Email address: This preference sets your email 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 email 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 email account settings in your email client application. You can also contact your system administrator for the information.

Note The `sendmail` function does not support email 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 email address and the email subject:

```
sendmail('recipient@someserver.com', 'Hello From MATLAB!');
```

You can supply multiple email 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 email 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 email 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 email 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 email 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 email (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)
```

Display Hyperlinks in the Command Window

In this section...
“Creating Hyperlinks to Web Pages” on page 10-9
“Transferring Files Using FTP” on page 10-9

Creating Hyperlinks to Web Pages

When creating a hyperlink to a Web page, append a full hypertext string on a single line as input to the `disp` or `fprintf` command. For example, the following command:

```
disp('<a href = "http://www.mathworks.com">The MathWorks Web Site</a>')
```

displays the following hyperlink in the Command Window:

The MathWorks Web Site

When you click this hyperlink, a MATLAB Web browser opens and displays the requested page.

Transferring Files Using FTP

To create a link to an FTP site, enter the site address as input to the `disp` command as follows:

```
disp('<a href = "ftp://ftp.mathworks.com">The MathWorks FTP Site</a>')
```

This command displays the following as a link in the Command Window:

The MathWorks FTP Site

When you click the link, a MATLAB browser opens and displays the requested FTP site.

Install and Use Raspberry Pi Hardware

- “Install Support for Raspberry Pi Hardware” on page 11-3
- “Guidelines for Entering Static IP Settings” on page 11-17
- “Open Interactive Examples” on page 11-18
- “Connecting to Raspberry Pi Hardware” on page 11-20
- “Connect to Raspberry Pi Hardware” on page 11-22
- “Troubleshoot Connecting to Raspberry Pi Hardware” on page 11-25
- “Get the IP Address of the Raspberry Pi Hardware” on page 11-27
- “The Raspberry Pi LED” on page 11-29
- “Turn the Raspberry Pi LED On and Off” on page 11-31
- “Flash the Raspberry Pi LED in Response to an Input” on page 11-34
- “The Raspberry Pi GPIO Pins” on page 11-35
- “Use the Raspberry Pi GPIO Pins as Digital Inputs and Outputs” on page 11-36
- “Troubleshoot Raspberry Pi GPIO Pins” on page 11-40
- “The Raspberry Pi Serial Port” on page 11-42
- “Use the Raspberry Pi Serial Port to Connect to a Device” on page 11-43
- “Troubleshoot the Raspberry Pi Serial Port” on page 11-48
- “The Raspberry Pi I2C Interface” on page 11-49
- “Use the Raspberry Pi I2C Interface to Connect to a Device” on page 11-50

- “Troubleshoot the Raspberry Pi I2C Interface” on page 11-54
- “The Raspberry Pi SPI Interface” on page 11-55
- “Use the Raspberry Pi SPI Interface to Connect to a Device” on page 11-57
- “The Raspberry Pi Camera Board” on page 11-61
- “Use the Raspberry Pi Camera Board to Capture Images and Video” on page 11-63
- “Troubleshoot the Raspberry Pi Camera Board” on page 11-65
- “The Raspberry Pi Linux Command Interface” on page 11-66
- “Run Linux Commands on Raspberry Pi Hardware” on page 11-67
- “Troubleshoot Running Linux Commands on Raspberry Pi Hardware” on page 11-70
- “Management of Raspberry Pi Files” on page 11-71
- “Manage Raspberry Pi Files” on page 11-72
- “Troubleshoot Managing Raspberry Pi Files” on page 11-73

Install Support for Raspberry Pi Hardware

In this section...
“Install the Support Package” on page 11-3
“Complete Additional Setup Tasks” on page 11-5

Add support for Raspberry Pi™ hardware to the MATLAB product by installing the MATLAB Support Package for Raspberry Pi Hardware.

This process installs the following items on your host computer:

- Third-party software development tools
- MATLAB commands
- Examples

This process also installs a customized version of Raspian Wheezy on the Raspberry Pi hardware.

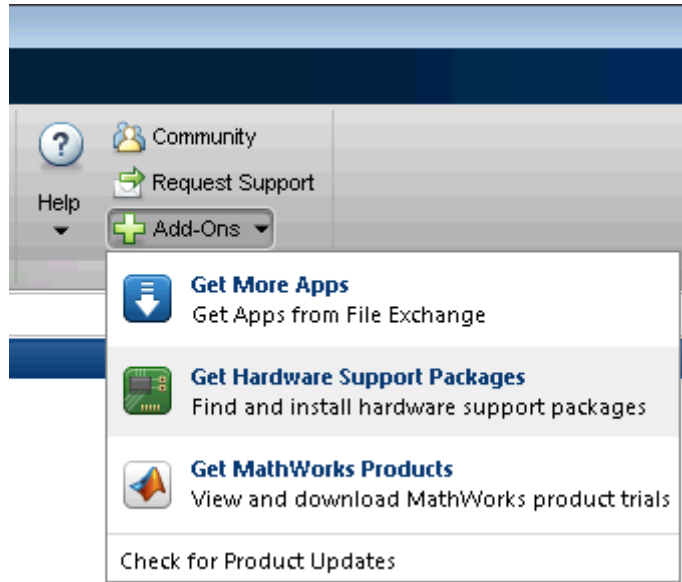
When you complete this installation, you can use MATLAB commands to control, and retrieve data from, Raspberry Pi hardware and peripherals.

The Raspberry Pi hardware is also referred to as a *board* or as *target hardware*.

Install the Support Package

To install the MATLAB Support Package for Raspberry Pi Hardware:

- 1 On the MATLAB Toolstrip, click **Add-Ons > Get Hardware Support Packages**.

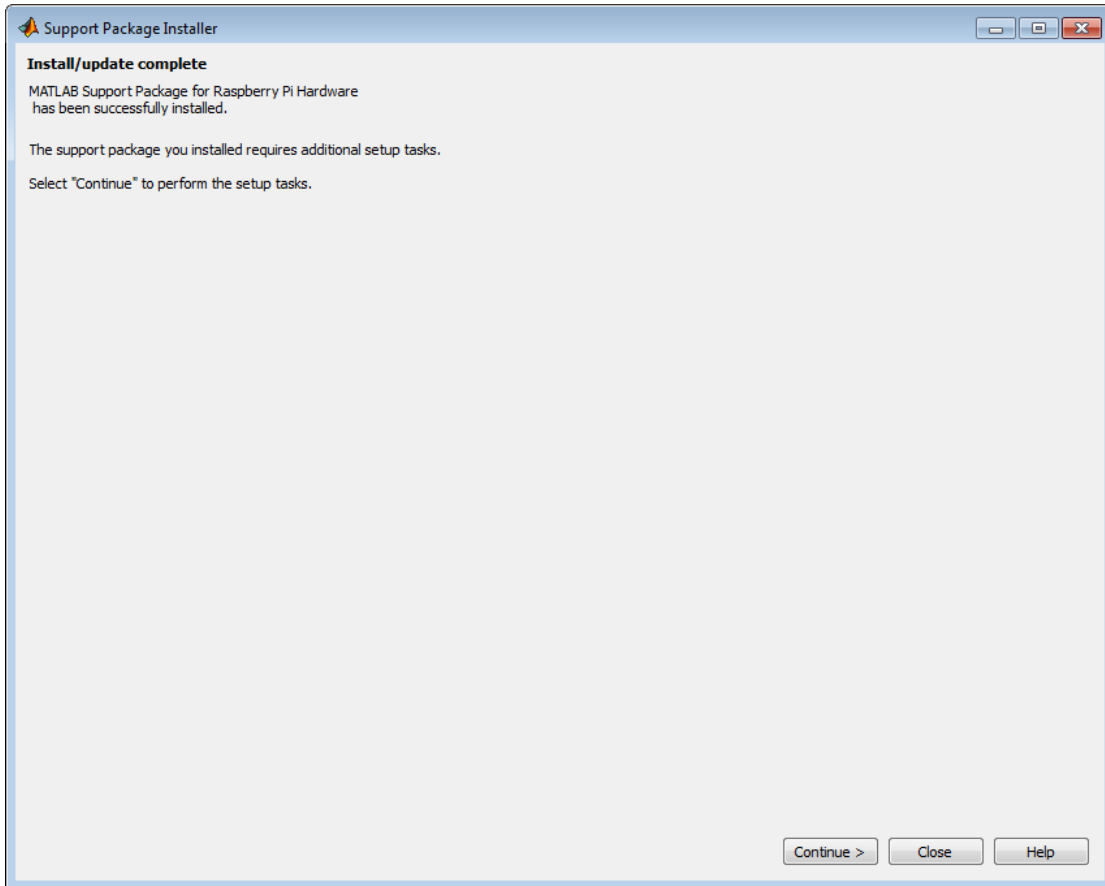


2 In Support Package Installer, follow the instructions on each screen.

