Image Acquisition Toolbox™

User's Guide

MATLAB®

R2023a

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

Image Acquisition Toolbox™ User's Guide

Revision History

March 2003	First printing	New for Version 1.0 (Release 13+)
September 2003	Online only	Revised for Version 1.1 (Release 13SP1)
June 2004	Online only	Revised for Version 1.5 (Release 14)
July 2004	Online only	Revised for Version 1.6 (Release 14+)
October 2004	Online only	Revised for Version 1.7 (Release 14SP1)
March 2005	Online only	Revised for Version 1.8 (Release 14SP2)
March 2005	Second printing	Minor Revision for Version 1.8
August 2005	Third printing	Minor Revision for Version 1.8
September 2005	Online only	Revised for Version 1.9 (Release 14SP3)
March 2006	Fourth printing	Revised for Version 1.10 (Release 2006a)
September 2006	Online only	Revised for Version 2.0 (Release 2006b)
March 2007	Online only	Revised for Version 2.1 (Release 2007a)
September 2007	Fifth printing	Revised for Version 3.0 (Release 2007b)
March 2008	Online only	Revised for Version 3.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.2 (Release 2008b)
March 2009	Online only	Revised for Version 3.3 (Release 2009a)
September 2009	Online only	Revised for Version 3.4 (Release 2009b)
March 2010	Online only	Revised for Version 3.5 (Release 2010a)
September 2010	Online only	Revised for Version 4.0 (Release 2010b)
April 2011	Online only	Revised for Version 4.1 (Release 2011a)
September 2011	Online only	Revised for Version 4.2 (Release 2011b)
March 2012	Online only	Revised for Version 4.3 (Release 2012a)
September 2012	Online only	Revised for Version 4.4 (Release 2012b)
March 2013	Online only	Revised for Version 4.5 (Release 2013a)
September 2013	Online only	Revised for Version 4.6 (Release 2013b)
March 2014	Online only	Revised for Version 4.7 (Release 2014a)
October 2014	Online only	Revised for Version 4.8 (Release 2014b)
March 2015	Online only	Revised for Version 4.9 (Release 2015a)
September 2015	Online only	Revised for Version 4.10 (Release 2015b)
October 2015	Online only	Rereleased for Version 4.9.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 5.0 (Release 2016a)
September 2016	Online only	Revised for Version 5.1 (Release 2016b)
March 2017	Online only	Revised for Version 5.2 (Release 2017a)
September 2017	Online only	Revised for Version 5.3 (Release 2017b)
March 2018	Online only	Revised for Version 5.4 (Release 2018a)
September 2018	Online only	Revised for Version 5.5 (Release 2018b)
March 2019	Online only	Revised for Version 6.0 (Release 2019a)
September 2019	Online only	Revised for Version 6.1 (Release 2019b)
March 2020	Online only	Revised for Version 6.2 (Release 2020a)
September 2020	Online only	Revised for Version 6.3 (Release 2020b)
March 2021	Online only	Revised for Version 6.4 (Release 2021a)
September 2021	Online only	Revised for Version 6.5 (Release 2021b)
March 2022	Online only	Revised for Version 6.6 (Release 2022a)
September 2022	Online only	Revised for Version 6.7 (Release 2022b)
March 2023	Online only	Revised for Version 6.7.1 (Release 2023a)

Contents

4 Image Acquisition Support Packages

5 Connecting to Hardware

6 Acquiring Image Data

7 Working with Acquired Image Data

8 Using Events and Callbacks

9 Using the From Video Device Block in Simulink

10 Configuring GigE Vision Devices

11 Using the GigE Vision Interface

Getting Started

The best way to learn about Image Acquisition Toolbox capabilities is to look at a simple example. This chapter introduces the toolbox and illustrates the basic steps to create an image acquisition application by implementing a simple motion detection application. The example cross-references other sections that provide more details about relevant concepts.

Image Acquisition Toolbox Product Description

Acquire images and video from industry-standard hardware

Image Acquisition Toolbox provides functions and blocks for connecting cameras to MATLAB® and Simulink®. It includes a MATLAB app that lets you interactively detect and configure hardware properties. You can then generate equivalent MATLAB code to automate your acquisition in future sessions. The toolbox enables acquisition modes such as processing in-the-loop, hardware triggering, background acquisition, and synchronizing acquisition across multiple devices.

Image Acquisition Toolbox supports all major standards and hardware vendors, including USB3 Vision, GigE Vision®, and GenICam™ GenTL. You can connect to machine vision cameras and frame grabbers, as well as high-end scientific and industrial devices.

Product Overview

In this section...

Introduction

The Image Acquisition Toolbox software is a collection of functions that extend the capability of the MATLAB numeric computing environment. The toolbox supports a wide range of image acquisition operations, including:

- Acquiring images through many types of image acquisition devices, from professional grade frame grabbers to USB-based webcams
- Viewing a preview of the live video stream
- Triggering acquisitions (includes external hardware triggers)
- Configuring callback functions that execute when certain events occur
- Bringing the image data into the MATLAB workspace

Many of the toolbox functions are MATLAB files. You can view the MATLAB code for these functions using this statement:

```
type function_name
```

You can extend Image Acquisition Toolbox capabilities by writing your own MATLAB files, or by using the toolbox in combination with other toolboxes, such as the Image Processing Toolbox™ software and the Data Acquisition Toolbox™ software.

The Image Acquisition Toolbox software also includes a Simulink block, called From Video Device, that you can use to bring live video data into a model.

Installation and Configuration Notes

To determine if the Image Acquisition Toolbox software is installed on your system, type this command at the MATLAB prompt:

```
ver
```

When you enter this command, MATLAB displays information about the version of MATLAB you are running, including a list of all toolboxes installed on your system and their version numbers.

For information about installing the toolbox, see the MATLAB Installation Guide.

For the most up-to-date information about system requirements, see the system requirements page, available in the products area of the MathWorks website.

Note With previous versions of the Image Acquisition Toolbox, the files for all of the adaptors were included in your installation. Starting with version R2014a, each adaptor is available separately through support packages. In order to use the Image Acquisition Toolbox, you must install the adaptor that your camera uses. See “Image Acquisition Support Packages for Hardware Adaptors” on page 4-2 for information about installing the adaptors using MATLAB Add-Ons.

The Image Processing Toolbox Software Required to Use the Image Acquisition Toolbox Software

The Image Acquisition Toolbox now requires you to have a license for the Image Processing Toolbox product starting in R2008b.

If you already have the Image Processing Toolbox product, you do not need to do anything.

If you do not have the Image Processing Toolbox product, the Image Acquisition Toolbox software R2008a and earlier will continue to work. If you want to use R2008b or future releases, and you have a current active license for the Image Acquisition Toolbox software, you can download the Image Processing Toolbox product for free. New customers will need to purchase both products to use the Image Acquisition Toolbox product.

If you have any questions, please contact MathWorks customer service.

Related Products

MathWorks provides several products that are relevant to the kinds of tasks you can perform with the Image Acquisition Toolbox software and that extend the capabilities of MATLAB. For information about these related products, see Image Acquisition Toolbox on the MathWorks website.

Supported Hardware

The list of hardware that the Image Acquisition Toolbox software supports can change in each release, since hardware support is frequently added. The MathWorks website is the best place to check for the most up to date listing. To see the full list of hardware that the toolbox supports, see Hardware Support from Image Acquisition Toolbox.

Image Acquisition Tool (GUI)

The functionality of the Image Acquisition Toolbox software is available in a desktop application. You connect directly to your hardware in the tool and can set acquisition parameters, and preview and acquire image data. You can log the data to MATLAB in several formats, and also generate a VideoWriter file, right from the tool.

To open the tool, type `imaqtool` at the MATLAB command line, or select **Image Acquisition** on the **Apps** tab in MATLAB. The tool has extensive Help in the desktop. As you click in different panes of the user interface, the relevant Help appears in the **Image Acquisition Tool Help** pane.

Most of the User's Guide describes performing tasks using the toolbox via the MATLAB command line. To learn how to use the desktop tool, see “Getting Started with the Image Acquisition Tool” on page 3-35.

Getting Started Doing Image Acquisition Programmatically

In this section...

Overview

This section illustrates the basic steps required to create an image acquisition application by implementing a simple motion detection application. The application detects movement in a scene by performing a pixel-to-pixel comparison in pairs of incoming image frames. If nothing moves in the scene, pixel values remain the same in each frame. When something moves in the image, the application displays the pixels that have changed values.

The example highlights how you can use the Image Acquisition Toolbox software to create a working image acquisition application with only a few lines of code.

Note To run the sample code in this example, you must have an image acquisition device connected to your system. The device can be a professional grade image acquisition device, such as a frame grabber, or a generic Microsoft® Windows® image acquisition device, such as a webcam. The code can be used with various types of devices with only minor changes.

Note With previous versions of the Image Acquisition Toolbox, the files for all of the adaptors were included in your installation. Starting with version R2014a, each adaptor is available separately through support packages. In order to use the Image Acquisition Toolbox, you must install the adaptor that your camera uses. See "Image Acquisition Support Packages for Hardware Adaptors" on page 4-2 for information about installing the adaptors using MATLAB Add-Ons.

To use the Image Acquisition Toolbox software to acquire image data, you must perform the following basic steps.

Step	Description
Step 1 on page 1-7:	Install and configure your image acquisition device
Step 2: on page 1-7	Retrieve information that uniquely identifies your image acquisition device to the Image Acquisition Toolbox software
Step 3 on page 1-9:	Create a video input object

Step	Description
Step 4 on page 1-10:	Preview the video stream (Optional)
Step 5: on page 1-11	Configure image acquisition object properties (Optional)
Step 6 on page 1-14:	Acquire image data
Step 7 on page 1-16:	Clean up

Step 1: Install Your Image Acquisition Device

Follow the setup instructions that come with your image acquisition device. Setup typically involves:

- Installing the frame grabber board in your computer.
- Installing any software drivers required by the device. These are supplied by the device vendor.
- Connecting a camera to a connector on the frame grabber board.
- Verifying that the camera is working properly by running the application software that came with the camera and viewing a live video stream.

Generic Windows image acquisition devices, such as webcams and digital video camcorders, typically do not require the installation of a frame grabber board. You connect these devices directly to your computer via a USB or FireWire port.

After installing and configuring your image acquisition hardware, start MATLAB on your computer by double-clicking the icon on your desktop. You do not need to perform any special configuration of MATLAB to perform image acquisition.

Step 2: Retrieve Hardware Information

In this step, you get several pieces of information that the toolbox needs to uniquely identify the image acquisition device you want to access. You use this information when you create an image acquisition object, described in “Step 3: Create a Video Input Object” on page 1-9.

The following table lists this information. You use the `imaqhwinfo` function to retrieve each item.

Device Information	Description
Adaptor name	An *adaptor* is the software that the toolbox uses to communicate with an image acquisition device via its device driver. The toolbox includes adaptors for certain vendors of image acquisition equipment and for particular classes of image acquisition devices. See “Determining the Adaptor Name” on page 1-8 for more information.

Device Information	Description
Device ID	The *device ID* is a number that the adaptor assigns to uniquely identify each image acquisition device with which it can communicate. See “Determining the Device ID” on page 1-8 for more information. **Note** Specifying the device ID is optional; the toolbox uses the first available device ID as the default.
Video format	The *video format* specifies the image resolution (width and height) and other aspects of the video stream. Image acquisition devices typically support multiple video formats. See “Determining the Supported Video Formats” on page 1-9 for more information. **Note** Specifying the video format is optional; the toolbox uses one of the supported formats as the default.

Determining the Adaptor Name

To determine the name of the adaptor, enter the `imaqhwinfo` function at the MATLAB prompt without any arguments.

```
imaqhwinfo
ans =

    InstalledAdaptors: {'dcam'  'winvideo'}
        MATLABVersion: '7.4 (R2007a)'
          ToolboxName: 'Image Acquisition Toolbox'
       ToolboxVersion: '2.1 (R2007a)'
```

In the data returned by `imaqhwinfo`, the `InstalledAdaptors` field lists the adaptors that are available on your computer. In this example, `imaqhwinfo` found two adaptors available on the computer: `'dcam'` and `'winvideo'`. The listing on your computer might contain only one adaptor name. Select the adaptor name that provides access to your image acquisition device. For more information, see “Determining the Device Adaptor Name” on page 5-2.

Determining the Device ID

To find the device ID of a particular image acquisition device, enter the `imaqhwinfo` function at the MATLAB prompt, specifying the name of the adaptor as the only argument. (You found the adaptor name in the first call to `imaqhwinfo`, described in “Determining the Adaptor Name” on page 1-8.) In the data returned, the `DeviceIDs` field is a cell array containing the device IDs of all the devices accessible through the specified adaptor.

Note This example uses the DCAM adaptor. You should substitute the name of the adaptor you would like to use.

```
info = imaqhwinfo('dcam')
info =

       AdaptorDllName: [1x77 char]
    AdaptorDllVersion: '2.1 (R2007a)'
```

```
          AdaptorName: 'dcam'
            DeviceIDs: {[1]}
           DeviceInfo: [1x1 struct]
```

Determining the Supported Video Formats

To determine which video formats an image acquisition device supports, look in the `DeviceInfo` field of the data returned by `imaqhwinfo`. The `DeviceInfo` field is a structure array where each structure provides information about a particular device. To view the device information for a particular device, you can use the device ID as a reference into the structure array. Alternatively, you can view the information for a particular device by calling the `imaqhwinfo` function, specifying the adaptor name and device ID as arguments.