For more information about the options on a particular screen, click the **Help** button.

3 At the **Install/update complete** screen, choose whether to perform additional setup tasks:

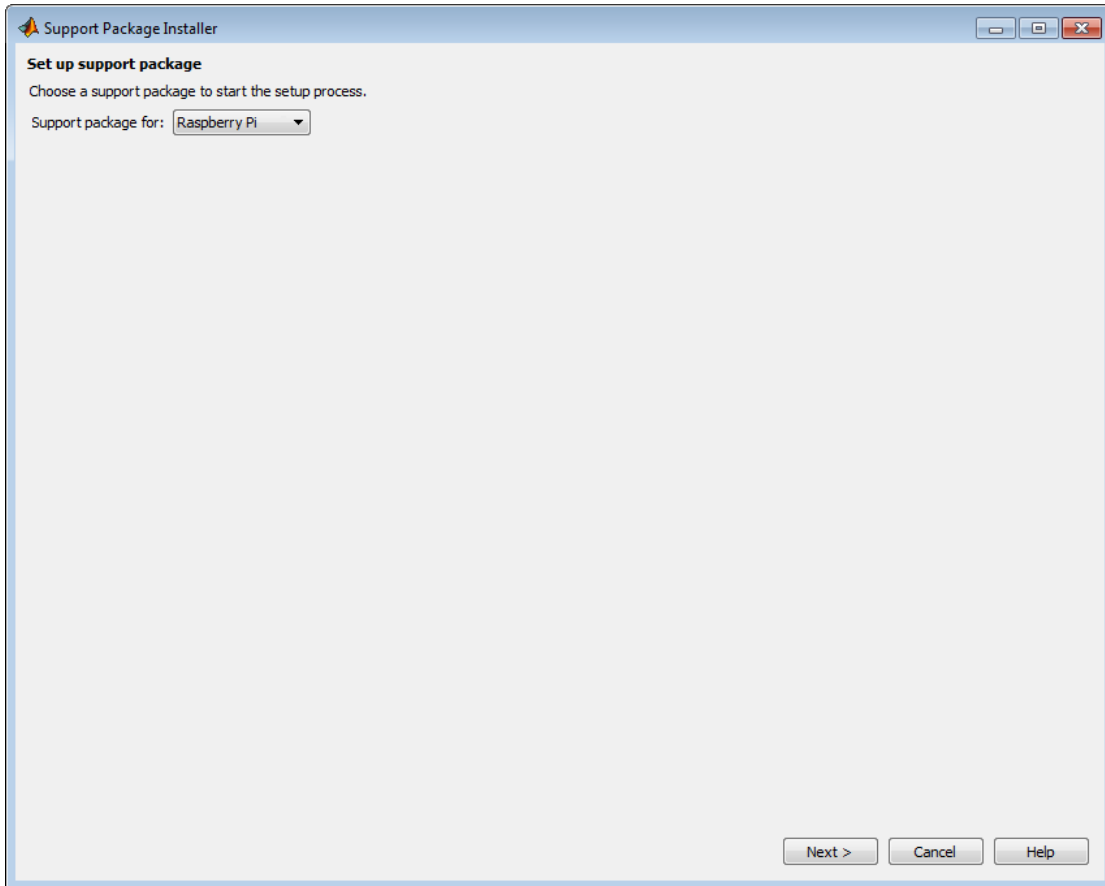
- Click **Continue** to set up the Raspberry Pi board.
- If you have already set up the Raspberry Pi board for this specific support package, click **Close**.



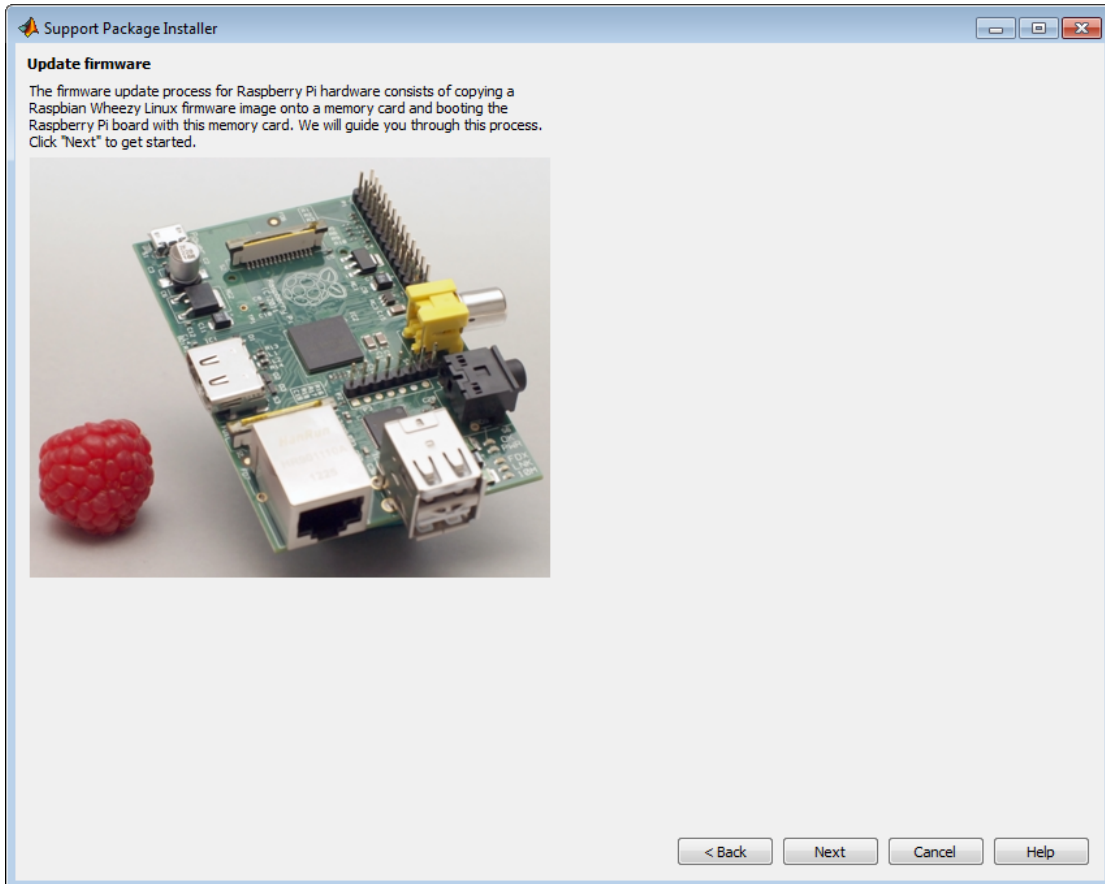
Complete Additional Setup Tasks

Install a customized version of Raspian Wheezy on the Raspberry Pi board:

- 1 If you clicked **Close** on the preceding **Install/update complete** screen, restart Support Package Installer by entering `targetupdater` in the MATLAB Command Window.
- 2 On the **Set up support package** screen, set **Support package for** to **Raspberry Pi (MATLAB)**, and click **Next**.

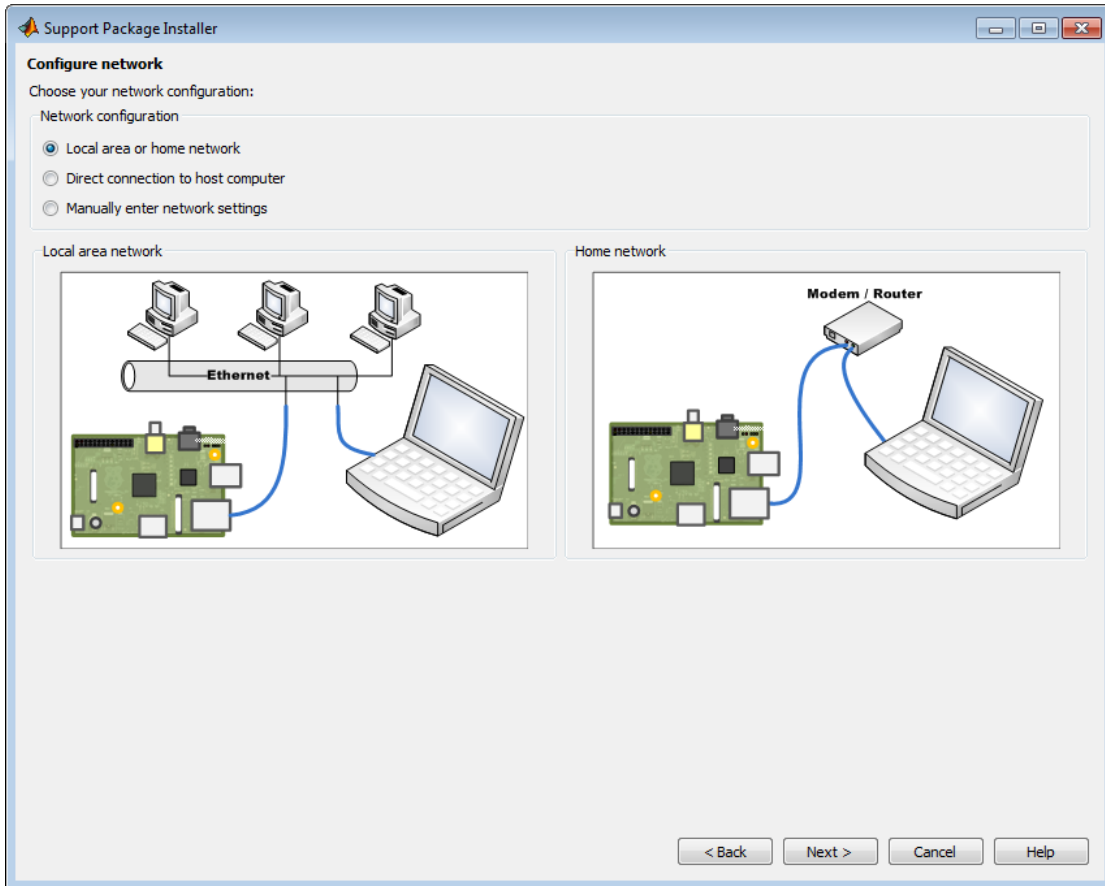


- 3 Review the information, and click **Next**. The firmware download takes several minutes.

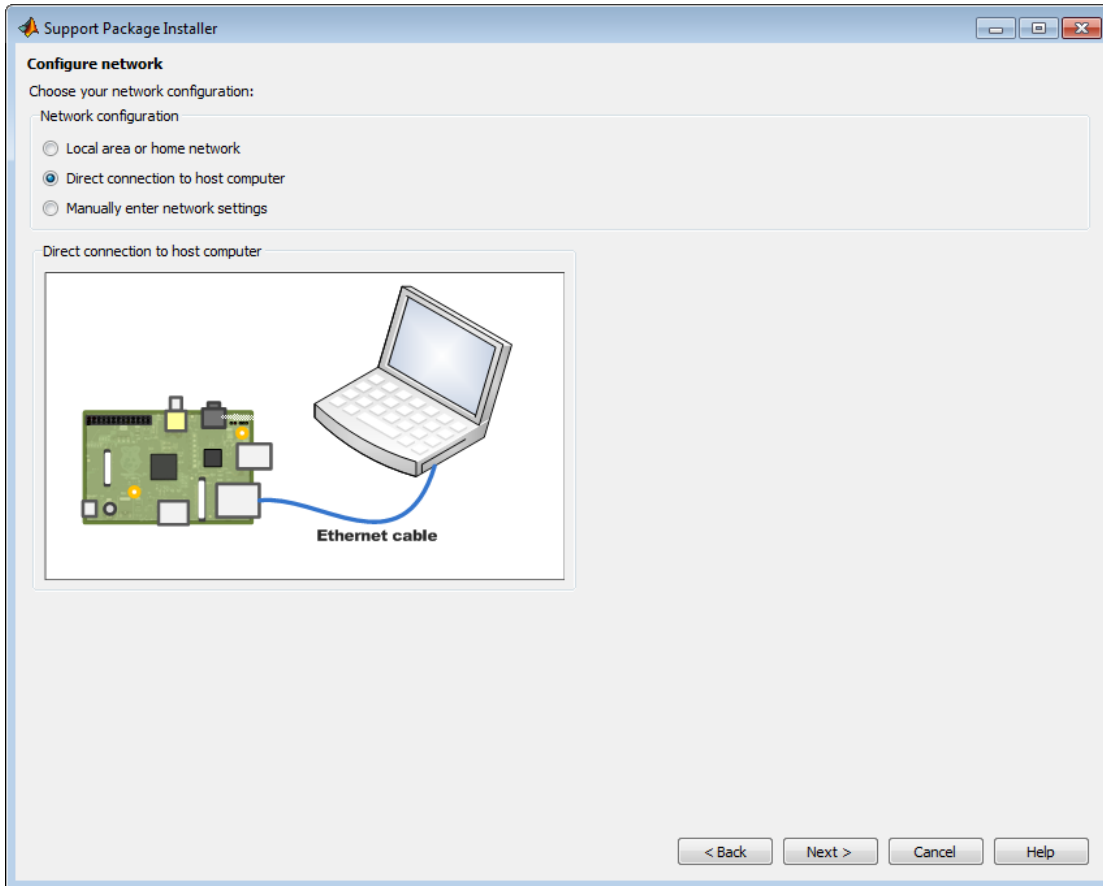


4 Choose your network configuration:

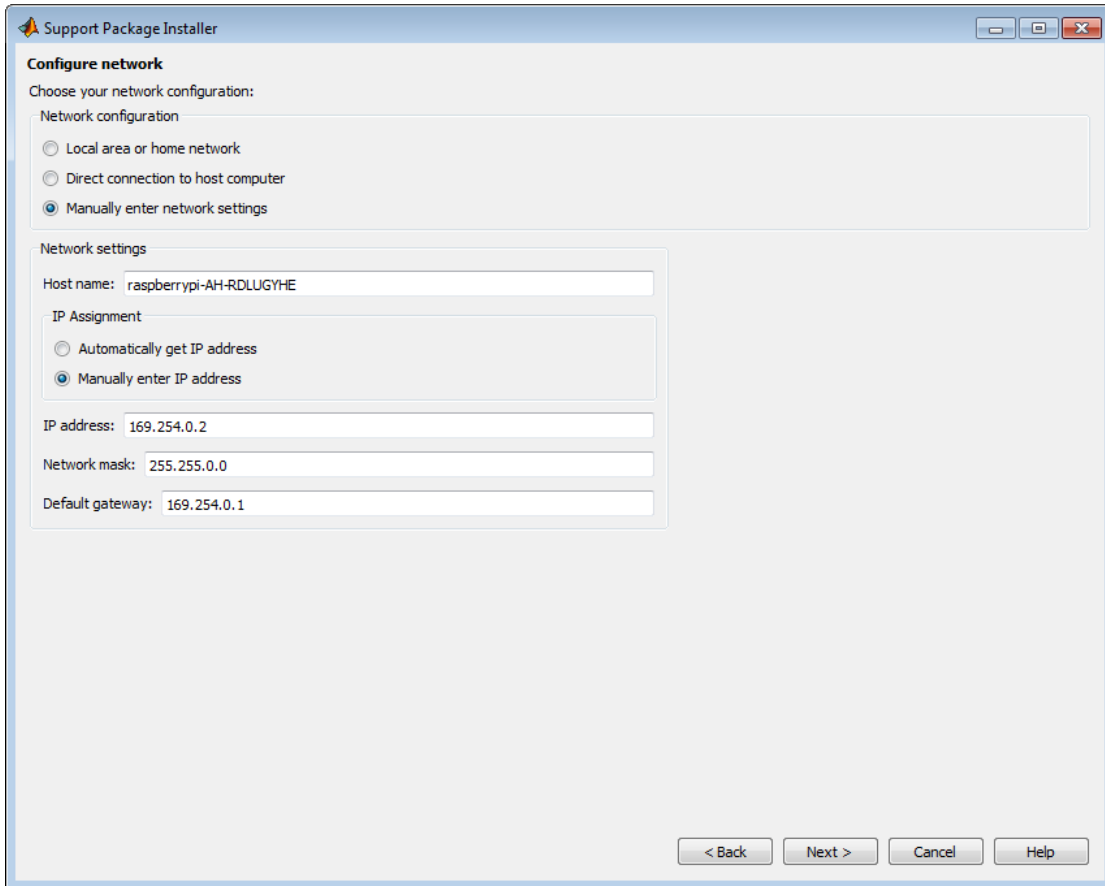
- **Local area or home network** — Confirm that the Raspberry Pi hardware is connected to your host computer using a local area network (LAN) or home network similar to the ones that the installer displays. Then, click **Next**. This option applies dynamic network settings provided by a DNS service on the network.



- **Direct connection to host computer** — Confirm that the Raspberry Pi hardware is connected to your host computer using a direct connection similar to the one that the installer displays. Then, click **Next**. This option applies static network settings based upon the network settings of the host computer.



- **Manually enter network settings** — To manually configure the network settings, select this option.



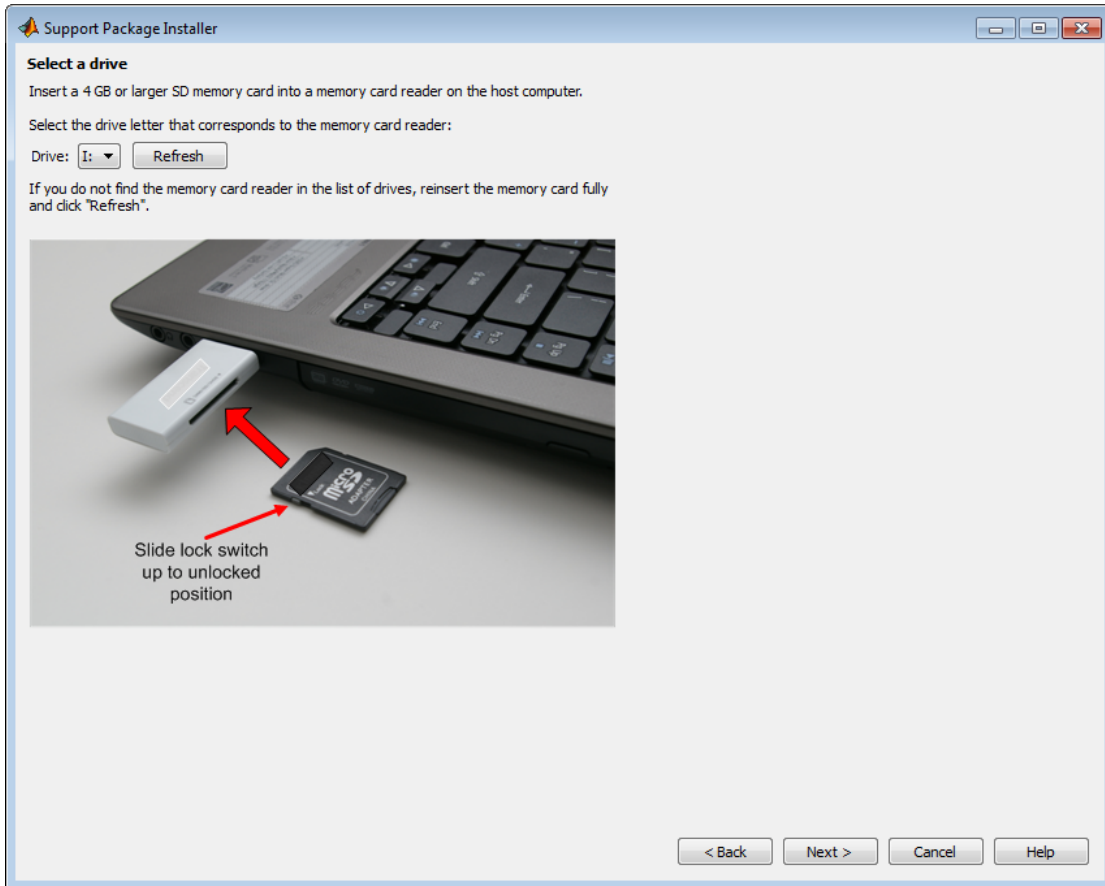
Selecting **Manually enter network settings** displays the following options:

- **Host name** — This parameter displays the host name that is assigned to the Raspberry Pi hardware. If multiple boards are connected to the network, edit the host name to make it unique.
- **Automatically get IP address** — This option applies dynamic network settings provided by a DHCP service on the network.

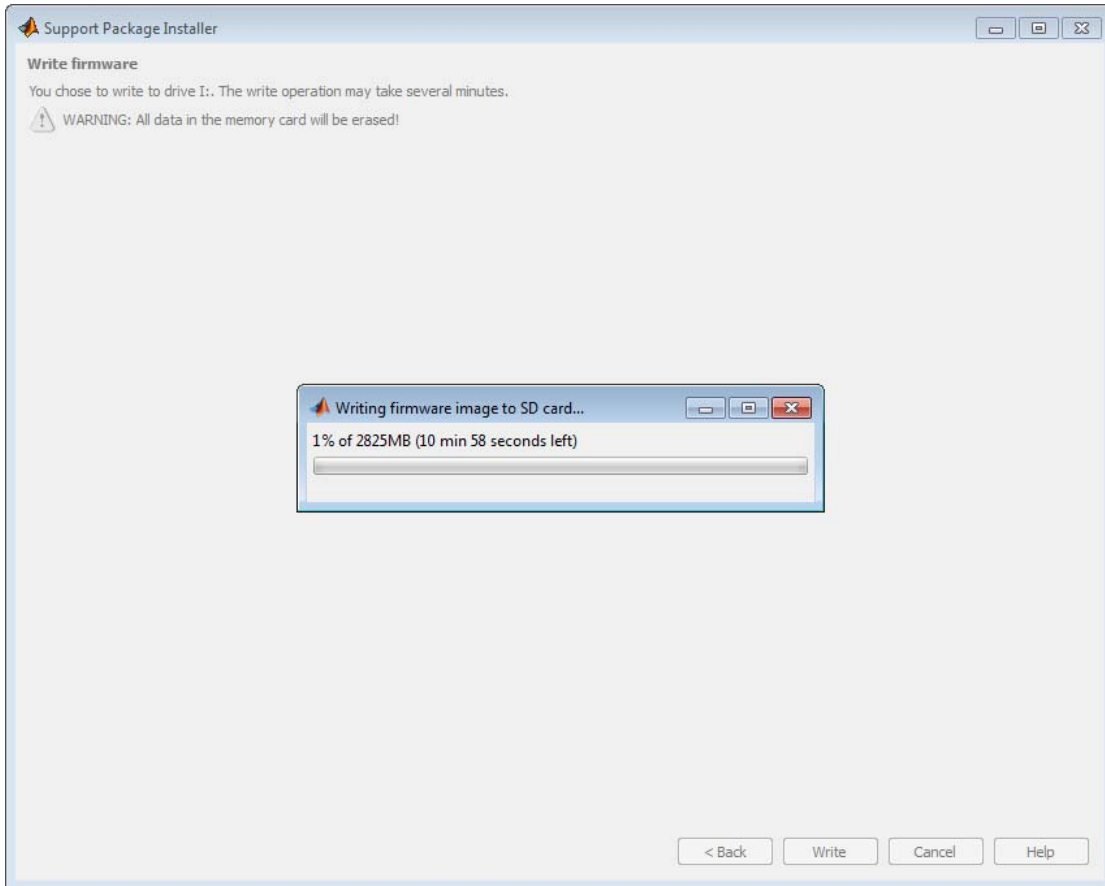
- **Manually enter IP address** — Use this option to edit the **IP address**, **Network mask**, and **Default gateway** settings. Also see “Guidelines for Entering Static IP Settings” on page 11-17.
- 5** Insert the SD card into a media card reader that is attached to your host computer. Your host computer assigns a drive letter to the memory card.

Click **Refresh**, select the drive letter assigned to the SD card, and click **Next**.

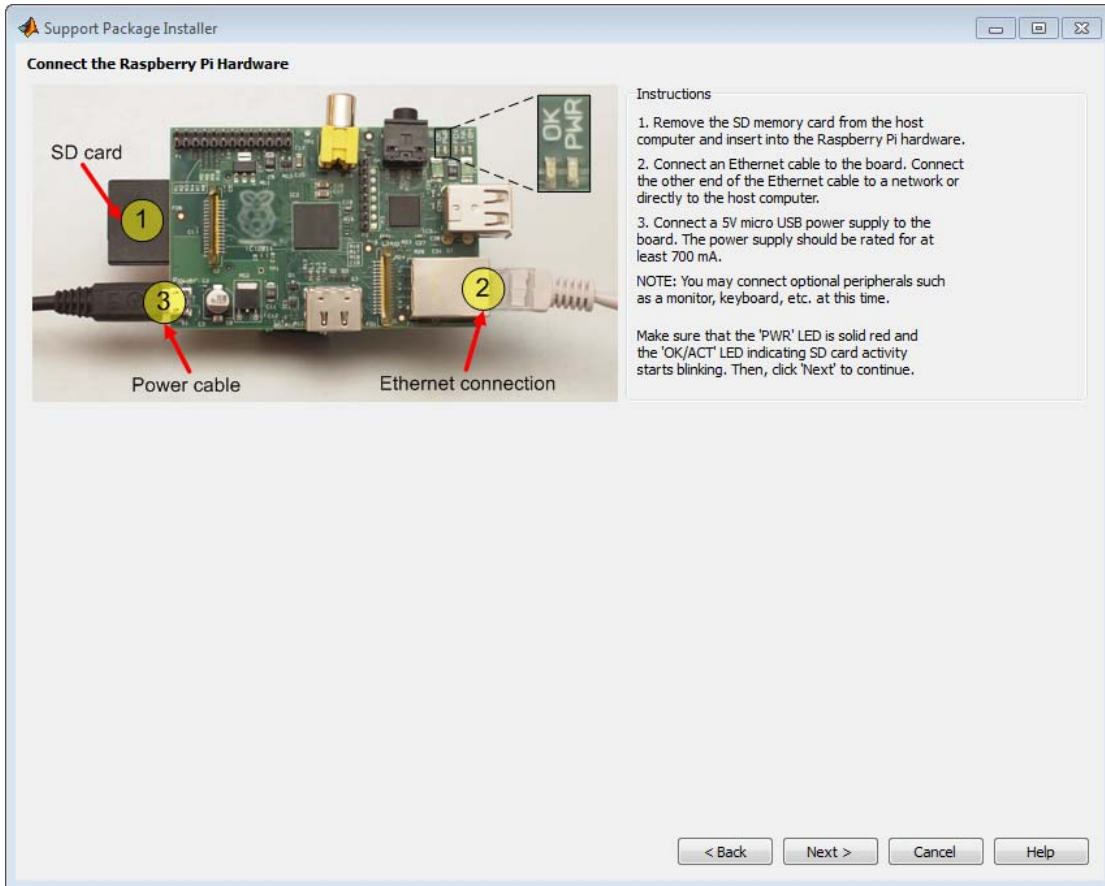
If multiple drive letters are available, identify the drive letter of the media card reader. Open the Windows Start menu, choose **Computer**, and review the list of **Devices with Removable Storage**



- 6 Click **Write**. The installer overwrites all previous data on the memory card with the firmware. This process takes several minutes to complete.



- 7 Make the connections shown in the figure. When the PWR LED is solid red, and the ACT or OK LED stops blinking, click **Next**.



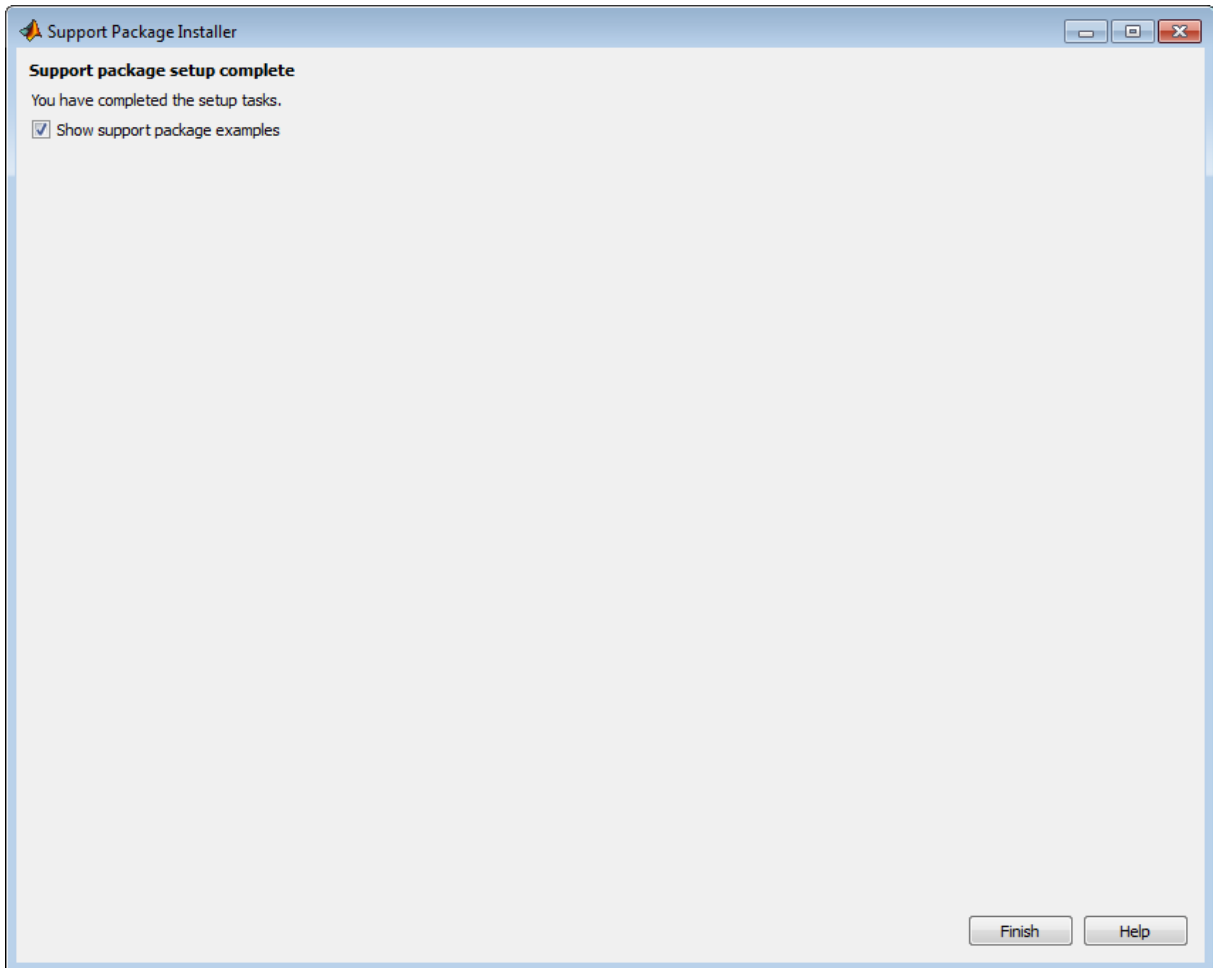
- 8 Keep a record of the IP address, host name, user name, and password. Then, click **Test Connection**.

The installer uses SSH to create a test connection to the board using the host name, user name, and password shown.

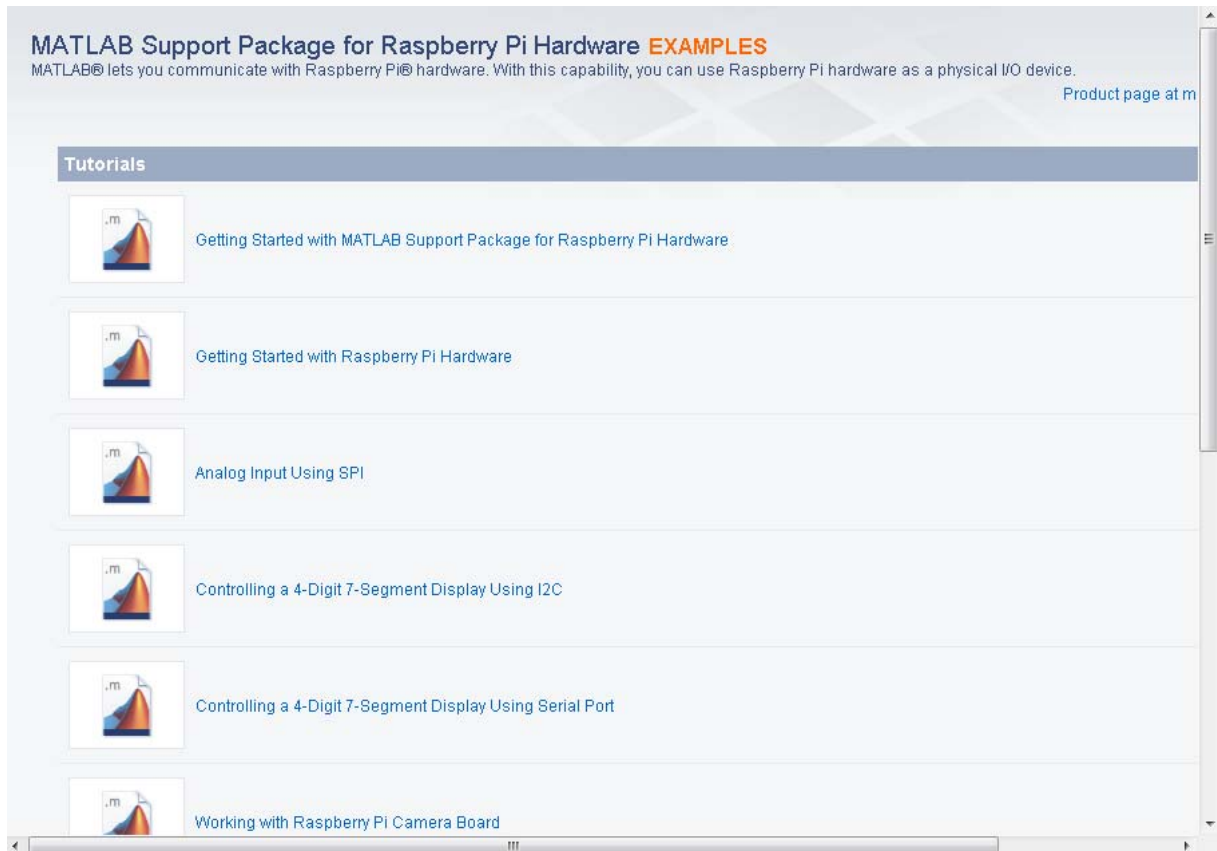
- If the test fails, click **Back** or use the `targetupdater` function to repeat the setup process.
- If the test succeeds, the software displays a "Connection successful" message.

9 Click Finish.

If you leave **Show support package examples** enabled, the installer opens the example page for Raspberry Pi hardware.



10 For experience using Simulink® models with Raspberry Pi hardware, complete the examples.



To reopen these examples later, see “Open Interactive Examples” on page 11-18.

Guidelines for Entering Static IP Settings

- The **IP address** must be unique for each device on the network.
- The **Network mask** must be the same for all devices on the network. This value is also known as **Subnet mask**.
- The **Default gateway** is usually the same for all devices on the network.

Start by entering `ipconfig` at the command line of your host computer. This command displays network settings of the Ethernet adapters on the host computer. Look for the settings of the Ethernet adapter that is connected to the target hardware.

Suppose that the Ethernet adapter connected to the target hardware has the following values:

```
IPv4 Address . . . . . : 192.168.1.2
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 192.168.1.1
```

In that case, for the target hardware:

- Set **IP address** to an unused IP address, between `192.168.1.3` and `192.168.1.254`
- Set **Network mask** to use the same network mask value, `255.255.255.0`

Open Interactive Examples

The following interactive tutorials and application examples come with the MATLAB Support Package for Raspberry Pi Hardware.

Tutorials:

- Getting Started with MATLAB Support Package for Raspberry Pi Hardware
- Getting Started with Raspberry Pi Hardware
- Analog Input Using SPI
- Controlling a 4-Digit 7-Segment Display Using I2C
- Controlling a 4-Digit 7-Segment Display Using Serial Port
- Working with Raspberry Pi Camera Board

Application examples:

- Build a Digital Voltmeter
- Track a Green Ball
- Build a Motion Sensor Camera
- Add Digital I/O Pins to Raspberry Pi Hardware Using MCP23017

After installing the support package, you can open these examples by entering the following command in a MATLAB Command Window:

```
raspi_examples
```







The software opens a window with links to each example.

MATLAB Support Package for Raspberry Pi Hardware **EXAMPLES**

MATLAB® lets you communicate with Raspberry Pi® hardware. With this capability, you can use Raspberry Pi hardware as a physical I/O device.

[Product page at m](#)

Tutorials

-  Getting Started with MATLAB Support Package for Raspberry Pi Hardware
-  Getting Started with Raspberry Pi Hardware
-  Analog Input Using SPI
-  Controlling a 4-Digit 7-Segment Display Using I2C
-  Controlling a 4-Digit 7-Segment Display Using Serial Port
-  Working with Raspberry Pi Camera Board

Connecting to Raspberry Pi Hardware

You can use MATLAB to connect to and interact with Raspberry Pi hardware. For example, you can:

- Control on-board LEDs.
- Read and write values to GPIO pins.
- Connect to devices that are connected to:
 - Serial port
 - I2C interface
 - SPI interface
- Record video and take still images using Camera Board.
- Use the Linux command shell.

When you create a connection to the Raspberry Pi hardware, you assign that connection to a handle whose name you specify. For example:

```
mypi = raspi;
```

Use the handle to control the Raspberry Pi hardware. For example, you can use the handle to illuminate an LED or read the logical state of a GPIO pin:

```
writeLED(mypi, 'led0', 1);  
readDigitalPin(mypi, 4);
```

You can also use the handle to create a connection to serial, I2C, and SPI devices attached to the Raspberry Pi hardware. For example, you can create a connection to a serial device and assign that connection to a handle:

```
myserial = serialdev(mypi, '/dev/ttyAMA0', 9600);
```

A connection remains active until you clear all of the handles that use the connection. You cannot create a new connection to a board while the previous connection to the same board is active. For example, the `mypi` and `myserial` handles from preceding examples both use the same connection. Even if you clear `mypi`, the connection remains active while `myserial` exists. Trying to create a new connection produces an error.


```
clear mypi;  
mynewpi = raspi;
```

```
Error using raspi (line 146)  
An active connection to raspberrypi-computername already exists.  
You cannot create another connection.
```

Clearing `myserial` closes the connection. You can then create a new connection to the Raspberry Pi hardware without producing an error.

```
clear myserial;  
mynewpi = raspi;
```

Related Examples

- “Connect to Raspberry Pi Hardware” on page 11-22
- “Troubleshoot Connecting to Raspberry Pi Hardware” on page 11-25

Connect to Raspberry Pi Hardware

In this section...

“Create Connection to One Board” on page 11-22

“Create Connection to a Board That Has Different Settings” on page 11-23

Create Connection to One Board

Use the `raspi` function to create a connection to Raspberry Pi hardware and assign the connection to a handle. Later, you can use the handle to interact with Raspberry Pi hardware and peripherals.

For example, enter:

```
mypi = raspi
```

```
mypi =
```

```
raspi with properties:
```

```
DeviceAddress: 'raspberrypi-computername'  
Port: 18725  
BoardName: 'Raspberry Pi Model B Rev 2'  
AvailableLEDs: {'led0'}  
AvailableDigitalPins: [4 7 8 9 10 11 14 15 17 18 22 23 24 25 27 30 31]  
AvailableSPIChannels: {}  
AvailableI2CBuses: {'i2c-0' 'i2c-1'}  
I2CBusSpeed: 100000
```

```
Supported peripherals
```

In this example:

- The `raspi` function uses the IP address, user name, and password from the most recent connection to the Raspberry Pi hardware.
- The handle, `mypi`, displays properties from the Raspberry Pi hardware.

For more information about using the handle to control and exchanging data with the Raspberry Pi hardware and peripherals, see:

- “LEDs”
- “GPIO Pins”
- “Serial Port”
- “I2C Interface”
- “SPI Interface”
- “Camera Board”
- “Linux”

Create Connection to a Board That Has Different Settings

To connect to Raspberry Pi hardware that has a different IP address, user name, or password from the previous connection, use `raspi` with arguments.

For example:

```
myotherpi = raspi('169.254.0.4', 'rocky', 'bullwinkle')
```

```
myotherpi =
```

```
raspi with properties:
```

```
DeviceAddress: '169.254.0.4'
Port: 18725
BoardName: 'Raspberry Pi Model B Rev 2'
AvailableLEDs: {'led0'}
AvailableDigitalPins: [4 7 8 9 10 11 14 15 17 18 22 23 24 25 27 30 31]
AvailableSPIChannels: {}
AvailableI2CBuses: {'i2c-0' 'i2c-1'}
I2CBusSpeed: 100000
```

```
Supported peripherals
```

Related Examples

- “Troubleshoot Connecting to Raspberry Pi Hardware” on page 11-25

Concepts

- “Connecting to Raspberry Pi Hardware” on page 11-20

Troubleshoot Connecting to Raspberry Pi Hardware

In this section...

“Connection Timed Out” on page 11-25

“Host Does Not Exist” on page 11-25

“Active Connection Already Exists” on page 11-26

Connection Timed Out

Creating a connection to Raspberry Pi hardware produces the following error:

```
mypi = raspi
```

```
Error using raspi (line 158)
```

```
Cannot establish an SSH connection to the board with device address  
"raspberrypi-computername".
```

Caused by:

```
Error using raspi.internal.sshclient/executeCommand (line 96)  
Error executing command: FATAL ERROR: Network error:  
Connection timed out
```

The MATLAB software cannot reach the Raspberry Pi hardware over the network connection. To solve this issue:

- Verify the power and network connection on the Raspberry Pi hardware.
- Perform steps described in “Complete Additional Setup Tasks” on page 11-5.

Host Does Not Exist

Creating a connection to Raspberry Pi hardware produces the following error:

```
Trial>> mypi = raspi
```

```
Error using raspi (line 160)
```

```
Cannot establish an SSH connection to the board with device address  
"raspberrypi-computername".
```

Caused by:

```
Error using raspi.internal.sshclient/executeCommand (line 96)
Error executing command: Unable to open connection:
Host does not exist
```

If you have just started the Raspberry Pi hardware, wait two to four minutes. Then, try creating a connection again.

If the Raspberry Pi hardware has a different IP address from the previous connection, complete the steps in “Get the IP Address of the Raspberry Pi Hardware” on page 11-27 and “Create Connection to a Board That Has Different Settings” on page 11-23.

Active Connection Already Exists

Reconnecting to Raspberry Pi hardware produces the following error.

```
Trial>> mypi = raspi
```

```
Error using raspi (line 146)
An active connection to raspberrypi-computername already exists.
You cannot create another connection.
```

The MATLAB Workspace contains one or more handles that were created using the previous connection. The connection is still active.

Clear handles that were created using the previous connection as described in “Connect to Raspberry Pi Hardware” on page 11-22

Related Examples

- “Connect to Raspberry Pi Hardware” on page 11-22

Concepts

- “Connecting to Raspberry Pi Hardware” on page 11-20

Get the IP Address of the Raspberry Pi Hardware

In this section...

“Hear the Spoken IP Address” on page 11-27

“Show the IP Address on a Display” on page 11-27

Hear the Spoken IP Address

To hear the IP address of the Raspberry Pi board, plug headphones or powered speakers into the audio socket on the board.

Restart the board and prepare to keep a record of the IP address.

The board uses a synthesized voice for the IP address. For example, it says: “My IP address is one hundred and seventy two point two eight point two zero one point one three seven.”

Show the IP Address on a Display

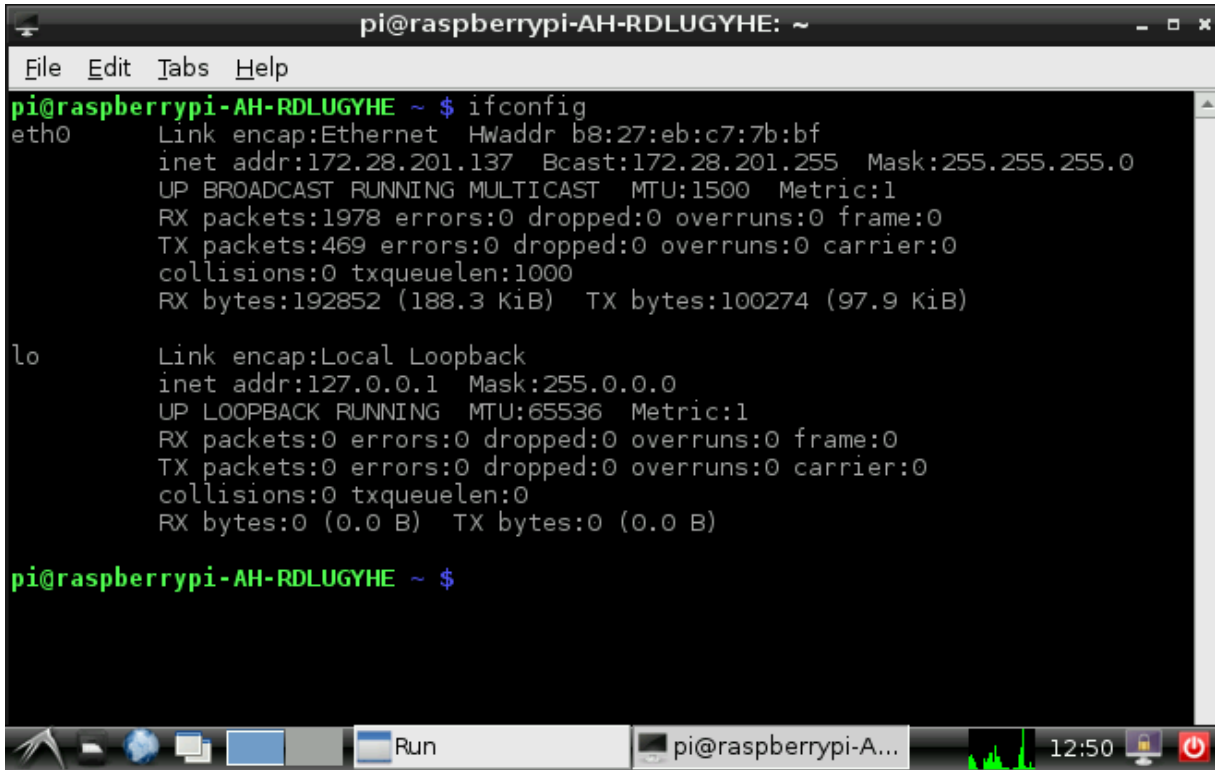
To display the IP address of the Raspberry Pi hardware, connect a keyboard and mouse to the USB ports on the board. Connect a monitor or TV to either the HDMI output or the S-video output on the board.

After starting the Raspberry Pi hardware, open the start menu on the Raspian Wheezy desktop. Select **Accessories > LXTerminal**.

The terminal displays the host name that Support Package Installer assigned to the Raspberry Pi hardware during the setup process. For example, `raspberrypi-computername` in the following illustration.

At the command prompt, enter `ifconfig`. The `inet` parameter on the second line displays the IP address of the board.

With the `raspi` function, you can use either the IP address or the host name as the `ipaddress` argument.



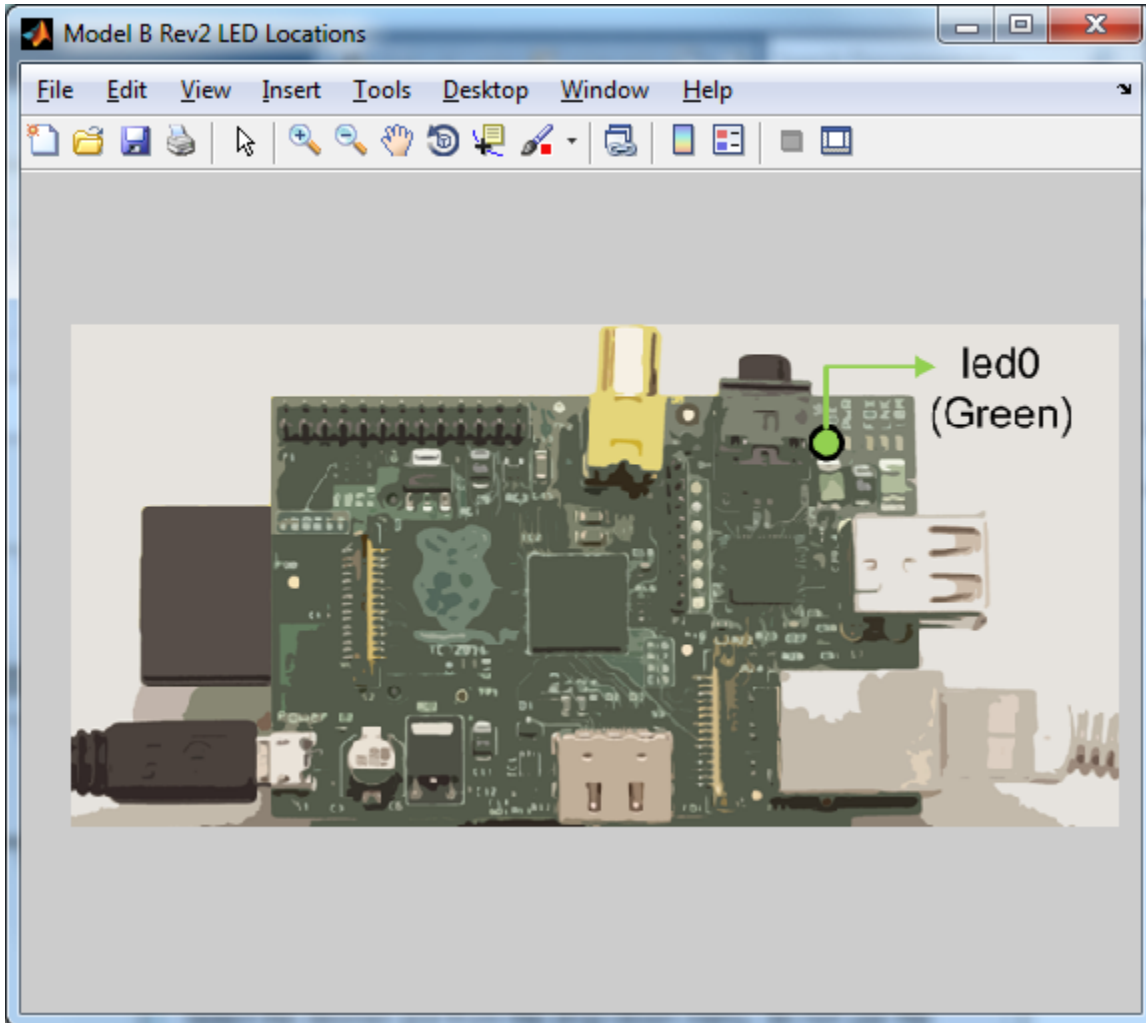
```
pi@raspberrypi-AH-RDLUGYHE: ~  
File Edit Tabs Help  
pi@raspberrypi-AH-RDLUGYHE ~ $ ifconfig  
eth0  Link encap:Ethernet  HWaddr b8:27:eb:c7:7b:bf  
      inet addr:172.28.201.137  Bcast:172.28.201.255  Mask:255.255.255.0  
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1  
      RX packets:1978 errors:0 dropped:0 overruns:0 frame:0  
      TX packets:469 errors:0 dropped:0 overruns:0 carrier:0  
      collisions:0 txqueuelen:1000  
      RX bytes:192852 (188.3 KiB)  TX bytes:100274 (97.9 KiB)  
  
lo    Link encap:Local Loopback  
      inet addr:127.0.0.1  Mask:255.0.0.0  
      UP LOOPBACK RUNNING  MTU:65536  Metric:1  
      RX packets:0 errors:0 dropped:0 overruns:0 frame:0  
      TX packets:0 errors:0 dropped:0 overruns:0 carrier:0  
      collisions:0 txqueuelen:0  
      RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)  
  
pi@raspberrypi-AH-RDLUGYHE ~ $
```

The image shows a terminal window on a Raspberry Pi. The window title is "pi@raspberrypi-AH-RDLUGYHE: ~". The terminal displays the output of the "ifconfig" command. It shows details for the "eth0" interface (Ethernet) and the "lo" interface (Local Loopback). The "eth0" interface has an IP address of 172.28.201.137 and a MAC address of b8:27:eb:c7:7b:bf. The "lo" interface has an IP address of 127.0.0.1. The terminal also shows the prompt "pi@raspberrypi-AH-RDLUGYHE ~ \$" at the bottom.

The Raspberry Pi LED

The Raspberry Pi hardware has one user-controllable LED. You can program this LED to function as a visual indicator that responds to an input or condition.

The labeling, location, and name of the LED varies by model and version. To locate and identify the LED, use the `showLEDs` function or the `AvailableLEDs` property.



To return the LED to its normal purpose as an indicator of SD card activity, restart the Raspberry Pi hardware.

For more information, see “LEDs”.

Turn the Raspberry Pi LED On and Off

This example shows how to turn the on-board LED on and off.

Create a connection to the Raspberry Pi board and assign the connection to a handle, `mypi`.