To get the list of the video formats supported by a device, look at `SupportedFormats` field in the device information structure. The `SupportedFormats` field is a cell array of character vectors where each character vector is the name of a video format supported by the device. For more information, see “Determining Supported Video Formats” on page 5-4.

```
dev_info = imaqhwinfo('dcam',1)

dev_info = 

          DefaultFormat: 'F7_Y8_1024x768'
    DeviceFileSupported: 0
             DeviceName: 'XCD-X700  1.05'
               DeviceID: 1
  VideoInputConstructor: 'videoinput('dcam', 1)'
 VideoDeviceConstructor: 'imaq.VideoDevice('dcam', 1)'
       SupportedFormats: {'F7_Y8_1024x768'  'Y8_1024x768'}
```

Step 3: Create a Video Input Object

In this step you create the video input object that the toolbox uses to represent the connection between MATLAB and an image acquisition device. Using the properties of a video input object, you can control many aspects of the image acquisition process. For more information about image acquisition objects, see “Creating Image Acquisition Objects” on page 5-6.

To create a video input object, use the `videoinput` function at the MATLAB prompt. The `DeviceInfo` structure returned by the `imaqhwinfo` function contains the default `videoinput` function syntax for a device in the `VideoInputConstructor` field. For more information the device information structure, see “Determining the Supported Video Formats” on page 1-9.

The following example creates a video input object for the DCAM adaptor. Substitute the adaptor name of the image acquisition device available on your system.

```
vid = videoinput('dcam',1,'Y8_1024x768')
```

The `videoinput` function accepts three arguments: the adaptor name, device ID, and video format. You retrieved this information in step 2. The adaptor name is the only required argument; the `videoinput` function can use defaults for the device ID and video format. To determine the default video format, look at the `DefaultFormat` field in the device information structure. See “Determining the Supported Video Formats” on page 1-9 for more information.

Instead of specifying the video format, you can optionally specify the name of a device configuration file, also known as a camera file. Device configuration files are typically supplied by frame grabber vendors. These files contain all the required configuration settings to use a particular camera with the device. See “Using Device Configuration Files (Camera Files)” on page 5-9 for more information.

Viewing the Video Input Object Summary

To view a summary of the video input object you just created, enter the variable name `vid` at the MATLAB command prompt. The summary information displayed shows many of the characteristics of the object, such as the number of frames that will be captured with each trigger, the trigger type, and the current state of the object. You can use video input object properties to control many of these characteristics. See “Step 5: Configure Object Properties (Optional)” on page 1-11 for more information.

```
vid

Summary of Video Input Object Using 'XCD-X700  1.05'.

    Acquisition Source(s):  input1 is available.

   Acquisition Parameters:  'input1' is the current selected source.
                            10 frames per trigger using the selected source.
                            'Y8_1024x768' video data to be logged upon START.
                            Grabbing first of every 1 frame(s).
                            Log data to 'memory' on trigger.

Trigger Parameters:  1 'immediate' trigger(s) on START.
            Status:  Waiting for START.
                     0 frames acquired since starting.
                     0 frames available for GETDATA.
```

Step 4: Preview the Video Stream (Optional)

After you create the video input object, MATLAB is able to access the image acquisition device and is ready to acquire data. However, before you begin, you might want to see a preview of the video stream to make sure that the image is satisfactory. For example, you might want to change the position of the camera, change the lighting, correct the focus, or make some other change to your image acquisition setup.

Note This step is optional at this point in the procedure because you can preview a video stream at any time after you create a video input object.

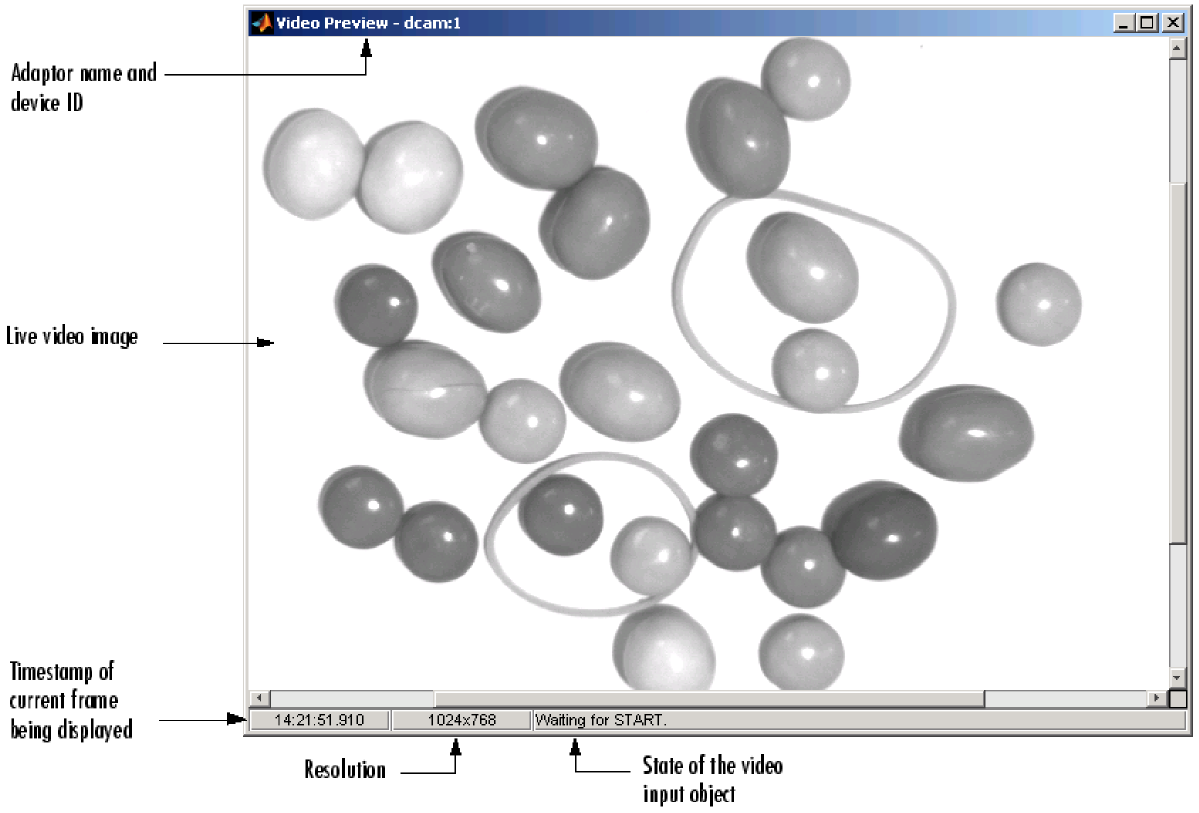
To preview the video stream in this example, enter the `preview` function at the MATLAB prompt, specifying the video input object created in step 3 as an argument.

```
preview(vid)
```

The `preview` function opens a Video Preview figure window on your screen containing the live video stream. To stop the stream of live video, you can call the `stoppreview` function. To restart the preview stream, call `preview` again on the same video input object.