```
mypi = raspi
```

```
mypi =
```

```
raspi with properties:
```

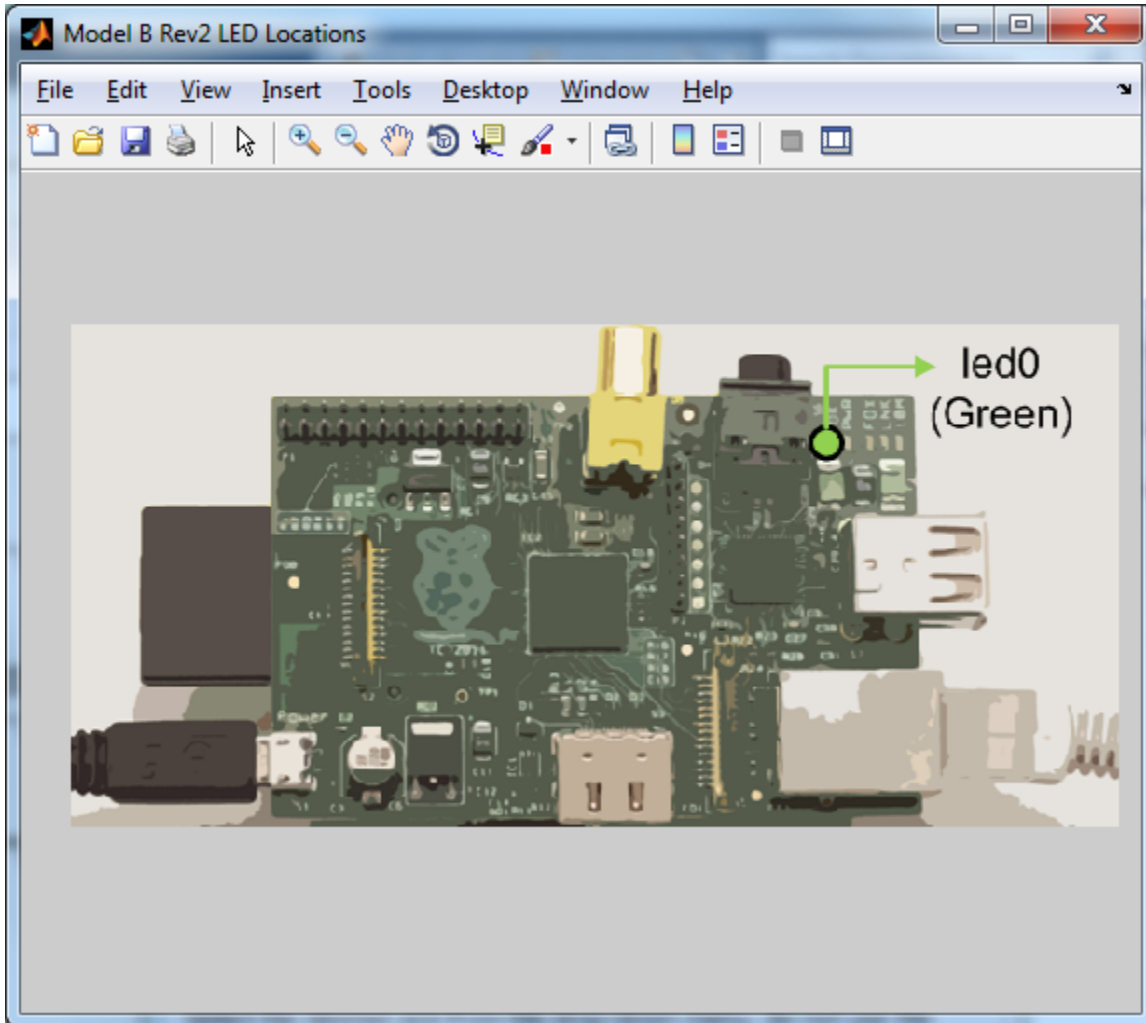
```
    DeviceAddress: 'raspberrypi-computername'  
        Port: 18725  
        BoardName: 'Raspberry Pi Model B Rev 2'  
    AvailableLEDs: {'led0'}  
AvailableDigitalPins: [4 14 15 17 18 22 23 24 25 27 30 31]  
AvailableSPIChannels: {}  
    AvailableI2CBuses: {'i2c-0' 'i2c-1'}  
        I2CBusSpeed: 100000
```

```
Supported peripherals
```

The `AvailableLEDs` property shows the name of the user-controllable LED.

To show the location of the user-controllable LED on the board, use `showLEDs`.

```
showLEDs(mypi)
```



Turn the specified LED on by setting its value to 1 or true.

```
writeLED(mypi, 'led0', 1)
```

Turn the LED off by setting its value to 0 or false.

```
writeLED(mypi, 'led0', false)
```

To restore the LED to its default purpose, which is to indicate SD card activity, restart the Raspberry Pi hardware.

Flash the Raspberry Pi LED in Response to an Input

This example shows how to flash the on-board LED when you press a button that is connected to a GPIO pin.

Excessive voltage and current can damage the Raspberry Pi hardware. Observe the manufacturer's precautions for handling the Raspberry Pi hardware and connecting it to other devices. For more information, see <http://www.raspberrypi.org/technical-help-and-resource-documents>.

Using a breadboard, set up the following circuit:

- Connect one of the +3.3V pins on the GPIO header to a button.
- Connect pin 23 on the GPIO header to a 220 or 330 Ohm resistor.
- Connect the unconnected ends of the button and resistor to each other.

Note Use `showLEDs` to show an illustration that identifies each pin.

Run the code and press the button. The button closes the circuit between the +3.3V pin and pin 23. When `readDigitalPin` detects the raised voltage, `if buttonPressed` becomes true, and `writeLED` toggles the LED on and off ten times.

```
for ii = 1:100
    buttonPressed = readDigitalPin(mypi,23);
    if buttonPressed
        for jj = 1:10
            writeLED(mypi, 'led0',1);
            pause(0.05);
            writeLED(mypi, 'led0',0);
            pause(0.05);
        end
    end
    pause(0.1);
end
```

The Raspberry Pi GPIO Pins

The Raspberry Pi hardware shares digital pins between the GPIO, Serial, SPI, and I2C interfaces. Enabling or disabling the SPI and I2C interfaces changes the availability of specific pins for use as GPIO pins.

For example, with Raspberry Pi, Model B, Rev 2:

- The SPI interface is disabled by default. Enabling the SPI interface uses pins 7, 8, 9, 10, and 11.
- The I2C interface is enabled by default. Disabling the I2C interface frees pins 2, 3, 28, and 29.
- With both interfaces disabled, the available digital pins are: 2, 3, 4, 7, 8, 9, 10, 11, 14, 15, 17, 18, 22, 23, 24, 25, 27, 28, 29, 30, 31.

You can configure a GPIO pin as an input or output. If a pin is unconfigured, reading from the pin configures it as an input, and writing to the pin configures it as an output.

When you write 1 to a GPIO pin, the pin outputs +3.3V. When you write 0 to the pin, or do nothing, the pin is grounded, and outputs 0V.

When you read the GPIO pin, Raspberry Pi hardware detects the voltage of the pin. If the input voltage has approximately the same voltage as ground, around 0V, the logical value of the pin is 0. If the input voltage is higher, approximately +3.3V, the logical value of the pin is 1.

To change a pin that has already been configured, you must use `configureDigitalPin`. The Raspberry Pi hardware requires this extra step to help prevent accidental damage to the board and other components. Otherwise, for example, you could burn out an input pin that is connected to ground by writing to it.

Use the Raspberry Pi GPIO Pins as Digital Inputs and Outputs

This example shows how to use the digital pins on the Raspberry Pi hardware as digital inputs and outputs.

Excessive voltage and current can damage the Raspberry Pi hardware. Observe the manufacturer's precautions for handling the Raspberry Pi hardware and connecting it to other devices. For more information, see <http://www.raspberrypi.org/technical-help-and-resource-documents>.

When you create a connection to the Raspberry Pi hardware, the `AvailableDigitalPins` property shows the list of digital pins that are available.

```
mypi = raspi
```

```
mypi =
```

```
raspi with properties:
```

```
DeviceAddress: 'raspberrypi-computername'  
Port: 18725  
BoardName: 'Raspberry Pi Model B Rev 2'  
AvailableLEDs: {'led0'}  
AvailableDigitalPins: [4 7 8 9 10 11 14 15 17 18 22 23 24 25 27 30 31]  
AvailableSPIChannels: {}  
AvailableI2CBuses: {'i2c-0' 'i2c-1'}  
I2CBusSpeed: 100000
```

```
Supported peripherals
```

The Raspberry Pi hardware shares some digital pins with the SPI and I2C interfaces. Enabling or disabling those interfaces changes the number of available pins.

To review the list of digital pins are available, use the `AvailableDigitalPins` property.

```
mypi.AvailableDigitalPins
```



```
ans =
```

```
Columns 1 through 13
```

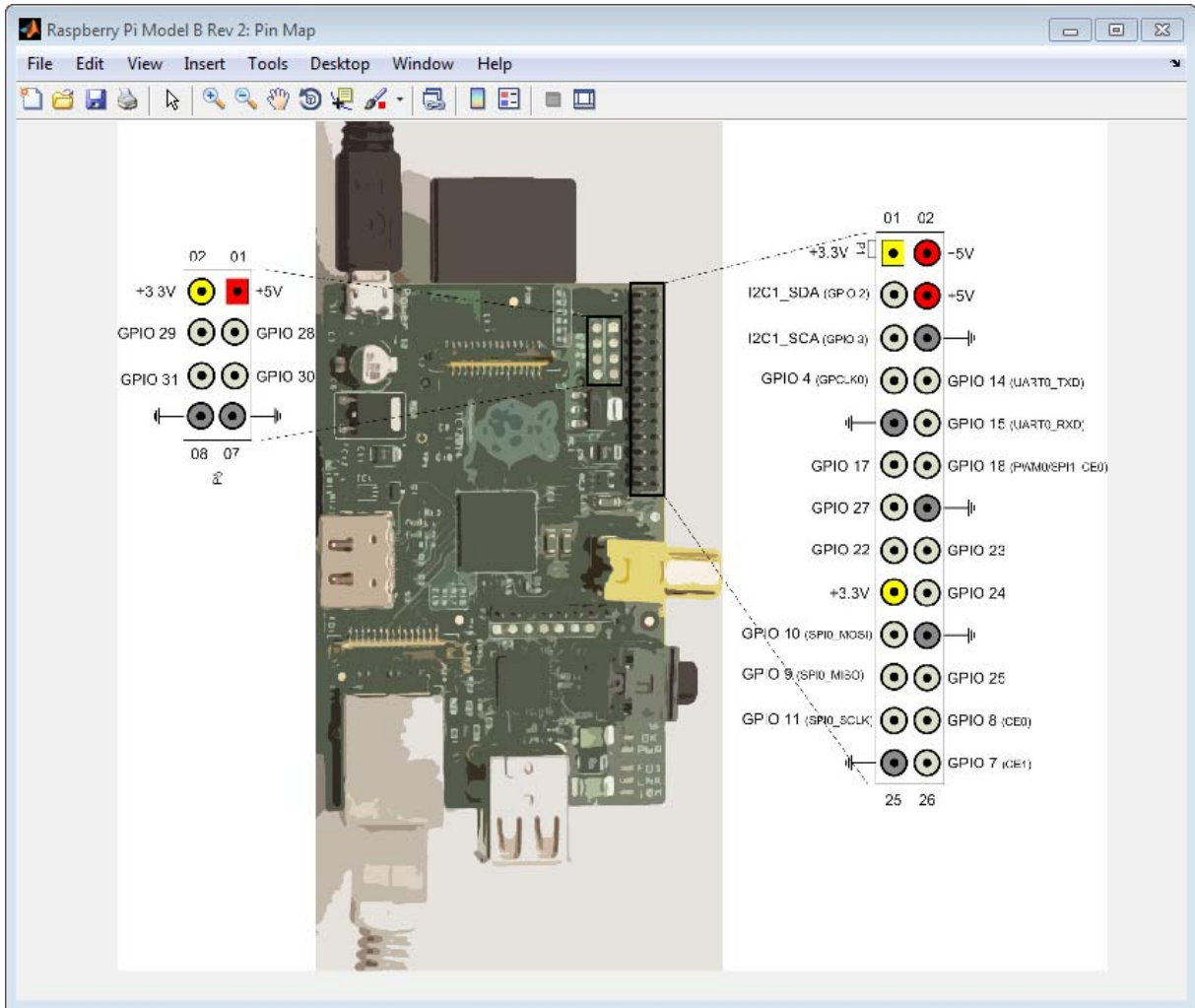
```
4 7 8 9 10 11 14 15 17 18 22 23 24
```

```
Columns 14 through 17
```

```
25 27 30 31
```

To show a pin diagram for the specific model of the Raspberry Pi hardware that you are using, use `showPins`.

```
showPins(myPi)
```



To configure a pin as a digital input, pass an input value to `configureDigitalPin`.

```
configureDigitalPin(myPi,4,'input')
```

This example configures pin 4 as an input.

To read the value of a digital pin, use `readDigitalPin`.

```
readDigitalPin(myPi,4)
```

```
ans =
```

```
1
```

This example shows that a wire connected to pin 4 has an elevated voltage, which produces a logical value of 1 (true). If the wire has no voltage, the logical value of pin 4 is 0 (false).

To configure a pin as a digital output, pass an output value to `configureDigitalPin`.

```
configureDigitalPin(myPi,7,'output')
```

To write a logical value to a digital pin, use `writeDigitalPin`.

```
writeDigitalPin(myPi,7,1)
```

This example writes a logical value of 1 to pin 7.

Troubleshoot Raspberry Pi GPIO Pins

In this section...
“Error Using raspi/writeDigitalPin” on page 11-40
“Error Using raspi/readDigitalPin” on page 11-40
“Unexpected Digital Pin Number” on page 11-41

Error Using raspi/writeDigitalPin

Writing a logical value to a pin produces the following error:

```
writeDigitalPin(mypi,7,1)
```

```
Trial>> writeDigitalPin(mypi,4,1)
Error using raspi/writeDigitalPin (line 451)
Digital pin 4 was previously configured for input.
Set the digital pin configuration to 'output' in order to write to it.
```

The pin is configured as an input. To solve this issue:

- Use a different pin number.
- Use `configureDigitalPin` to reconfigure the pin as an output.

For more information, see “Use the Raspberry Pi GPIO Pins as Digital Inputs and Outputs” on page 11-36.

Error Using raspi/readDigitalPin

Reading the logical value of a pin produces an error.

```
readDigitalPin(mypi,4)
```

```
Error using raspi/readDigitalPin (line 433)
Digital pin 4 was previously configured for output. Set the digital pin
configuration to 'input' in order to read from it.
```

The pin is configured as an output.

To solve this issue, do either of the following:

- Use a different pin number.
- Use `configureDigitalPin` to reconfigure the pin as an input.

For more information, see “Use the Raspberry Pi GPIO Pins as Digital Inputs and Outputs” on page 11-36.

Unexpected Digital Pin Number

Using a specific pin number produces an error.