While a preview window is open, the video input object sets the value of the `Previewing` property to `'on'`. If you change characteristics of the image by setting image acquisition object properties, the image displayed in the preview window reflects the change.

The following figure shows the Video Preview window for the example.


Adaptor name and device ID
Video Preview - dcam:1
Live video image
Timestamp of current frame being displayed
14:21:51.910
1024x768
Waiting for START.
Resolution
State of the video input object

Video Preview Window

To close the Video Preview window, click the **Close** button in the title bar or use the `closepreview` function, specifying the video input object as an argument.

```
closepreview(vid)
```

Calling `closepreview` without any arguments closes all open Video Preview windows.

Step 5: Configure Object Properties (Optional)

After creating the video input object and previewing the video stream, you might want to modify characteristics of the image or other aspects of the acquisition process. You accomplish this by setting the values of image acquisition object properties. This section

- Describes the types of image acquisition objects on page 1-11 used by the toolbox
- Describes how to view all the properties on page 1-12 supported by these objects, with their current values
- Describes how to set the values on page 1-13 of object properties

Types of Image Acquisition Objects

The toolbox uses two types of objects to represent the connection with an image acquisition device:

- Video input objects
- Video source objects

A video input object represents the connection between MATLAB and a video acquisition device at a high level. The properties supported by the video input object are the same for every type of device. You created a video input object using the `videoinput` function in step 3 on page 1-9.

When you create a video input object, the toolbox automatically creates one or more video source objects associated with the video input object. Each video source object represents a collection of one or more physical data sources that are treated as a single entity. The number of video source objects the toolbox creates depends on the device and the video format you specify. At any one time, only one of the video source objects, called the *selected* source, can be active. This is the source used for acquisition. For more information about these image acquisition objects, see “Creating Image Acquisition Objects” on page 5-6.

Viewing Object Properties

To view a complete list of all the properties supported by a video input object or a video source object, use the `get` function. To list the properties of the video input object created in step 3, enter this code at the MATLAB prompt.

```
get(vid)
```

The `get` function lists all the properties of the object with their current values.

```
General Settings:
    DeviceID = 1
    DiskLogger = []
    DiskLoggerFrameCount = 0
    EventLog = [1x0 struct]
    FrameGrabInterval = 1
    FramesAcquired = 0
    FramesAvailable = 0
    FramesPerTrigger = 10
    Logging = off
    LoggingMode = memory
    Name = Y8_1024x768-dcam-1
    NumberOfBands = 1
    Previewing = on
    ReturnedColorSpace = grayscale
    ROIPosition = [0 0 1024 768]
    Running = off
    Tag = 
    Timeout = 10
    Type = videoinput
    UserData = []
    VideoFormat = Y8_1024x768
    VideoResolution = [1024 768]
    .
    .
    .
```

To view the properties of the currently selected video source object associated with this video input object, use the `getselectedsource` function in conjunction with the `get` function. The `getselectedsource` function returns the currently active video source. To list the properties of the currently selected video source object associated with the video input object created in step 3, enter this code at the MATLAB prompt.

```
get(getselectedsource(vid))
```

The `get` function lists all the properties of the object with their current values.

Note Video source object properties are device specific. The list of properties supported by the device connected to your system might differ from the list shown in this example.

```
General Settings:
    Parent = [1x1 videoinput]
    Selected = on
    SourceName = input1
    Tag = 
    Type = videosource

  Device Specific Properties:
    FrameRate = 15
    Gain = 2048
    Shutter = 2715
```

Setting Object Properties

To set the value of a video input object property or a video source object property, you reference the object property as you would a field in a structure, using dot notation.

Some properties are read only; you cannot set their values. These properties typically provide information about the state of the object. Other properties become read only when the object is running. To view a list of all the properties you can set, use the `set` function, specifying the object as the only argument.

To implement continuous image acquisition, the example sets the `TriggerRepeat` property to `Inf`. To set this property, enter this code at the MATLAB prompt.

```
vid.TriggerRepeat = Inf;
```

To help the application keep up with the incoming video stream while processing data, the example sets the `FrameGrabInterval` property to `5`. This specifies that the object acquire every fifth frame in the video stream. (You might need to experiment with the value of the `FrameGrabInterval` property to find a value that provides the best response with your image acquisition setup.) This example shows how you can set the value of an object property by referencing the property as you would reference a field in a MATLAB structure.

```
vid.FrameGrabInterval = 5;
```

To set the value of a video source object property, you must first use the `getselectedsource` function to retrieve the object. (You can also get the selected source by searching the video input object `Source` property for the video source object that has the `Selected` property set to `'on'`.)

To illustrate, the example assigns a value to the `Tag` property.

```
vid_src = getselectedsource(vid);

vid_src.Tag = 'motion detection setup';
```

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For

information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Step 6: Acquire Image Data

After you create the video input object and configure its properties, you can acquire data. This is typically the core of any image acquisition application, and it involves these steps:

- **Starting the video input object** — You start an object by calling the `start` function. Starting an object prepares the object for data acquisition. For example, starting an object locks the values of certain object properties (they become read only). Starting an object does not initiate the acquiring of image frames, however. The initiation of data logging depends on the execution of a trigger.

  The following example calls the `start` function to start the video input object. Objects stop when they have acquired the requested number of frames. Because the example specifies a continuous acquisition, you must call the `stop` function to stop the object.
- **Triggering the acquisition** — To acquire data, a video input object must execute a trigger. Triggers can occur in several ways, depending on how the `TriggerType` property is configured. For example, if you specify an immediate trigger, the object executes a trigger automatically, immediately after it starts. If you specify a manual trigger, the object waits for a call to the `trigger` function before it initiates data acquisition. For more information, see “Acquiring Image Data” on page 6-2.

  In the example, because the `TriggerType` property is set to `'immediate'` (the default) and the `TriggerRepeat` property is set to `Inf`, the object automatically begins executing triggers and acquiring frames of data, continuously.
- **Bringing data into the MATLAB workspace** — The toolbox stores acquired data in a memory buffer, a disk file, or both, depending on the value of the video input object `LoggingMode` property. To work with this data, you must bring it into the MATLAB workspace. To bring multiple frames into the workspace, use the `getdata` function. Once the data is in the MATLAB workspace, you can manipulate it as you would any other data. For more information, see “Working with Image Data in MATLAB Workspace” on page 7-9.

Note The toolbox provides a convenient way to acquire a single frame of image data that doesn't require starting or triggering the object. See “Bringing a Single Frame into the Workspace” on page 7-7 for more information.

Running the Example

To run the example, enter the following code at the MATLAB prompt. The example loops until a specified number of frames have been acquired. In each loop iteration, the example calls `getdata` to bring the two most recent frames into the MATLAB workspace. To detect motion, the example subtracts one frame from the other, creating a difference image, and then displays it. Pixels that have changed values in the acquired frames will have nonzero values in the difference image.

The `getdata` function removes frames from the memory buffer when it brings them into the MATLAB workspace. It is important to move frames from the memory buffer into the MATLAB workspace in a timely manner. If you do not move the acquired frames from memory, you can quickly exhaust all the memory available on your system.

Note The example uses functions in the Image Processing Toolbox software.

```
% Create video input object.
vid = videoinput('dcam',1,'Y8_1024x768')

% Set video input object properties for this application.
vid.TriggerRepeat = 100;
vid.FrameGrabInterval = 5;

% Set value of a video source object property.
vid_src = getselectedsource(vid);
vid_src.Tag = 'motion detection setup';

% Create a figure window.
figure;

% Start acquiring frames.
start(vid)

% Calculate difference image and display it.
while(vid.FramesAvailable >= 2)
    data = getdata(vid,2);
    diff_im = imabsdiff(data(:,:,:,1),data(:,:,:,2));
    imshow(diff_im);
    drawnow     % update figure window
end

stop(vid)
```

Note that a `drawnow` is used after the call to `imshow` in order to ensure that the figure window is updated. This is good practice when updating a GUI or figure inside a loop.

The following figure shows how the example displays detected motion. In the figure, areas representing movement are displayed.

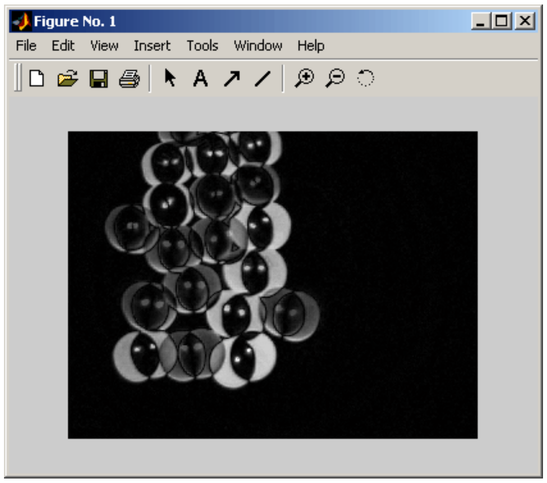


Figure Window Displayed by Example

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Image Data in the MATLAB Workspace

In the example, the `getdata` function returns the image frames in the variable `data` as a 480-by-640-by-1-by-10 array of 8-bit data (`uint8`).

```
whos
  Name           Size                   Bytes  Class

  data           4-D                  3072000  uint8 array
  dev_info       1x1                     1601  struct array
  info           1x1                     2467  struct array
  vid            1x1                     1138  videoinput object
  vid_src        1x1                      726  videosource object
```

The height and width of the array are primarily determined by the video resolution of the video format. However, you can use the `ROIPosition` property to specify values that supersede the video resolution. Devices typically express video resolution as column-by-row; MATLAB expresses matrix dimensions as row-by-column.

The third dimension represents the number of color bands in the image. Because the example data is a grayscale image, the third dimension is 1. For RGB formats, image frames have three bands: red is the first, green is the second, and blue is the third. The fourth dimension represents the number of frames that have been acquired from the video stream.

Step 7: Clean Up

When you finish using your image acquisition objects, you can remove them from memory and clear the MATLAB workspace of the variables associated with these objects.

```
delete(vid)
clear
close(gcf)
```

For more information, see “Deleting Image Acquisition Objects” on page 5-28.

Introduction

This chapter describes the Image Acquisition Toolbox software and its components.

Toolbox Components Overview

In this section...

Introduction

Image Acquisition Toolbox enables you to acquire images and video from cameras and frame grabbers directly into MATLAB and Simulink. You can detect hardware automatically, and configure hardware properties. Advanced workflows let you trigger acquisitions while processing in-the-loop, perform background acquisitions, and synchronize sampling across several multimodal devices. With support for multiple hardware vendors and industry standards, you can use imaging devices, ranging from inexpensive Web cameras to high-end scientific and industrial devices that meet low-light, high-speed, and other challenging requirements.

The Image Acquisition Toolbox software implements an object-oriented approach to image acquisition. Using toolbox functions, you create an object that represents the connection between MATLAB and specific image acquisition devices. Using properties of the object you can control various aspects of the acquisition process, such as the amount of video data you want to capture. "Creating Image Acquisition Objects" on page 5-6 describes how to create objects.

Once you establish a connection to a device, you can acquire image data by executing a trigger. In the toolbox, all image acquisition is initiated by a trigger. The toolbox supports several types of triggers that let you control when an acquisition takes place. For example, using hardware triggers you can synchronize an acquisition with an external device. "Acquiring Image Data" on page 6-2 describes how to trigger the acquisition of image data.

To work with the data you acquire, you must bring it into the MATLAB workspace. When the frames are acquired, the toolbox stores them in a memory buffer. The toolbox provides several ways to bring one or more frames of data into the workspace where you can manipulate it as you would any other multidimensional numeric array. "Bringing Image Data into the MATLAB Workspace" on page 7-3 describes this process.

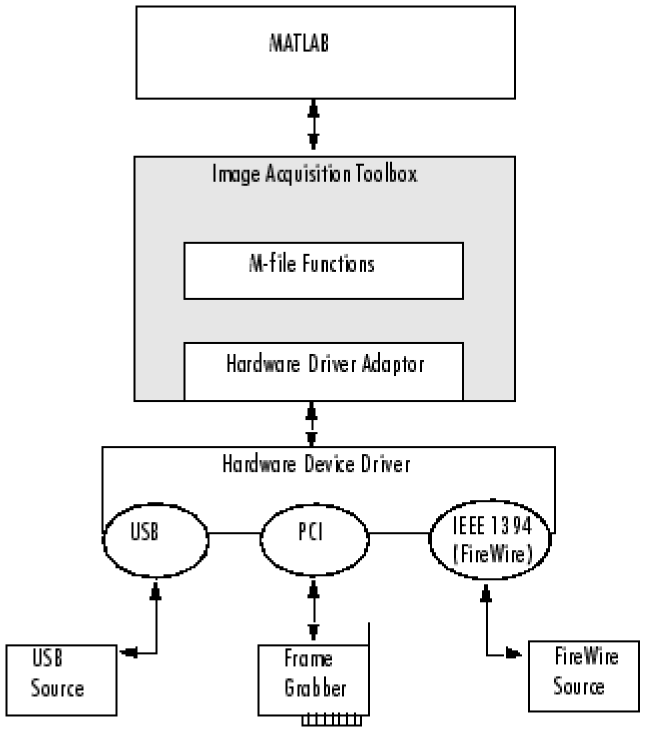
Finally, you can enhance your image acquisition application by using event callbacks. The toolbox has defined certain occurrences, such as the triggering of an acquisition, as events. You can associate the execution of a particular function with a particular event. "Using Events and Callbacks" on page 8-2 describes this process.