```
writeDigitalPin(mypi,7,1)
```

```
writeDigitalPin(mypi,7,1)
Error using raspi/checkDigitalPin (line 688)
Unexpected digital pin number.
Use AvailableDigitalPins property for a list of digital
pin numbers you can use.
```

```
Error in raspi/writeDigitalPin (line 444)
    obj.checkDigitalPin(pinNumber);
```

The pin is unavailable for use as a digital pin. The I2C interface or the SPI interface might be using the pin.

To solve this issue, do either of the following:

- Use the `AvailableDigitalPins` property to identify which pins are available, and then use a different pin number.
- Disable the I2C or SPI interface that is using the pin. For more information, see `disableSPI` and `disableI2C`.

For more information, see “Use the Raspberry Pi GPIO Pins as Digital Inputs and Outputs” on page 11-36.

The Raspberry Pi Serial Port

The Raspberry Pi serial port provides low speed +3.3V TTL RS-232 data communication with a wide variety of devices, such as sensors, displays, ADCs, and DACs. The serial port UART connects to two pins on the GPIO header:

- **GPIO 14 (UART0_TXD)** transmits data to the RxD pin on the peripheral device.
- **GPIO 15 (UART0_RXD)** receives data from the TxD pin on the peripheral device.

By default, the serial console in the customized version of Raspian Wheezy on your Raspberry Pi hardware is disabled. To use the `serialdev`, the serial console must be disabled.

For more information, see:

- “Serial Port”
- http://en.wikipedia.org/wiki/Serial_port

Do not connect the Raspberry Pi serial port to 12V RS-232 serial ports, which are found on many older computers.

Use the Raspberry Pi Serial Port to Connect to a Device

This example shows how to create a connection to a serial device, write data to the device, and read data from the device.

By default, the serial console in the customized version of Raspian Wheezy on your Raspberry Pi hardware is disabled. To use the `serialdev`, the serial console must be disabled.

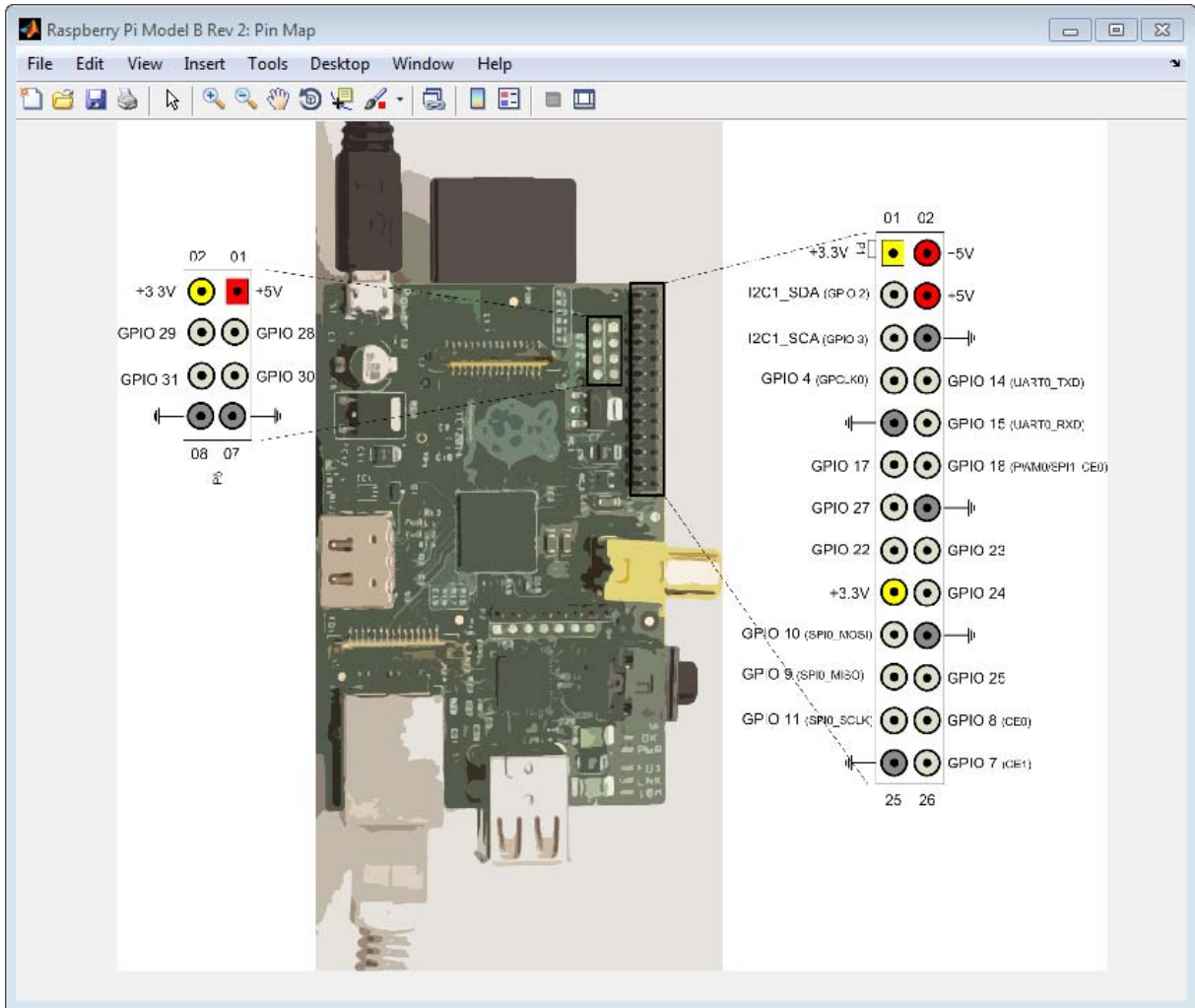
Excessive voltage and current can damage the Raspberry Pi hardware. Observe the manufacturer's precautions for handling the Raspberry Pi hardware and connecting it to other devices. For more information, see <http://www.raspberrypi.org/technical-help-and-resource-documents>.

Create a connection to the Raspberry Pi board.

```
mypi = raspi;
```

Show the location of the Tx and Rx pins, GPIO 14 (UART0_TXD) and GPIO 15 (UART0_RXD), on the GPIO header.

```
showPins(mypi)
```



Raspberry Pi hardware uses +3.3V. Do not connect Raspberry Pi hardware directly to devices that use higher voltages.

Connect the Raspberry Pi board to a +3.3V serial device.

- To receive data, connect the GPIO 15 (UART0_RXD) pin on the Raspberry Pi board to the TxD pin on the serial device.

- To transmit data, connect the GPIO 14 (UART0_TXD) pin on the Raspberry Pi board to the Rx pin on the serial device.
- Connect a ground pin, GND, on the Raspberry Pi board to the GND pin on the serial device.
- Connect a +3.3V pin on the Raspberry Pi board to the VCC pin on the serial device.

You can check the status of the serial console.

```
system(mypi, 'rpi-serial-console status')
```

```
ans =
```

```
Serial console on /dev/ttyAMA0 is enabled
```

By default, the serial console is enabled in the custom version of Raspian Wheezy on the Raspberry Pi hardware.

If the serial console is enabled, and you want to use `serialdev`, disable the console and reboot the Raspberry Pi hardware.

```
system(mypi, 'sudo rpi-serial-console disable');  
system(mypi, 'sudo shutdown -r now');  
clear mypi;
```

When the Raspberry Pi hardware finishes rebooting, you can use `serialdev` to exchange data with serial devices.

Before continuing, research the manufacturer's product information to determine which baud rate, data bits, parity, and stop bit settings the serial device supports.

Use `serialdev` to create a connection to the serial device and assign the connection to a handle.

```
myserialdevice = serialdev(mypi, '/dev/ttyAMA0')
```

```
myserialdevice =
```

```
serialdev with properties:
```

```
BaudRate: 115200
DataBits: 8
  Parity: 'none'
StopBits: 1
Timeout: 10
```

In this example, the connection uses the default values for baud rate (115200), data bits (8), parity ('none'), and stop bit (1).

If the serial device requires nondefault values, use a set of optional arguments to override those defaults.

```
myserialdevice = serialdev(mypi, '/dev/ttyAMA0', 115200, 8, 'none', 2)

myserialdevice =
```

```
  serialdev with properties:
```

```
BaudRate: 115200
DataBits: 8
  Parity: 'none'
StopBits: 2
Timeout: 10
```

This example overrides the default value of `StopBits` by setting it to 2. It uses the other arguments to maintain the correct sequence of arguments to the left of the rightmost overriding value.

You can write a values the serial device.

```
write(myserialdevice, [10 12], 'uint16')
```

This example writes two values to the serial device. It overrides the default precision, `uint8`, by setting it to `uint16`.

You can also read an array of values from the serial port.

```
output = read(myserialdevice, 100);
```

This example reads a 100-element array of `uint8` values from the serial device.

If the serial connection times out during read operations, you can adjust the time out period by assigning a new value to the `Timeout` property.

```
myserialdevice.Timeout = 20
```

```
myserialdevice =
```

```
    serialdev with properties:
```

```
        BaudRate: 115200
```

```
        DataBits: 8
```

```
        Parity: 'none'
```

```
        StopBits: 1
```

```
        Timeout: 20
```

To use the serial console, enable the serial console and reboot the Raspberry Pi hardware.

```
system(mypi, 'sudo rpi-serial-console enable');  
system(mypi, 'sudo shutdown -r now');  
clear mypi;
```

When the Raspberry Pi hardware finishes rebooting, you can communicate with the Linux command interface over a USB to TTL serial cable that is connected to the serial pins on the GPIO header. However, you cannot use `serialdev` until you disable the console.

Troubleshoot the Raspberry Pi Serial Port

Missing or Garbled Data

When you try to exchange data with the serial device, the data is garbled or missing.

```
write(myserialdevice,[10 12], 'uint16')
```

Creating a connection and writing data to a serial device does not provide any indication of success or failure.

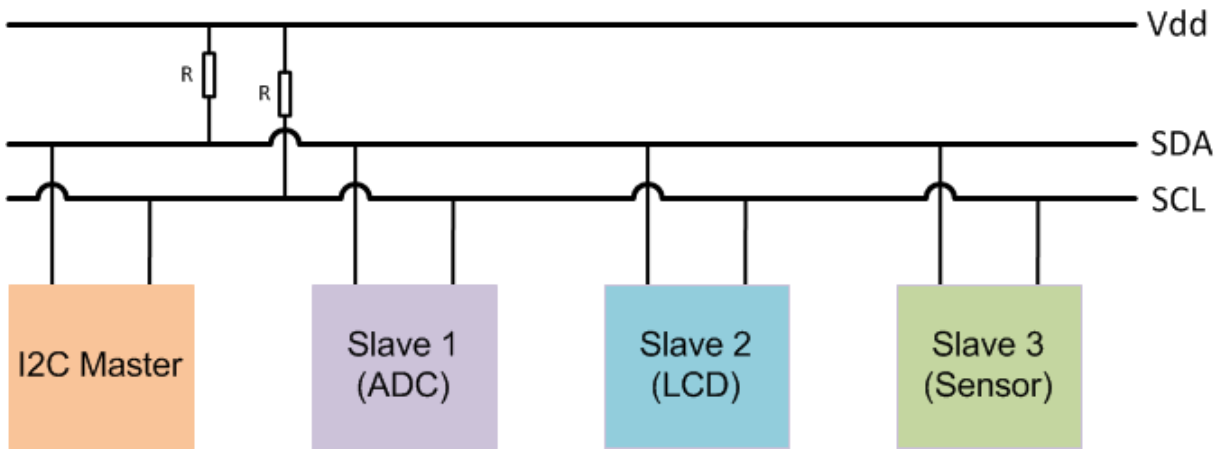
The device is not connected to the Raspberry Pi serial port. Or, the connection to the serial device is not configured correctly.

To solve this issue, do the following:

- Verify the physical connection between the Rx and Tx pins and their counterparts on the serial device.
- Verify the serial port settings that the serial device requires. Clear the handle for the current serial connection. Create a new connection that uses the correct serial port settings.
- Verify the data precision that the serial device requires. Write data using the correct serial port settings.

The Raspberry Pi I2C Interface

Inter-Integrated Circuit (I2C) is a protocol for communicating with low-speed peripherals.



Depending on the model and revision of your board, Raspberry Pi hardware has one or two I2C buses. Each bus has an I2C Master connected to two bidirectional lines, serial data line (SDA), and serial clock (SCL). These two lines are connected to a pair of pins, such as I2C1_SDA (GPIO2) and I2C1_SCL (GPIO3), on the GPIO header.

You can connect multiple I2C devices, such as ADCs, LCDs, and sensors, to the I2C pins on the Raspberry Pi hardware. Each I2C device on an I2C bus must have a unique address. Most devices have a default address that is assigned by the manufacturer. If the address is not unique, follow the manufacturer's instructions for reconfiguring the address. Often, you can reconfigure the address using a pair of jumpers on the device. The Raspberry Pi hardware supports only 7-bit addresses. This 7-bit address space supports 128 unique addresses.

The I2C pins on the Raspberry Pi hardware are pulled up with 1.8 kOhm resistors. The I2C devices must support +3.3V and not draw more current than the Raspberry Pi's maximum.

Use the Raspberry Pi I2C Interface to Connect to a Device

This example shows how to create a connection to an I2C device, write data to the device, and read data from the device.

Excessive voltage and current can damage the Raspberry Pi hardware. Observe the manufacturer's precautions for handling the Raspberry Pi hardware and connecting it to other devices. For more information, see <http://www.raspberrypi.org/technical-help-and-resource-documents>.

Create a connection to the Raspberry Pi board.

```
mypi = raspi
```

```
mypi =
```

```
raspi with properties:
```

```
DeviceAddress: 'raspberrypi-computername'  
Port: 18725  
BoardName: 'Raspberry Pi Model B Rev 2'  
AvailableLEDs: {'led0'}  
AvailableDigitalPins: [4 7 8 9 10 11 14 15 17 18 22 23 24 25 27 30 31]  
AvailableSPIChannels: {}  
AvailableI2CBuses: {'i2c-0' 'i2c-1'}  
I2CBusSpeed: 100000
```

```
Supported peripherals
```

The default I2C bus speed is 100000 bits per second.

You can redisplay the `AvailableI2CBuses` and `I2CBusSpeed` properties.

```
mypi.AvailableI2CBuses
```

```
mypi.I2CBusSpeed
```

```
ans =
```

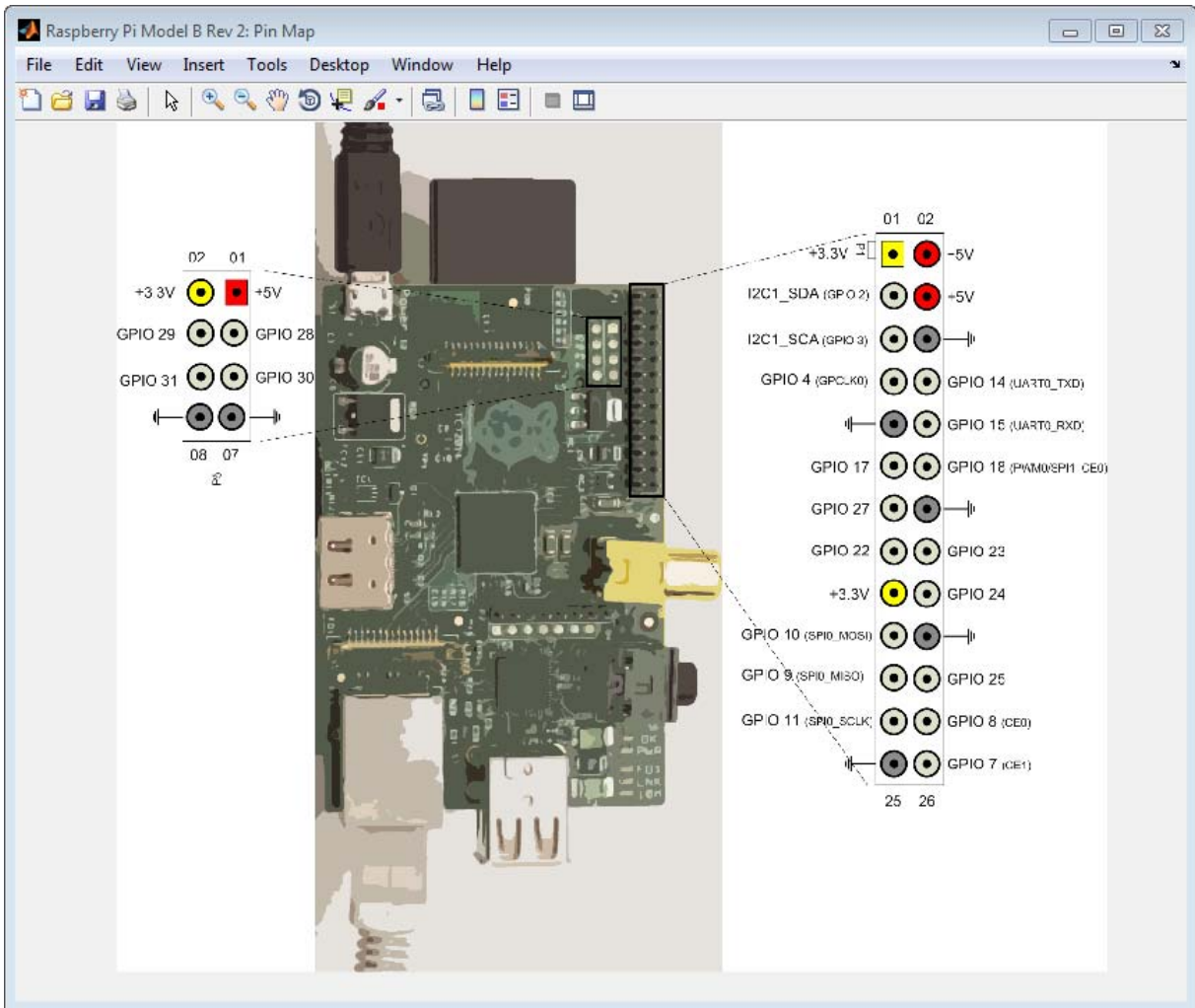
```
'i2c-0' 'i2c-1'
```

ans =

100000

Show the location of the I2C pins on the GPIO header.

showPins(myPi)



The pin map shows that, for this model and revision of the board, the `i2c-1` bus is available on the GPIO header pins `I2C1_SDA` (GPIO 2) and `I2C1_SCL` (GPIO 3).

Raspberry Pi hardware uses +3.3V. Do not connect Raspberry Pi hardware directly to devices that deliver higher voltages.

Before continuing, research the manufacturer's product information to determine which settings the I2C device supports. Then, connect the Raspberry Pi board to the I2C device.

For example, with the MCP4725 12-bit DAC, connect:

- `I2C1_SDA` (GPIO2) pin on the Raspberry Pi board to the SDA pin on the DAC.
- `I2C1_SCL` (GPIO3) pin on the Raspberry Pi board to the SCL pin on the DAC.
- GND on the Raspberry Pi board to the GND pin on the DAC.
- +3.3V on the Raspberry Pi board to the VDD pin on the DAC.
- VOUT pin on the DAC to the positive lead on the voltmeter.
- GND to the negative lead on the voltmeter.

Get the addresses of I2C devices that are attached to the I2C bus, `'i2c-1'`.

```
scanI2Cbus(mypi, 'i2c-1')
```

```
ans =
```

```
    '0x62'
```

Create a connection to the I2C DAC at `'0x62'` and assign that connection to a handle, `i2cdac`.

```
i2cdac = i2cdev(mypi, 'i2c-1', '0x62')
```

```
i2cdac =
```

```
    i2cdev with properties:
```



```
Bus: 'i2c-1'  
Address: '0x62'
```

To write a value to the I2C device.

```
write(i2cdac,4092);
```

To read a value from an I2C sensor, physically connect the sensor, use `scanI2Cbus` to get the address, use `i2cdev` to create a connection to the device. Then, use `read` to get the value.

```
addr = scanI2Cbus(mypi, 'i2c-1');  
i2csensor = i2cdev(mypi, 'i2c-1', char(addr));  
read(i2csensor,1)
```

If you are not using I2C, disable I2C to make additional GPIO pins available.

```
disableI2C(mypi)
```

When you use I2C again, enable I2C.

```
enableI2C(mypi)
```

To change the I2C bus speed, `mypi.I2CbusSpeed`, use `enableI2C` with the `i2cBusSpeed` argument.

```
disableI2C(mypi);  
enableI2C(mypi,400000);  
mypi.I2CbusSpeed
```

```
ans =
```

```
40000
```

Troubleshoot the Raspberry Pi I2C Interface

Trying to create a connection with an I2C device produces an error:

```
addr = scanI2Cbus(mypi, 'i2c-1');  
i2csensor = i2cdev(mypi, 'i2c-1', char(addr));
```

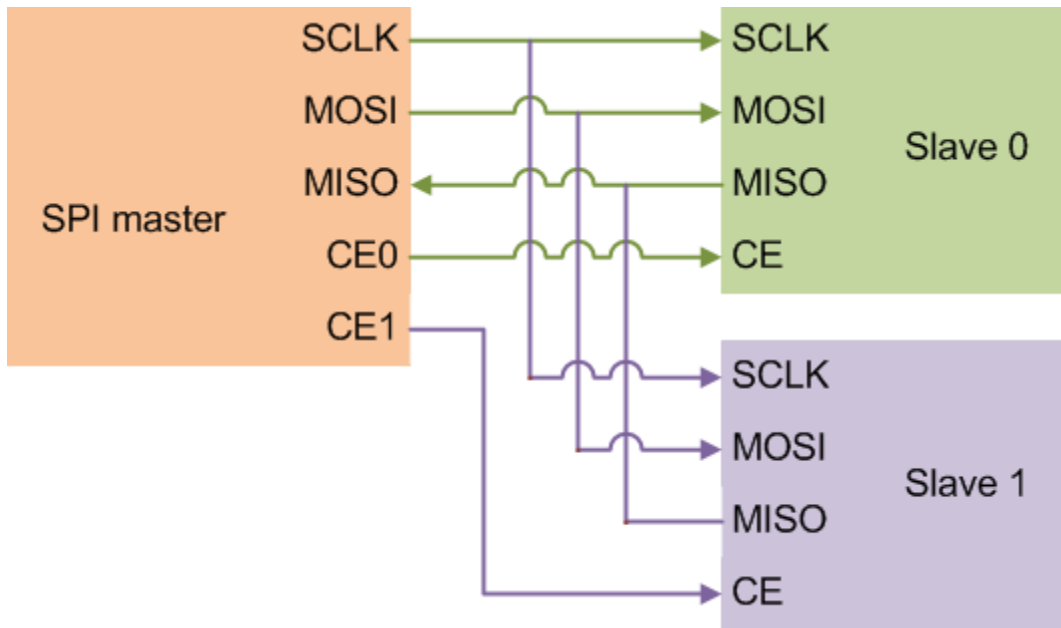
```
Error using raspi.internal.i2cdev (line 67)  
An active connection to I2C device at address 0x62 already exists.  
You cannot create another connection.
```

```
Error in raspi/i2cdev (line 507)  
    i2cObj = raspi.internal.i2cdev(obj, varargin{:});
```

Another connection to the device at that address exists. Create a connection to a device that has a different address, or clear the handle for the current connection.

The Raspberry Pi SPI Interface

Serial Peripheral Interface (SPI) is a full-duplex serial protocol for communicating with high-speed peripherals.



The SPI Master on Raspberry Pi hardware can drive two SPI peripheral device. The SPI Master has four pins:

- GPIO 11 (SPI0_SCLK) outputs a serial clock signal to synchronize communications.
- GPIO 10 (SPI0_MOSI) outputs data to the SPI peripheral device.
- GPIO 9 (SPI0_MISO) receives data from the SPI peripheral device.
- GPIO 8 (SPI0_CE0) enables one SPI peripheral device.
- GPIO 7 (SPI0_CE1) enables the other SPI peripheral device.

You can connect two SPI devices, such as displays, sensors, and flash storage to the SPI pins on the Raspberry Pi hardware. Connect both devices to the SCLK, MOSI, and MISO pins. Connect each device to one of the CE pins.

SPI on Raspberry Pi hardware supports:

- Modes 0, 1, 2 or 3
- 8 bits per word
- Data speeds: 500000, 1000000, 2000000, 4000000, 8000000, 16000000, 32000000

For more information, see:

- “SPI Interface”
- http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus

Use the Raspberry Pi SPI Interface to Connect to a Device

This example shows how to exchange data with an SPI device.

Excessive voltage and current can damage the Raspberry Pi hardware. Observe the manufacturer's precautions for handling the Raspberry Pi hardware and connecting it to other devices. For more information, see <http://www.raspberrypi.org/technical-help-and-resource-documents>.

Create a connection to the Raspberry Pi board.

```
mypi = raspi
```

```
mypi =
```

```
raspi with properties:
```

```

    DeviceAddress: 'raspberrypi-computername'
      Port: 18725
    BoardName: 'Raspberry Pi Model B Rev 2'
  AvailableLEDs: {'led0'}
AvailableDigitalPins: [4 7 8 9 10 11 14 15 17 18 22 23 24 25 27 30 31]
AvailableSPIChannels: {}
  AvailableI2CBuses: {'i2c-0' 'i2c-1'}
    I2CBusSpeed: 100000

```

```
Supported peripherals
```

By default, SPI is disabled, so `AvailableSPIChannels` does not show any channels.

Enable SPI and get the channels.

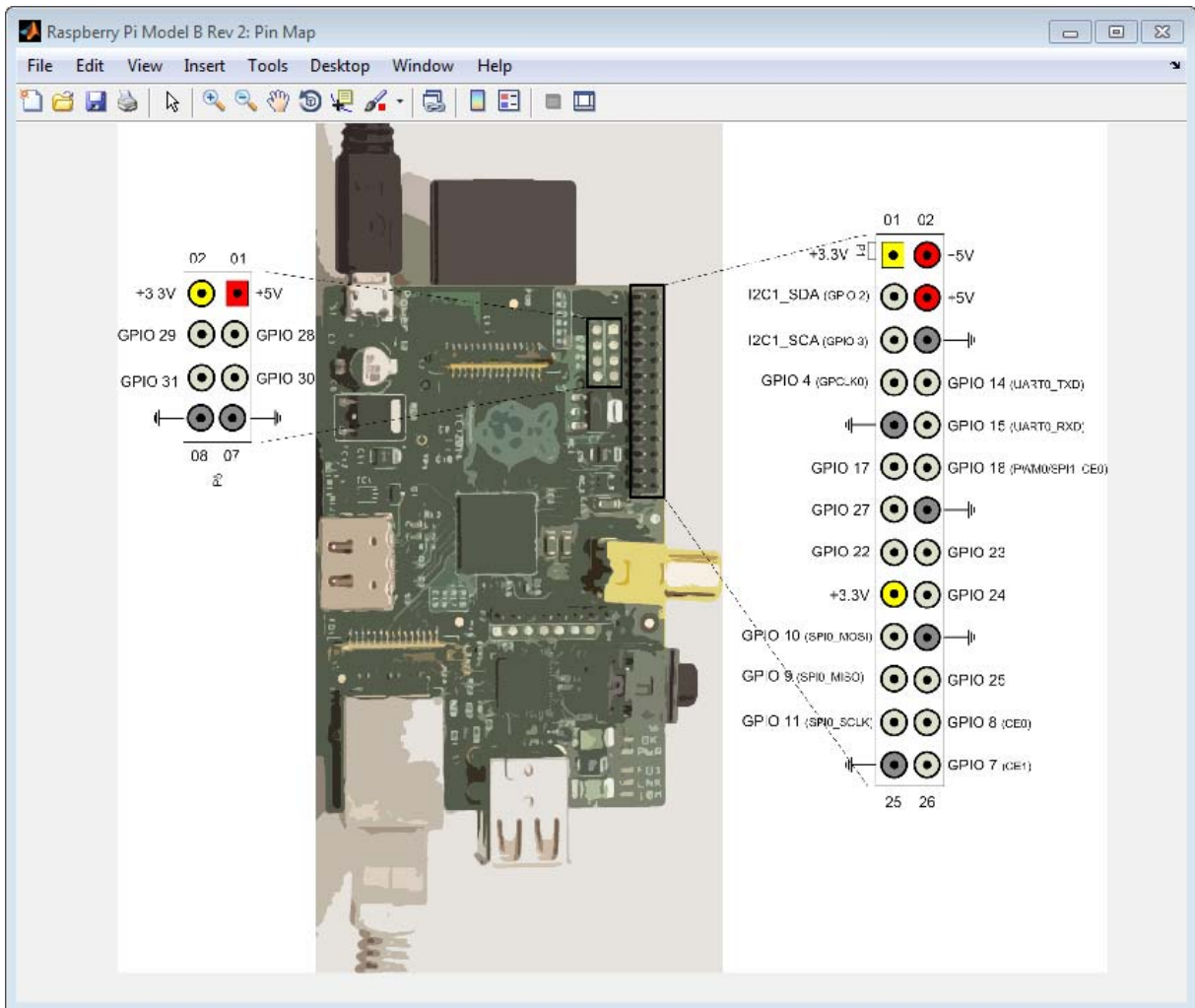
```
enableSPI(mypi);
mypi.AvailableSPIChannels
```

```
ans =
```

```
    'CE0'    'CE1'
```

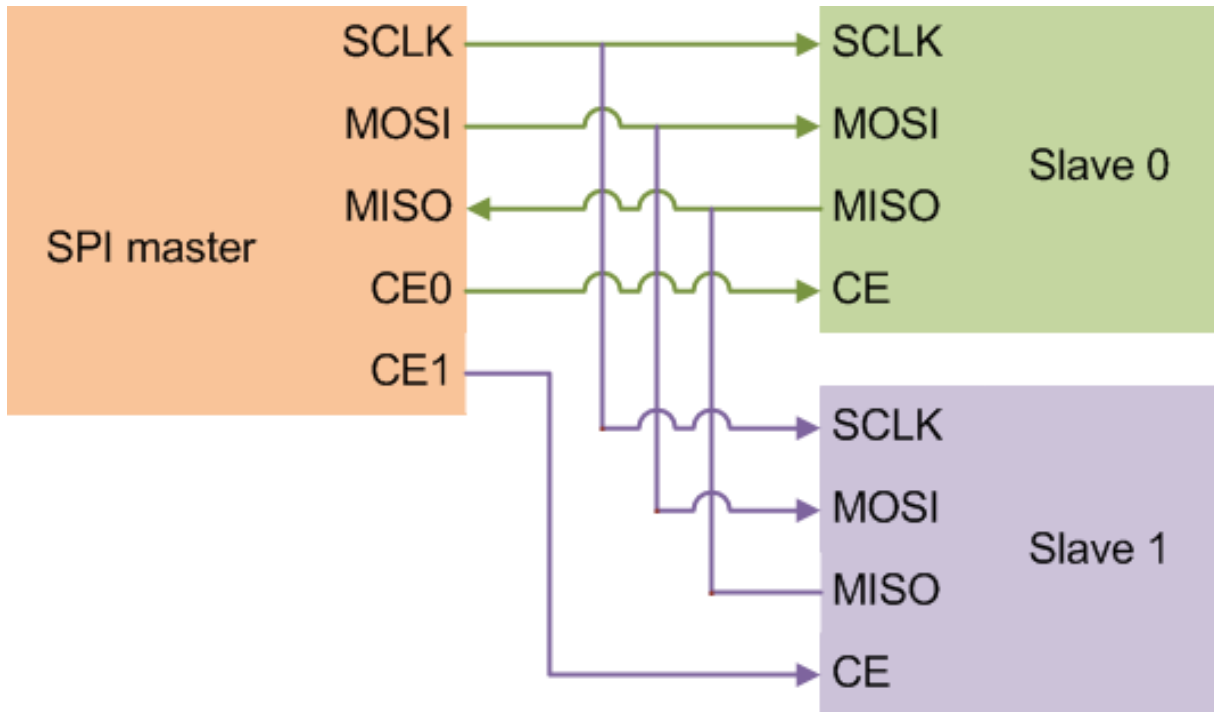
Show the location of the SPI pins, such as GPIO 10 (SPI0_MOSI), GPIO 9 (SPI0_MISO), and GPIO 11 (SPI0_SCLK) in the following illustration.

```
showPins(myPi)
```



Before continuing, research the manufacturer's product information to determine which settings the SPI device supports.

Physically connect the Raspberry Pi hardware to one or two SPI devices. Connect the SCLK, MOSI, and MISO pins to their counterparts on the SPI devices. Connect the CE0 pin on Raspberry Pi hardware to the CE pin on one SPI device. Connect the CE1 pin on Raspberry Pi hardware to the CE pin on other SPI device.



Create a connection to one of the SPI devices.

```
myspidevice = spidev(mypi, 'CE1')
```

```
myspidevice =
spidev with properties:
  Channel: 'CE1'
  Mode: 0
  BitsPerWord: 8
  Speed: 500000
```

The SPI device determines the data speed. Raspberry Pi hardware supports speeds from 0.5 MHz to 32 MHz (*myspidevice.Speed* from 500000 to 32000000)

SPI is full duplex. Perform read or write operations concurrently using `writeRead`. To read data from SPI, send dummy values. To write data to SPI, discard the data it returns.

```
out = writeRead(myspidevice,[hex2dec('08') hex2dec('D4')])
```

```
out =  
      7 211
```

If you are not using SPI, disable SPI to make additional GPIO pins available.

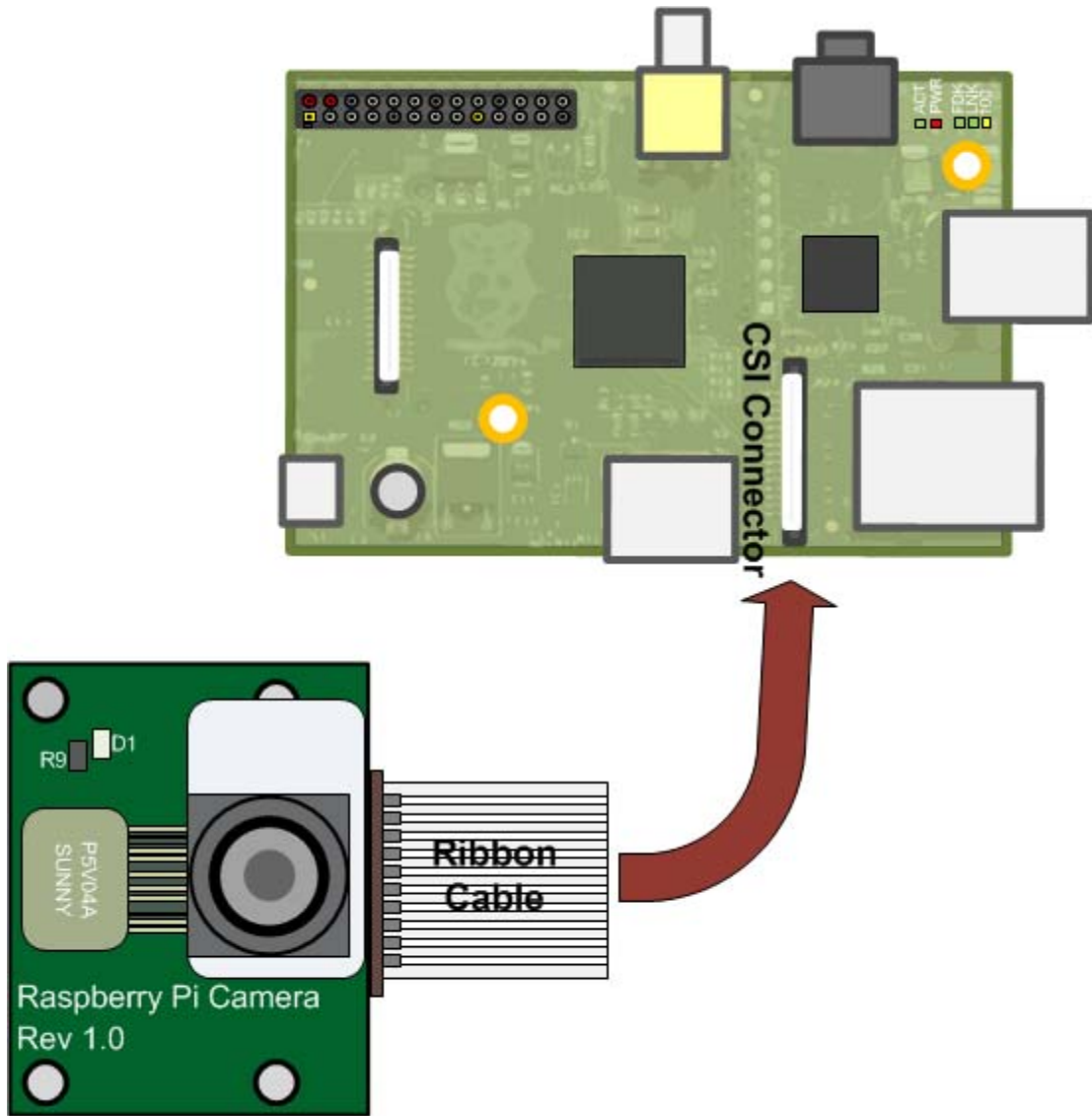
```
disableSPI(myipi)
```


The Raspberry Pi Camera Board

The Camera Board is an add-on device for capturing still images and video on the Raspberry Pi hardware.

The Camera Board provides many configurable settings such as exposure compensation, orientation, region of interest (ROI), and special effects.

Follow the manufacturer's instructions for connecting the Camera Board to the CSI connector on the Raspberry Pi hardware.



Use the Raspberry Pi Camera Board to Capture Images and Video

This example shows how to create a connection to the Camera Board, capture still images, and record video.

Create a connection to the Raspberry Pi board.

```
mypi = raspi;
```

Create a connection to the Camera Board and assign the connection to a handle, `mycam`. You can use Name-Value pairs to override the default values of most properties, like the `Resolution` property, shown here.

```
mycam = cameraboard(mypi, 'Resolution', '1280x720')
```

```
mycam =
```

```
cameraboard with properties:
```

```
          Name: Camera Board
          Re '1280x720'      (View available resolutions)
          Rotation: 0        (0, 90, 180 or 270)
HorizontalFlip: 0
VerticalFlip: 0
          FrameRate: 30     (2 to 30)
          Recording: 0

Quality
          Brightness: 50    (0 to 100)
          Contrast: 0       (-100 to 100)
          Saturation: 0     (-100 to 100)
          Sharpness: 0      (-100 to 100)

Exposure and AWB
          ExposureMode: 'auto' (View available exposure modes)
          ExposureCompensation: 0 (-10 to 10)
          AWBMode: 'auto' (View available AWB modes)
          MeteringMode: 'average' (View available metering modes)
```

```
Effects
    ImageEffect: 'none'          (View available image effects)
    VideoStabilization: 'off'
    ROI: [0.00 0.00 1.00 1.00] (0.0 to 1.0 [top, left, width, height])
```

Import and display a sequence of ten snapshots on your host computer.

```
for ii = 1:10
    img = snapshot(mycam);
    imagesc(img);
    drawnow;
end
```

If the image is upside down, change the orientation of the image.

```
mycam.Rotation = 180;
```

You can change the values of many mycamera properties listed in the "Name-Value Pair Arguments" for cameraboard.

Record a 10 second video.

```
record(mycam, 'myvideo.mp4', 10)
```

Before the specified number of seconds have elapsed, you can stop recording video.

```
stop(mycam)
```

Copy the video from the board to your host computer.

```
getFile(mypi, 'myvideo.mp4', 'C:\MATLAB')
```

To free up space, delete the video from the board.