Note With previous versions of the Image Acquisition Toolbox, the files for all of the adaptors were included in your installation. Starting with version R2014a, each adaptor is available separately through support packages. In order to use the Image Acquisition Toolbox, you must install the adaptor that your camera uses. See "Image Acquisition Support Packages for Hardware Adaptors" on page 4-2 for information about installing the adaptors using MATLAB Add-Ons.

Toolbox Components

The toolbox uses components called hardware device adaptors to connect to devices through their drivers. The toolbox includes adaptors that support devices produced by several vendors of image acquisition equipment. In addition, the toolbox includes an adaptor for generic Windows video acquisition devices.

The following figure shows these components and their relationship.



The Image Acquisition Toolbox Software Components

The Image Acquisition Explorer App

The functionality of the Image Acquisition Toolbox software is available in a desktop application. You connect directly to your hardware in the app and can then set acquisition parameters, and preview and acquire image data. You can log the data to a file or to the MATLAB workspace, and also generate a MATLAB script with the device configuration, right from the app.

To open the app, type `imageAcquisitionExplorer` at the MATLAB command line, or select **Image Acquisition Explorer** on the **Apps** tab in MATLAB. For more information, see "Get Started with Image Acquisition Explorer" on page 3-5.

Supported Devices

The Image Acquisition Toolbox software includes adaptors that provide support for several vendors of professional grade image acquisition equipment, devices that support the IIDC 1394-based Digital Camera Specification (DCAM), and devices that provide Windows Driver Model (WDM) or Video for Windows (VFW) drivers, such as USB and IEEE® 1394 (FireWire, i.LINK®) Web cameras, Digital video

(DV) camcorders, and TV tuner cards. For the latest information about supported hardware, visit the Image Acquisition Toolbox product page at the MathWorks Web site (`www.mathworks.com/products/image-acquisition`).

The DCAM specification, developed by the 1394 Trade Association, describes a generic interface for exchanging data with IEEE 1394 (FireWire) digital cameras that is often used in scientific applications. The toolbox's DCAM adaptor supports Format 7, also known as partial scan mode. The toolbox uses the prefix `F7_` to identify Format 7 video format names.

Note The toolbox supports only connections to IEEE 1394 (FireWire) DCAM-compliant devices using the Carnegie Mellon University DCAM driver. The toolbox is not compatible with any other vendor-supplied driver, even if the driver is DCAM compliant.

You can add support for additional hardware by writing an adaptor. For more information, see “Support for Additional Hardware” on page 15-2.

Note With previous versions of the Image Acquisition Toolbox, the files for all of the adaptors were included in your installation. Starting with version R2014a, each adaptor is available separately through support packages. In order to use the Image Acquisition Toolbox, you must install the adaptor that your camera uses. See “Image Acquisition Support Packages for Hardware Adaptors” on page 4-2 for information about installing the adaptors using MATLAB Add-Ons.

Setting Up Image Acquisition Hardware

In this section...

Introduction

To acquire image data, you must perform the setup required by your particular image acquisition device. In a typical image acquisition setup, an image acquisition device, such as a camera, is connected to a computer via an image acquisition board, such as a frame grabber, or via a Universal Serial Bus (USB) or IEEE 1394 (FireWire) port. The setup required varies with the type of device.

After installing and configuring your image acquisition hardware, start MATLAB on your computer by double-clicking the icon on your desktop. You do not need to perform any special configuration of MATLAB to acquire data.

Note With previous versions of the Image Acquisition Toolbox, the files for all of the adaptors were included in your installation. Starting with version R2014a, each adaptor is available separately through support packages. In order to use the Image Acquisition Toolbox, you must install the adaptor that your camera uses. See “Image Acquisition Support Packages for Hardware Adaptors” on page 4-2 for information about installing the adaptors using MATLAB Add-Ons.

Setting Up Frame Grabbers

For frame grabbers, also known as imaging boards, setup typically involves the following tasks:

- Installing the frame grabber in your computer
- Installing any software drivers required by the frame grabber. These are supplied by the device vendor.
- Connecting the camera, or other image acquisition device, to a connector on the frame grabber
- Verifying that the camera is working properly by running the application software that came with the frame grabber and viewing a live video stream

Setting Up Generic Windows Video Acquisition Devices

IEEE 1394 (FireWire) and generic Windows video acquisition devices that use Windows Driver Model (WDM) or Video for Windows (VFW) device drivers typically require less setup. Plug the device into the USB or IEEE 1394 (FireWire) port on your computer and install the device driver provided by the vendor.

Setting Up DCAM Devices

If you intend to access a DCAM-compliant IEEE 1394 (FireWire) camera, you must install and configure the Carnegie Mellon University (CMU) DCAM driver. The toolbox is not compatible with any other vendor-supplied driver, even if the driver is DCAM compliant. See “Manually Installing the CMU DCAM Driver on Windows” on page 16-6 for more information.

Resetting Your Image Acquisition Hardware

To return MATLAB and your image acquisition hardware to a known state, where no image acquisition objects exist and the hardware is not configured, use the `imaqreset` function.

If you connect another image acquisition device to your system after MATLAB is started, you can use `imaqreset` to make the toolbox aware of the new hardware.

A Note About Frame Rates and Processing Speed

The frame rate describes how fast an image acquisition device provides data, typically measured as frames per second.

Devices that support industry-standard video formats must provide frames at the rate specified by the standard. For RS170 and NTSC, the standard dictates a frame rate of 30 frames per second (30 Hz). The CCIR and PAL standards define a frame rate of 25 Hz. Nonstandard devices can be configured to operate at higher rates. Generic Windows image acquisition devices, such as webcams, might support many different frame rates. Depending on the device being used, the frame rate might be configurable using a device-specific property of the image acquisition object.

The rate at which the Image Acquisition Toolbox software can process images depends on the processor speed, the complexity of the processing algorithm, and the frame rate. Given a fast processor, a simple algorithm, and a frame rate tuned to the acquisition setup, the Image Acquisition Toolbox software can process data as it comes in.

Preview Live Data from Image Acquisition Device

In this section...

Introduction

After you connect MATLAB to the image acquisition device you can view the live video stream using the Video Preview window. Previewing the video data can help you make sure that the image being captured is satisfactory.

For example, by looking at a preview, you can verify that the lighting and focus are correct. If you change characteristics of the image, by using video input object and video source object properties, the image displayed in the Video Preview window changes to reflect the new property settings.

The following sections provide more information about using the Video Preview window.