```
deleteFile(mypi, 'myvideo.mp4')
```

Troubleshoot the Raspberry Pi Camera Board

If you have trouble using the Camera Board:

- Verify the physical connection between the Camera Board's ribbon cable and the CSI connector.
- Use `getFile` to transfer `.mp4` video and `.jpg` image files to your host computer. Then use `deleteFile` to permanently remove those same files from the memory on the Raspberry Pi hardware.
- Use `clear` to remove the current handles. Then use `raspi` and `cameraboard` to create a new connection to the Camera Board.

The Raspberry Pi Linux Command Interface

You can access the Linux command interface on the Raspberry Pi hardware. The `system` function runs Linux commands on the Raspberry Pi hardware. The `openShell` function opens a terminal window on the host computer that is connected to the Linux command-line interface on the Raspberry Pi hardware. Both functions use SSH encryption.

For more information, see:

- “Linux”.
- http://en.wikipedia.org/wiki/Secure_Shell
- http://elinux.org/CLI_Spells
- http://linuxcommand.org/learning_the_shell.php

Run Linux Commands on Raspberry Pi Hardware

Use the `system` function to run Linux commands on the Raspberry Pi hardware. Pass commands, such as `'ls -al'`, as the second argument. The function returns standard output from the Linux command line.

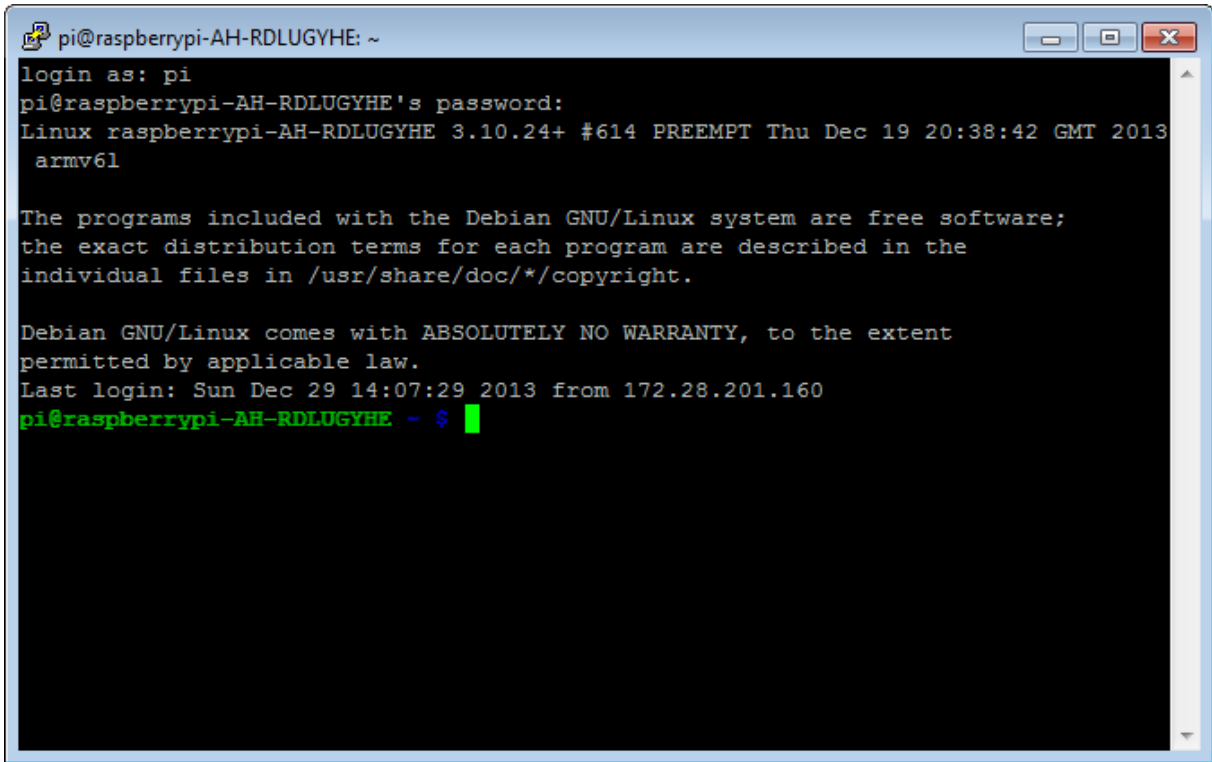
```
system(mypi, 'ls -al')
```

```
ans =
```

```
total 100
drwxr-xr-x 10 pi pi 4096 Nov 22 14:18 .
drwxr-xr-x 3 root root 4096 Sep 25 16:22 ..
-rw----- 1 pi pi 21712 Nov 13 17:40 .bash_history
-rw-r--r-- 1 pi pi 220 Sep 25 16:22 .bash_logout
-rw-r--r-- 1 pi pi 3243 Sep 25 16:22 .bashrc
drwxr-xr-x 4 pi pi 4096 Oct 1 18:17 .cache
drwxr-xr-x 6 pi pi 4096 Oct 2 12:01 .config
drwx----- 3 pi pi 4096 Oct 1 18:17 .dbus
drwxr-xr-x 2 pi pi 4096 Nov 13 17:30 Desktop
-rw-r--r-- 1 pi pi 35 Nov 13 17:41 .dmrc
drwx----- 2 pi pi 4096 Oct 1 18:17 .gvfs
drwxr-xr-x 3 pi pi 4096 Oct 2 14:46 MATLAB
-rw-r--r-- 1 pi pi 5781 Feb 3 2013 ocr_pi.png
-rw-r--r-- 1 pi pi 675 Sep 25 16:22 .profile
drwxrwxr-x 2 pi pi 4096 Mar 10 2013 python_games
drwxr-xr-x 8 pi pi 4096 Oct 2 12:41 wiringPi
-rw----- 1 pi pi 66 Nov 13 17:41 .Xauthority
-rw----- 1 pi pi 261 Nov 13 17:41 .xsession-errors
-rw----- 1 pi pi 449 Nov 13 17:40 .xsession-errors.old
```

Use the `openShell` function to open an SSH terminal that is connected to the Linux command-line interface on the Raspberry Pi hardware. Log in as the `pi` user. The default password for `pi` is `raspberry`.

```
mypi = raspi()
openShell(mypi)
```



```
pi@raspberrypi-AH-RDLUGYHE: ~
login as: pi
pi@raspberrypi-AH-RDLUGYHE's password:
Linux raspberrypi-AH-RDLUGYHE 3.10.24+ #614 PREEMPT Thu Dec 19 20:38:42 GMT 2013
  armv61

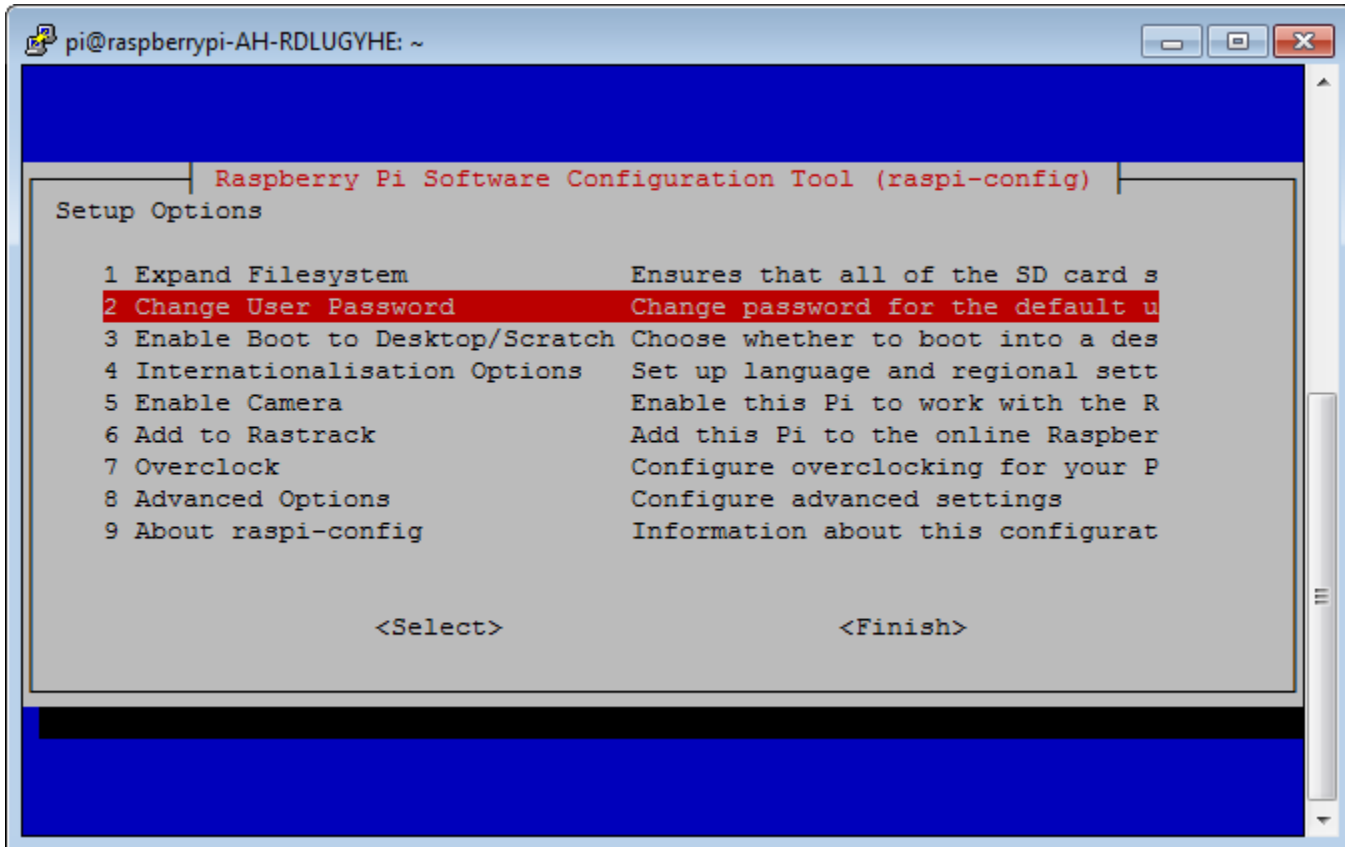
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Dec 29 14:07:29 2013 from 172.28.201.160
pi@raspberrypi-AH-RDLUGYHE ~ $
```

It is a good security practice to change the default password. Use the SSH terminal window to run the raspi-config utility.

```
sudo raspi-config
```

Select 2 Change User Password and change the password.



After changing the password, clear the mypi handle and any handles that you created using mypi. Then, use raspi to create a new connection based on the new password.

```
clear mypi
mypi = raspi('172.28.201.137', 'pi', 'newpassword')
```

For more information, see “Linux”.

Troubleshoot Running Linux Commands on Raspberry Pi Hardware

You changed the pi user's password, and now the `system` function does not work. The current `mypi` handle uses the old password. To regain control of the MATLAB Command Window, press **Ctrl+C**. Clear the `mypi` handle and any other handle based on the old password. Create a new connection using `raspi` with the arguments that specify the new password. Use the `system` function again.

For more information, see: "Linux".

Management of Raspberry Pi Files

You can manage files on the Raspberry Pi hardware. The `getFile` function downloads files from the Raspberry Pi hardware. The `putFile` function uploads files to the Raspberry Pi hardware. The `deleteFile` function deletes files on the Raspberry Pi hardware. All of these functions:

- Use SCP and SSH protocols, which provide encrypted communications.
- Use the current folder in MATLAB as the default file location in on the host computer.
- Use the present working directory (`pwd`) as the default file location on the Raspberry Pi hardware.
- Accept absolute or relative paths to specify nondefault file locations.
- Let you specify the destination file name.

For more information, see:

- “Linux”.
- http://en.wikipedia.org/wiki/Secure_Shell
- http://en.wikipedia.org/wiki/Secure_copy

Manage Raspberry Pi Files

You can download files from, upload files to, and delete files from the Raspberry Pi hardware.

To download a file from the Raspberry Pi hardware to your host computer, use the `getFile` function. Use the second argument to specify the path and name of the file.

```
system(mypi, 'ls')  
getFile(mypi, '/home/pi/.profile')
```

By default, `getFile` saves the file to the current folder in MATLAB. You can use a third argument to specify a download folder on your host computer.

```
getFile(mypi, '/home/pi/.profile', 'C:\Users\myusername\Desktop')
```

To upload a file to the Raspberry Pi hardware, use the `putFile` function.

```
putFile(mypi, 'C:\Users\myusername\Desktop\.profile', '/home/pi/')
```

If you use the Camera Board to record video, download the video file. Then, delete the file from the Raspberry Pi hardware.

```
getFile(mypi, 'myvideo.mp4', 'C:\MATLAB');  
deleteFile(mypi, 'myvideo.mp4')
```

For more information, see “Linux”.

Troubleshoot Managing Raspberry Pi Files

You changed the pi user's password, and now the `getFile`, `putFile`, and `deleteFile` functions do not work. The current `mypi` handle uses the old password. To regain control of the MATLAB Command Window, press **Ctrl+C**. Clear the `mypi` handle and any other handle based on the old password. Create a new connection using `raspi` with arguments that specify the new password. Use the `getFile`, `putFile`, or `deleteFile` function again.

For more information, see: "Linux".

Webcam Support in MATLAB

- “Webcam Acquisition Overview” on page 12-2
- “Connecting to Webcams” on page 12-4
- “Acquiring Images from Webcams” on page 12-6
- “Setting Properties for Webcam Acquisition” on page 12-17
- “Installing the Webcam Support Package” on page 12-22

Webcam Acquisition Overview

In this section...
“Webcam Support” on page 12-2
“Supported Platforms” on page 12-3

Webcam Support

You can use MATLAB Webcam support to bring live images from any USB Video Class (UVC) compliant Webcam into MATLAB. This includes Webcams that may be built into laptops or other devices, as well as Webcams that plug into your computer via a USB port.

Using simple MATLAB functions, you can detect connected Webcams, acquire individual snapshots from a Webcam, and optionally set up a loop for acquiring images. The `webcamlist` function allows you to detect the connected Webcams. The `webcam` function creates the Webcam object that is used to acquire images. And the `snapshot` function returns a single image from the camera. You can also preview your image and set properties for the image.

For more information, see

- “Connecting to Webcams” on page 12-4 for how to use the `webcamlist` function to detect your cameras
- “Acquiring Images from Webcams” on page 12-6 for how to acquire live images from your camera into MATLAB
- “Setting Properties for Webcam Acquisition” on page 12-17 for how to set Webcam-specific or camera-specific properties for the acquisition

Note Webcam support is available only through Hardware Support Packages. You must download and install the necessary files using the Support Package Installer. For instructions, see “Installing the Webcam Support Package” on page 12-22.

Supported Platforms

The MATLAB Webcam support can be used on the following platforms:

- Microsoft Windows 32-bit and 64-bit (Windows 7 or later)
- Mac OS X 64-bit
- Linux

Connecting to Webcams

Use the `webcamlist` function to return the list of available UVC-compliant Webcams connected to your system. The function returns a cell array of camera names. The list supports the plug and play scenario, where using the `webcamlist` function again in the same MATLAB session returns an updated list of cameras if you plug in different cameras during the session.

If you have a single Webcam connected to your system, the output shows one camera:

```
webcamlist

ans =

    'Logitech USB Camera'
```

If you have multiple Webcams connected to your system, the output shows all the cameras in a cell array:

```
webcamlist

ans =

    'Dell Camera C250'
    'Logitech USB Camera'
```

In this case `webcamlist` detects the built-in Webcam in the Dell® computer, and a connected USB Webcam.

If you have two cameras connected by USB ports, the output is:

```
webcamlist

ans =

    'Logitech Webcam 250'
    'Logitech Webcam Pro 9000'
```

The name of the camera that is shown in this output, for example 'Logitech Webcam 250', is the name you can use to create the Webcam object in order to acquire images.

For more information, see:

- “Acquiring Images from Webcams” on page 12-6 for how to acquire live images from your camera into MATLAB
- “Setting Properties for Webcam Acquisition” on page 12-17 for how to set object-specific or device-specific properties for the acquisition

Note Webcam support is available only through Hardware Support Packages. You must download and install the necessary files using the Support Package Installer. For instructions, see “Installing the Webcam Support Package” on page 12-22.

Acquiring Images from Webcams

In this section...
“Creating a Webcam Object” on page 12-6
“Acquiring Webcam Images” on page 12-10
“Acquiring Webcam Images in a Loop” on page 12-14
“Supported Functions for Webcam” on page 12-16

Creating a Webcam Object

Use the `webcam` function to create a Webcam object. You can use it in one of three ways:

- Connect to the first or only camera, by using no input arguments

- Specify a camera by name, by using the Webcam name (as a string) as an input argument

- Specify a camera by the list order, by using an index number as the input argument

Note Webcam support is available only through a Hardware Support Package. You must download and install the necessary files using the Support Package Installer. For instructions, see “Installing the Webcam Support Package” on page 12-22.

Find the name of your camera by using the `webcamlist` function. Run `webcamlist` first to make sure that MATLAB can discover your camera(s). In this example, it discovers the built-in Webcam in the Dell computer, and a connected Logitech® Webcam.

```
webcamlist
```

```
ans =
```

```
    'Logitech Webcam 250'
```

```
    'Dell Camera C250'
```

No Input Argument

If you use the `webcam` function with no input argument, it creates the object and connects to the first camera returned by `webcamlist`. This will be the first camera shown on the list if you have multiple cameras. If you only have one camera connected to your system, it will use that one. So in the example shown above, it will create the object using the Logitech camera, since that appears in the `webcamlist` output first.

```
% Use cam as the name of the object.

cam = webcam

cam =

webcam with properties:

                Name: 'Logitech Webcam 250'
            Resolution: '640x480'
    AvailableResolutions: {1x11 cell}
                Exposure: -4
                    Gain: 253
                Saturation: 32
            WhiteBalance: 8240
            ExposureMode: 'auto'
                Sharpness: 48
                Brightness: 128
    BacklightCompensation: 1
                Contrast: 32
```

You can see that it created the object and connected to the Logitech Webcam.

Index as Input Argument

If you use the `webcam` function with an index as the input argument, it creates the object corresponding to that index and connects to that camera. The index corresponds to the order of cameras in the cell array returned by `webcamlist` when you have multiple cameras connected. So in the example shown above, device 1 is the Logitech camera and device 2 is the built-in Dell Webcam.

```
% Use cam as the name of the object. Use 2 to connect to the Dell camera.

cam = webcam(2)

cam =

webcam with properties:

                Name: 'Dell Camera C250'
```

```

        Resolution: '320x240'
AvailableResolutions: ('320x240' '160x120' '80x60')
        Brightness: 128
        Contrast: 32
        Gain: 0

```

You can see that it created the object and connected to the Dell Webcam. If you only have one camera, you do not need to use the index. You can use the `webcam` function with no input argument and it creates the object with the single camera that is connected. The index is useful when you have multiple cameras.

Camera Name as Input Argument

If you use the `webcam` function with the name of the camera (as a string) as the input argument, it creates the object and connects to the camera with that name. You can use the exact name that is displayed by the `webcamlist` function. In the example above it would be 'Logitech Webcam 250'. You can also use a shortened version of the name, for example, the brand of the camera. In this case you could simply use 'Logitech' and it would connect to the Logitech Webcam.

```
% Use cam as the name of the object. Use 'Logitech' to connect to the Logitech camera.
```

```
cam = webcam('Logitech')
```

```
cam =
```

```
webcam with properties:
```

```

        Name: 'Logitech Webcam 250'
        Resolution: '640x480'
AvailableResolutions: {1x11 cell}
        Exposure: -4
        Gain: 253
        Saturation: 32
        WhiteBalance: 8240
        ExposureMode: 'auto'
        Sharpness: 48
        Brightness: 128

```

```
BacklightCompensation: 1
Contrast: 32
```

You can see that it created the object and connected to the Logitech Webcam.

When the `webcam` object is created, it connects to the camera and establishes exclusive access to the camera and starts streaming data from the camera. You can then preview the data and acquire images using the `snapshot` function, as described in the next section.

Acquiring Webcam Images

Acquiring images from Webcams and bringing them into MATLAB is a simple process. The general procedure is to make sure your camera is connected, create a `webcam` object, preview the image, and acquire snapshots. This typical workflow is outlined here.

- 1 See what cameras are connected to your system and make sure MATLAB can detect them.

```
webcamlist

ans =

    'Logitech Webcam 250'
    'Dell Camera C250'
```

The output is a list of any Webcams that are connected to your system. In this example, it discovers a built-in Webcam in the Dell computer, and a connected Logitech Webcam.

- 2 Create a webcam object called `cam`, using the Logitech camera.