- “Opening a Video Preview Window” on page 2-7
- “Stopping the Preview Video Stream” on page 2-8
- “Closing a Video Preview Window” on page 2-9

Instead of using the toolbox's Video Preview window, you can display the live video preview stream in any Handle Graphics® image object you specify. In this way, you can include video previewing in a GUI of your own creation. The following sections describe this capability.

- “Previewing Data in Custom GUIs” on page 2-9
- “Performing Custom Processing of Previewed Data” on page 2-11

Opening a Video Preview Window

To open a Video Preview window, use the `preview` function. The Video Preview window displays the live video stream from the device. You can only open one preview window per device. If multiple devices are used, you can open multiple preview windows at the same time.

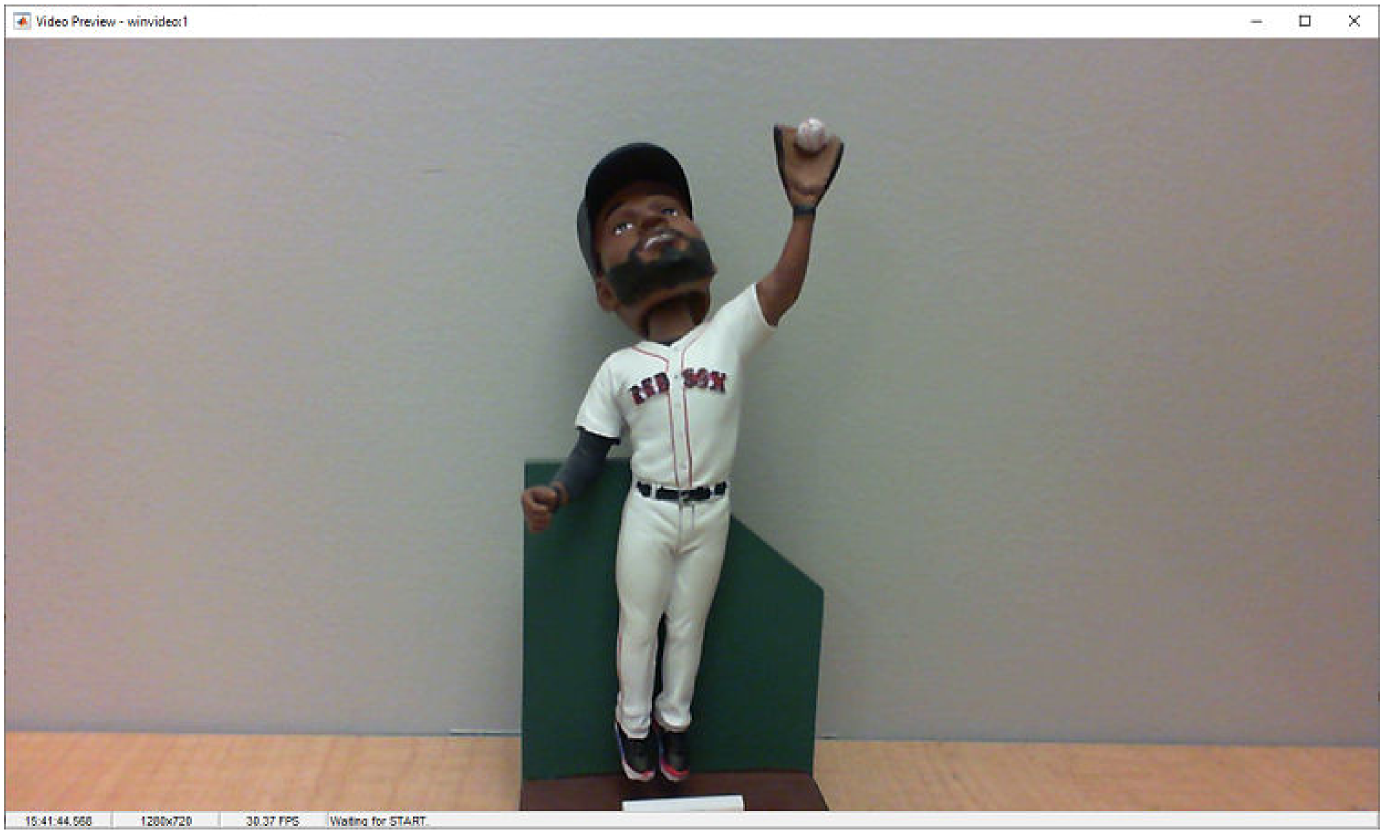
The following example creates a video input object and then opens a Video Preview window for the video input object.

```
vid = videoinput('winvideo');
preview(vid);
```

The following figure shows the Video Preview window created by this example. The Video Preview window displays the live video stream. The size of the preview image is determined by the value of the video input object's `ROIPosition` property. The Video Preview window displays the video data at 100% magnification.

In addition to the preview image, the Video Preview window includes information about the image, such as the timestamp of the video frame, the video resolution, the frame rate, and the current status of the video input object.

Note Because video formats typically express resolution as width-by-height, the Video Preview window expresses the size of the image frame as column-by-row, rather than the standard MATLAB row-by-column format.



Note The Image Acquisition Toolbox Preview window supports the display of up to 16-bit image data. The Preview window was designed to only show 8-bit data, but many cameras return 10-, 12-, 14-, or 16-bit data. The Preview window display supports these higher bit-depth cameras. However, larger bit data is scaled to 8-bit for the purpose of displaying previewed data. To capture the image data in the Preview window in its full bit depth for grayscale images, set the `PreviewFullBitDepth` property to `'on'`.

Stopping the Preview Video Stream

When you use the `preview` function to start previewing image data, the Video Preview window displays a view of the live video stream coming from the device. To stop the updating of the live video stream, call the `stoppreview` function.

This example creates a video input object and opens a Video Preview window. The example then calls the `stoppreview` function on this video input object. The Video Preview window stops updating the image displayed and stops updating the timestamp. The status displayed in the Video Preview window also changes to indicate that previewing has been stopped.

```
vid = videoinput('winvideo');
preview(vid)
stoppreview(vid)
```

To restart the video stream in the Video Preview window, call `preview` again on the same video input object.

```
preview(vid)
```

Closing a Video Preview Window

To close a particular Video Preview window, use the `closepreview` function, specifying the video input object as an argument. You do not need to stop the live video stream displayed in the Video Preview window before closing it.

```
closepreview(vid)
```

To close all currently open Video Preview windows, use the `closepreview` function without any arguments.

```
closepreview
```

Note When called without an argument, the `closepreview` function only closes Video Preview windows. The `closepreview` function does not close any other figure windows in which you have directed the live preview video stream. For more information, see “Previewing Data in Custom GUIs” on page 2-9.

Previewing Data in Custom GUIs

Instead of using the toolbox's Video Preview window, you can use the `preview` function to direct the live video stream to any Handle Graphics image object. In this way, you can incorporate the toolbox's previewing capability in a GUI of your own creation. (You can also perform custom processing as the live video is displayed. For information, see “Performing Custom Processing of Previewed Data” on page 2-11.)