```
cam = webcam('Logitech')
```

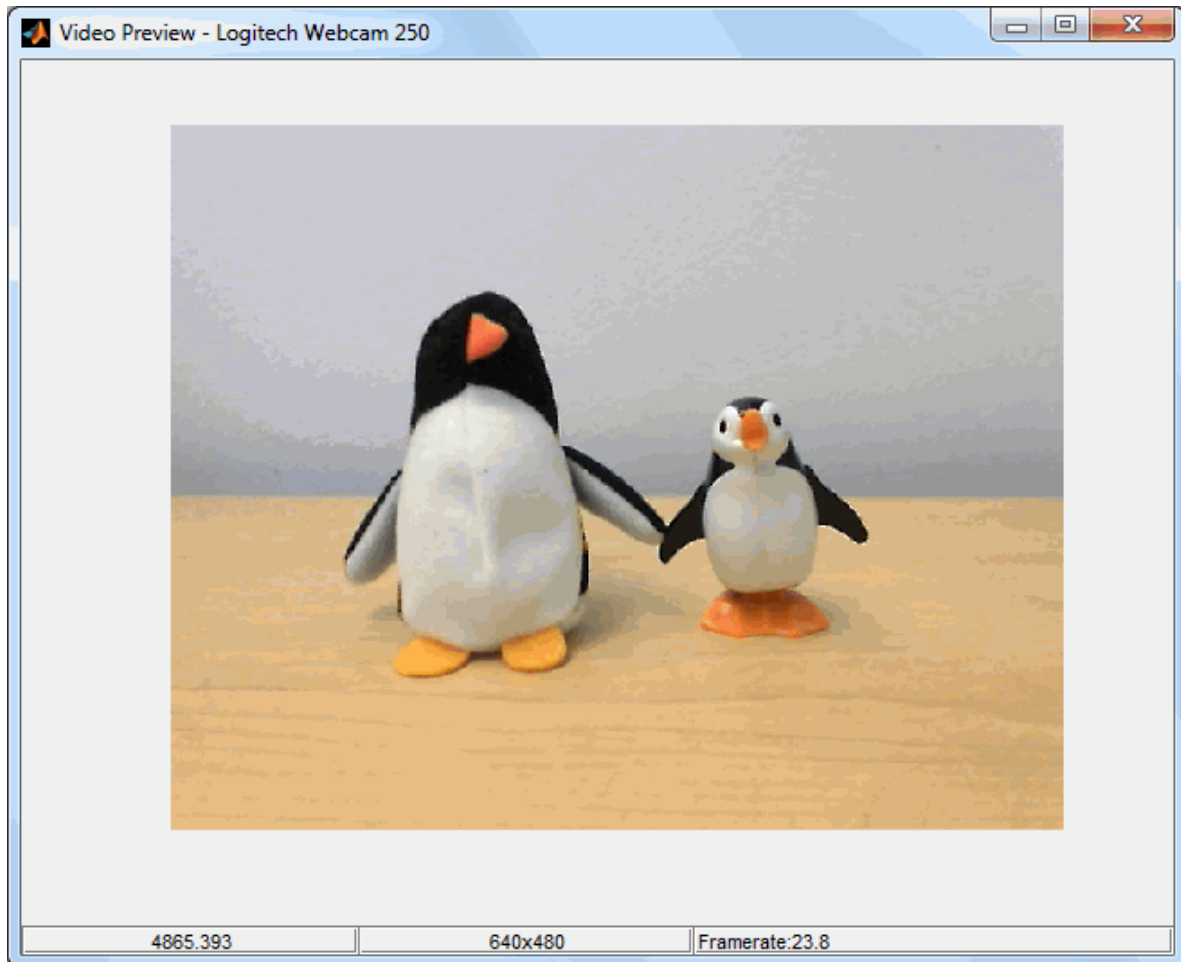
```
cam =
```

```
webcam with properties:
```

```
                Name: 'Logitech Webcam 250'  
            Resolution: '640x480'  
    AvailableResolutions: {1x11 cell}  
                Exposure: -4  
                  Gain: 253  
                Saturation: 32  
            WhiteBalance: 8240  
            ExposureMode: 'auto'  
                Sharpness: 48  
                Brightness: 128  
    BacklightCompensation: 1  
                Contrast: 32
```

- 3 Preview the image. The size of the preview image is determined by the value of the `resolution` property. The preview window shows a live RGB image from the Webcam. The preview window also displays the camera name, resolution, frame rate, and the timestamp in seconds. Timestamp is the elapsed time since the object was created. To preview your image, call the `preview` function on the object name, which is `cam` in this example.

```
preview(cam)
```



The banner of the preview window shows the camera name. The lower portion of the window shows the timestamp in seconds, resolution, and the frame rate in frames per second. Timestamp is the elapsed time since the object was created.

The preview updates dynamically, so if you change a property while previewing, the image changes to reflect the property change.

- 4 Set any properties that you need to change. For example, you might want to change the resolution.

First you can see the resolutions your camera supports using the `AvailableResolutions` property.

```
cam.AvailableResolutions
```

```
ans =
```

```
Columns 1 through 6
```

```
'640x480' '160x90' '160x100' '160x120' '176x144' '320x180'
```

```
Columns 7 through 11
```

```
'320x200' '320x240' '352x288' '640x360' '640x400'
```

Change the resolution.

```
set(cam, 'Resolution', '320x240');
```

For information on which properties you can set for Webcams and how to set them, see “Setting Properties for Webcam Acquisition” on page 12-17.

- 5 You can close the preview at any time using the `closePreview` function.

```
closePreview(cam)
```

If you do not explicitly close the preview, it closes when you clear the `webcam` object.

- 6 Acquire a single image from the camera using the `snapshot` function and assign it to the variable `img`.

```
img = snapshot(cam);
```

- 7 Display the acquired image.

```
imshow(img)
```

The `imshow` function is part of the Image Processing Toolbox™. If you do not have that product, you can use the `image` function that is part of MATLAB.

```
image(img)
```

- 8 Clean up by clearing the object.

```
clear('cam');
```

Acquiring Webcam Images in a Loop

The `snapshot` function acquires a single image from a Webcam. If you want to acquire images in a loop, you can do that with some extra programming.

This example uses MATLAB and Image Processing Toolbox to find circles in a video stream from a Webcam.

- 1 Create a webcam object called `cam`, using a Logitech Webcam.

```
cam = webcam('Logitech')
```

```
cam =
```

```
webcam with properties:
```

```
                Name: 'Logitech Webcam 250'  
            Resolution: '640x480'  
AvailableResolutions: {1x11 cell}  
            Exposure: -4  
                Gain: 253  
            Saturation: 32  
        WhiteBalance: 8240  
    ExposureMode: 'auto'  
            Sharpness: 48  
            Brightness: 128  
BacklightCompensation: 1  
                Contrast: 32
```

- 2 Preview the image.

```
preview(cam)
```

- 3** Set any properties that you need to change. For example, you might want to change the brightness, if the camera supports that device-specific property.

```
set(cam, 'Brightness', 150);
```

For more information on what properties you can set for Webcams and how to set the properties, see “Setting Properties for Webcam Acquisition” on page 12-17.

- 4** Create the loop and perform processing.

```
for idx = 1:100
    % Acquire a single image.
    rgbImage = snapshot(cam);

    % Convert RGB to grayscale.
    grayImage = rgb2gray(rgbImage);

    % Find circles.
    [centers, radii] = imfindcircles(grayImage, [60 80]);

    % Display the image.
    imshow(rgbImage);
    hold on;
    viscircles(centers, radii);
    drawnow
end
```

- 5** Clean up by clearing the object.

```
clear('cam');
```

Supported Functions for Webcam

You can use these functions with the MATLAB Webcam feature.

Function	Purpose
webcamlist	Returns list of Webcams that are connected to your system. <code>webcamlist</code>
webcam	Creates webcam object and connects to the single camera on your system. If you have multiple cameras and you use the webcam function with no input argument, it creates the object and connects it to the first camera it finds. <code>cam = webcam</code> For information on how to create the object with an input argument if you have multiple cameras connected, see “Creating a Webcam Object” on page 12-6.
preview	Preview the images from the Webcam. Use name of object as input argument, which is <code>cam</code> in this example. <code>preview(cam)</code>
snapshot	Acquire a single image from the Webcam. Use name of object as input argument, which is <code>cam</code> in this example. <code>img = snapshot(cam);</code>
closePreview	Close the preview window. <code>closePreview(cam)</code>
set/get	Set and get property values. Object name is the first argument, followed by property/value pairs, which can be strings or numerics. This example sets the camera’s resolution to the value shown. <code>set(cam, 'Resolution', '320x240');</code> For more information on how to set object- or device-specific properties for the acquisition and what properties can be set, see “Setting Properties for Webcam Acquisition” on page 12-17.

Setting Properties for Webcam Acquisition

You can set object-specific properties for the webcam object to use with any Webcam. You can also set device-specific properties for your specific Webcam, if supported by your device. You use the `set` function to set object-specific property values for the Webcam object, and either the `set` function or the dot notation to set device-specific properties. These two methods are described in the following two sections.

Note You can set device-specific properties only for Webcams connected to a Windows system.

Object-Specific Properties

You can use the `get` and `set` functions to display property values or set property values for the Webcam object. The preview window is dynamic, so if you set a property while previewing your image, you can see the change take effect.

Set properties after creating the Webcam object and before acquiring images.

To set an object-specific property, use the object name as the first argument, followed by property-value pairs, which can be strings or numerics. This example sets the camera resolution to the value shown for the webcam object `cam`.

```
set(cam, 'Resolution', '320x240');
```

You can use these webcam object-specific properties for any Webcam.

Object-Specific Property	Description
Name	A read-only property that specifies the camera name. It is dynamically populated and uses the name that is shown in the output of the <code>webcamlist</code> function. For example, 'Logitech Webcam 250'.
Resolution	Specifies the video resolution (width by height) of the incoming video stream of the current webcam object. Webcams typically support acquiring images at multiple resolutions, and you can change the resolution using this property and the object name. By default, we will select the default resolution of the camera. Use this syntax to change it: <code>set(cam, 'Resolution', '160x120');</code>
AvailableResolutions	Displays the list of all available resolutions for the selected Webcam. Use object name: <code>cam.AvailableResolutions</code> <code>ans =</code> <code>'320x240'</code> <code>'160x120'</code> <code>'80x60'</code>
FrameRate	Displays frames per second for the acquisition.

Device-Specific Properties

You can also set device-specific properties that are specific to your Webcam if your device allows for programmatic access. These properties vary depending on your device. See the table below for a list of the possible properties for a UVC compatible Webcam. Your camera may not have all of these. You can only set properties that your camera allows. See the properties for your camera by looking at the output when you create the webcam object. For example, the following shows the available properties for a Logitech Webcam.


```
cam = webcam('Logitech')

cam =

webcam with properties:

        Name: 'Logitech Webcam 250'
        Resolution: '640x480'
        AvailableResolutions: {1x11 cell}
        Exposure: -4
        Gain: 253
        Saturation: 32
        WhiteBalance: 8240
        ExposureMode: 'auto'
        Sharpness: 48
        Brightness: 128
        BacklightCompensation: 1
        Contrast: 32
```

Note that device-specific properties can only be set for Webcams connected to a Windows system.

To set a device-specific property, use the object name and property name in dot notation as the first argument, and the value you want to set as the second argument. This example sets the camera brightness to the value shown for the webcam object `cam`.

```
cam.Brightness = 150;
```

These are the properties that a UVC Webcam can have. Your specific camera may not have all of these. The camera could also have mode properties, which are not listed here. See your camera documentation for the list of properties your device supports.

Possible Device-Specific Properties	Description
BacklightCompensation	Configures backlight compensation modes to adjust the camera to capture images dependent on environmental conditions. Values are on and off.
Brightness	Indicates the brightness level, which adjusts for the amount of lighting on the image.
Contrast	Indicates contrast level, which adjusts for the difference between brightest and dimmest areas in the image.
ColorEnable	Specifies the color enable setting. Values are on and off.
Gain	Indicates a multiplier for the RGB color values. The value 0 is normal. Positive values are brighter and negative values are darker.
Gamma	Indicates gamma measurement.
Hue	Indicates hue setting, which adjusts the color tint of the image through the red-yellow-blue spectrums.
PowerLineFrequency	Option for reducing flicker caused by the frequency of a power line.
Saturation	Indicates saturation level, which adjusts the amount of color in the image.
Sharpness	Indicates sharpness level, which adjusts the clarity of the image.
WhiteBalance	Indicates color temperature in degrees Kelvin.
Pan	Camera control property for panning, in degrees.
Tilt	Camera control property for tilting, in degrees.
Roll	Camera control property for rolling, in degrees.

Possible Device-Specific Properties	Description
Zoom	Camera control property for zooming, in millimeters.
Exposure	Camera control property for specifying exposure, in log base 2 seconds ($1/2^n$ seconds).
Iris	Camera control property for specifying iris setting, in units of f-stop x 10.
Focus	Camera control property for setting focus, as the distance to the optimally focused target, in millimeters.

Note Webcam support is available only through a Hardware Support Package. You must download and install the necessary files using the Support Package Installer. For instructions, see “Installing the Webcam Support Package” on page 12-22.

Installing the Webcam Support Package

You can use the MATLAB Webcam support to bring live images from any USB Video Class (UVC) Webcam into MATLAB. This includes Webcams that may be built into laptops or other devices, as well as Webcams that plug into your computer via a USB port. To use the Webcam feature, you must install the Webcams Support package.

The Webcam support is available through the Hardware Support Packages. Using this installation process, you download and install the following file(s) on your host computer:

- MATLAB files for Webcam support
- An example that shows how to acquire images using a Webcam

Installing the Support Package

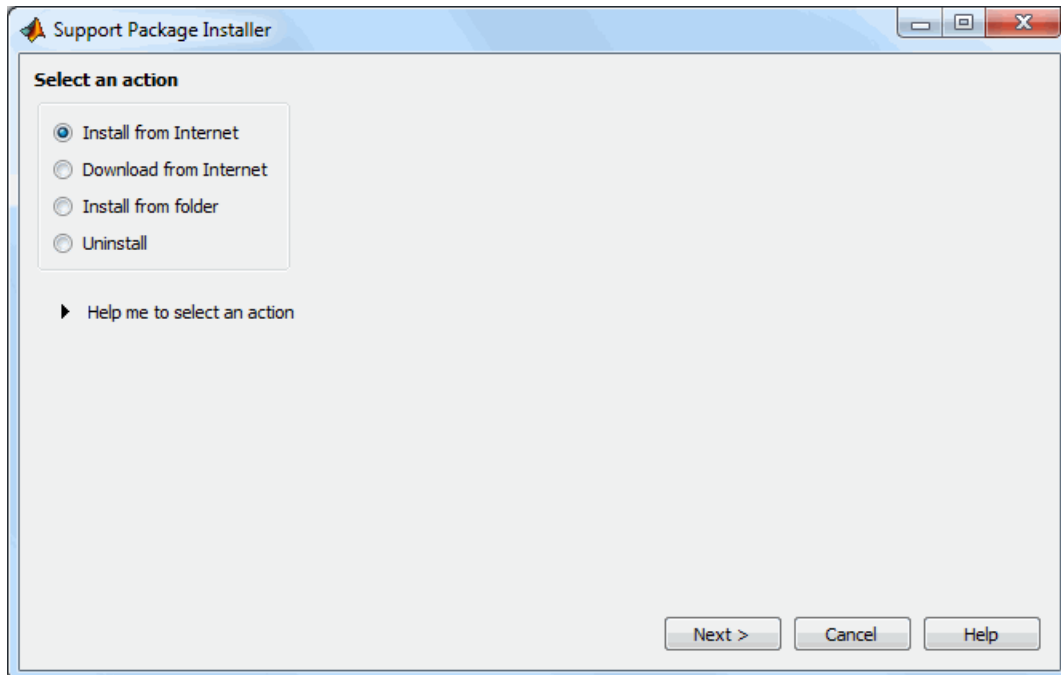
1 In MATLAB type:

```
supportPackageInstaller
```

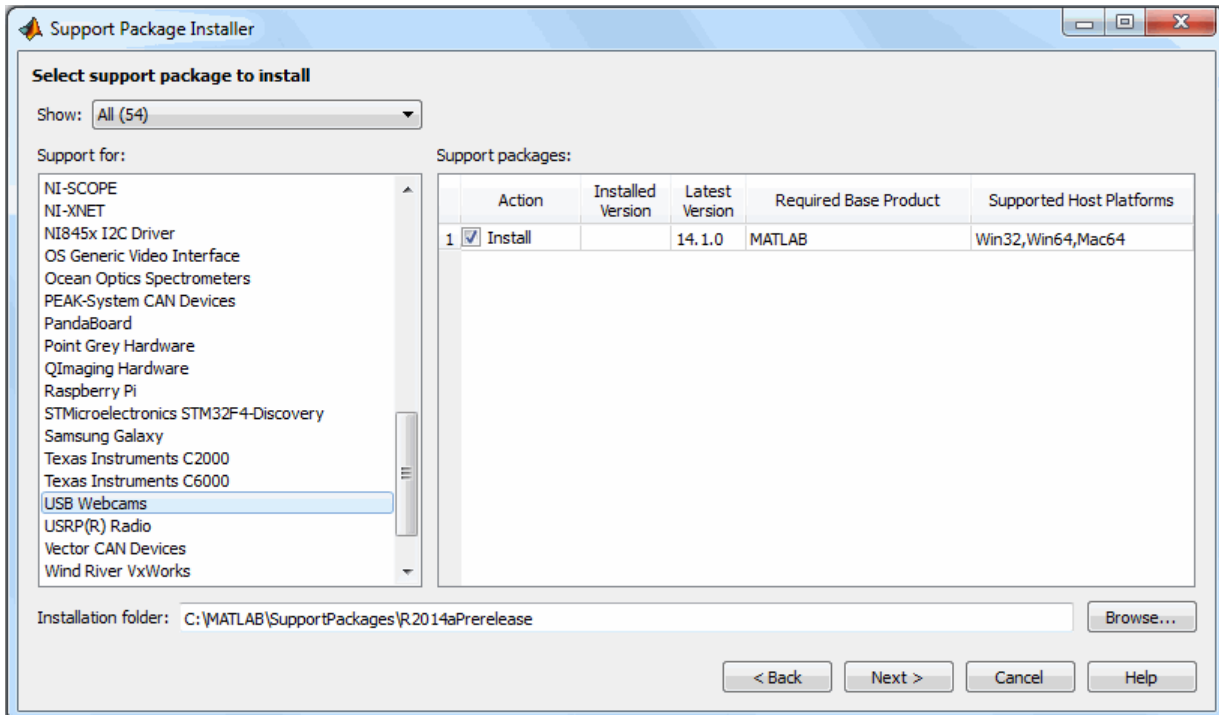
to open the Support Package Installer.

You can also open the installer from MATLAB by selecting **Home > Resources > Add-Ons > Get Hardware Support Packages**.

2 On the **Select an action** screen, select **Install from Internet** and then click **Next**. This option is selected by default. Support Package Installer downloads and installs the support package and third-party software from the Internet.



- 3 On the **Select support package to install** screen, select **USB Webcams** from the list.

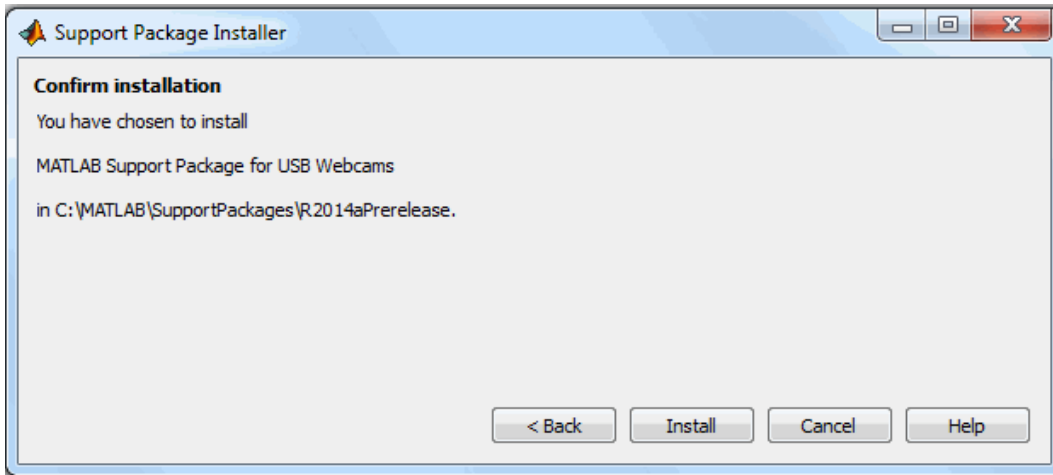


Accept or change the **Installation folder** and click **Next**.

Note You must have write privileges for the Installation folder.

- 4 If you are prompted to log in to your MathWorks® account, click **Log In** to continue.
- 5 On the **MATHWORKS AUXILIARY SOFTWARE LICENSE AGREEMENT** screen, select the **I accept** check box and click **Next**.

- 6 On the **Confirm installation** screen, Support Package Installer confirms that you are installing the MATLAB Support Package for USB Webcams, and lists the installation location. Confirm your selection and click **Install**.



Support Package Installer displays a progress bar while it downloads and installs the Webcam support package.

- 7 After the installation is complete you will see a confirmation message on the Support Package Installer **Install/update complete** screen. Click **Finish** to close the Support Package Installer.
- 8 If you selected the **Show support package examples** option (recommended), the Help displays the example.
- 9 You will be prompted to restart your computer. You must restart for the installation to be complete. Click **OK** and then restart your computer.