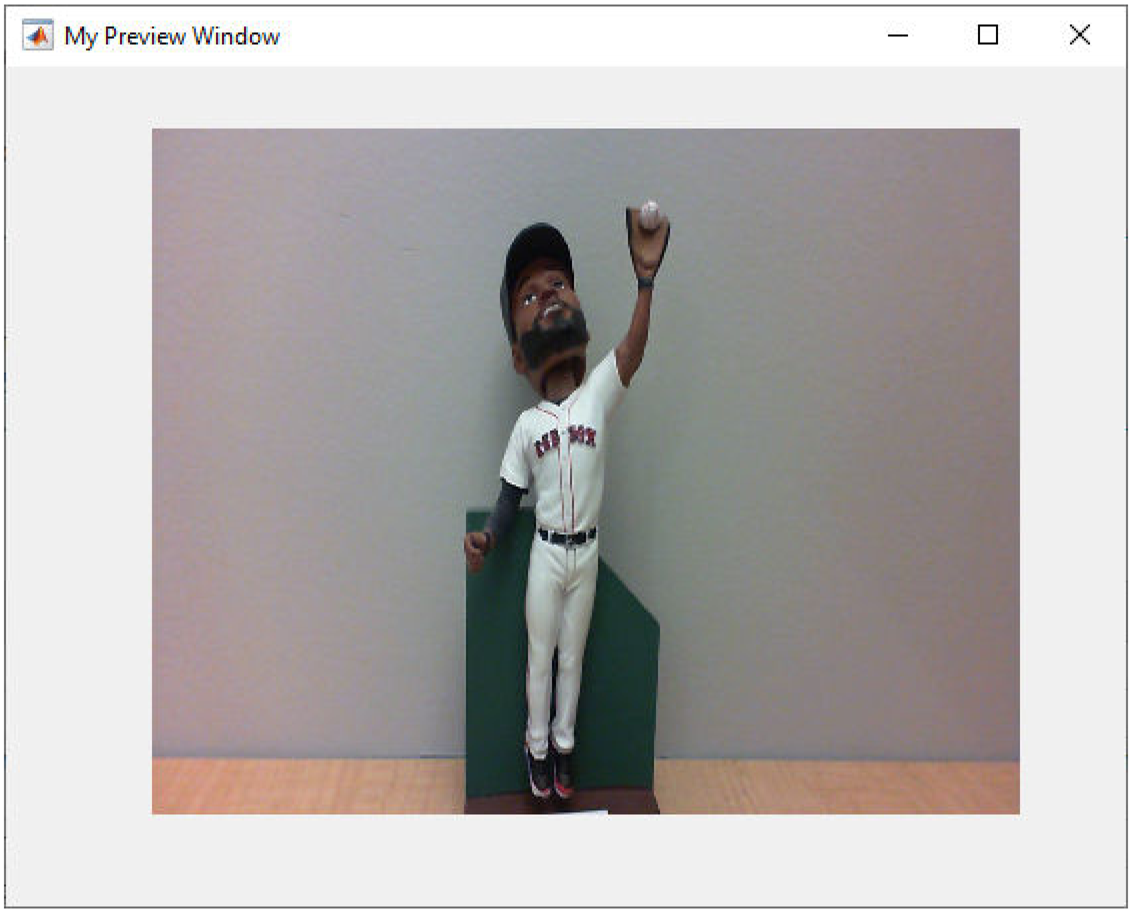
To use this capability, create an image object and then call the `preview` function, specifying a handle to the image object as an argument. The `preview` function outputs the live video stream to the image object you specify.

The following example creates a figure window and then creates an image object in the figure, the same size as the video frames. The example then calls the `preview` function, specifying a handle to the image object.

```
% Create a video input object.
vid = videoinput('winvideo');

% Create a figure window. This example turns off the default
% toolbar, menubar, and figure numbering.
```

```
figure('Toolbar','none',...
       'Menubar', 'none',...
       'NumberTitle','Off',...
       'Name','My Preview Window');

% Create the image object in which you want to display
% the video preview data. Make the size of the image
% object match the dimensions of the video frames.

vidRes = vid.VideoResolution;
nBands = vid.NumberOfBands;
hImage = image( zeros(vidRes(2), vidRes(1), nBands) );

% Display the video data in your GUI.

preview(vid, hImage);
```

When you run this example, it creates the GUI shown in the following figure.



Custom Preview

Performing Custom Processing of Previewed Data

When you specify an image object to the `preview` function (see “Previewing Data in Custom GUIs” on page 2-9), you can optionally also specify a function that `preview` executes every time it receives an image frame.

To use this capability, follow these steps:

1. Create the function you want executed for each image frame, called the update preview window function. For information about this function, see “Creating the Update Preview Window Function” on page 2-11.
2. Create an image object.
3. Configure the value of the image object's `'UpdatePreviewWindowFcn'` application-defined data to be a function handle to your update preview window function. For more information, see “Specifying the Update Preview Function” on page 2-12.
4. Call the `preview` function, specifying the handle of the image object as an argument.

Note If you specify an update preview window function, in addition to whatever processing your function performs, it must display the video data in the image object. You can do this by updating the `CData` of the image object with the incoming video frames. For some performance guidelines about updating the data displayed in an image object, see Technical Solution 1-1B022.

Creating the Update Preview Window Function

When `preview` calls the update preview window function you specify, it passes your function the following arguments.

Argument	Description	
`obj`	Handle to the video input object being previewed	
`event`	A data structure containing the following fields:	
	`Data`	Current image frame specified as an H-by-W-by-B array, where H is the image height and W is the image width, as specified in the `ROIPosition` property, and B is the number of color bands, as specified in the `NumberOfBands` property
	`Resolution`	Character vector specifying the current image width and height, as defined by the `ROIPosition` property
	`Status`	Character vector describing the status of the video input object
	`Timestamp`	Character vector specifying the time associated with the current image frame, in the format `hh:mm:ss:ms`
	`FrameRate`	Character vector specifying the current frame rate of the video input object in frames per second
`himage`	Handle to the image object in which the data is to be displayed	

The following example creates an update preview window function that displays the timestamp of each incoming video frame as a text label in the custom GUI. The update preview window function uses getappdata to retrieve a handle to the text label uicontrol object from application-defined data in the image object. The custom GUI stores this handle to the text label uicontrol object — see “Specifying the Update Preview Function” on page 2-12.

Note that the update preview window function also displays the video data by updating the CData of the image object.

```
function mypreview_fcn(obj,event,himage)
% Example update preview window function.

% Get timestamp for frame.
tstampstr = event.Timestamp;

% Get handle to text label uicontrol.
ht = getappdata(himage,'HandleToTimestampLabel');

% Set the value of the text label.
ht.String = tstampstr;

% Display image data.
himage.CData = event.Data
```

Specifying the Update Preview Function

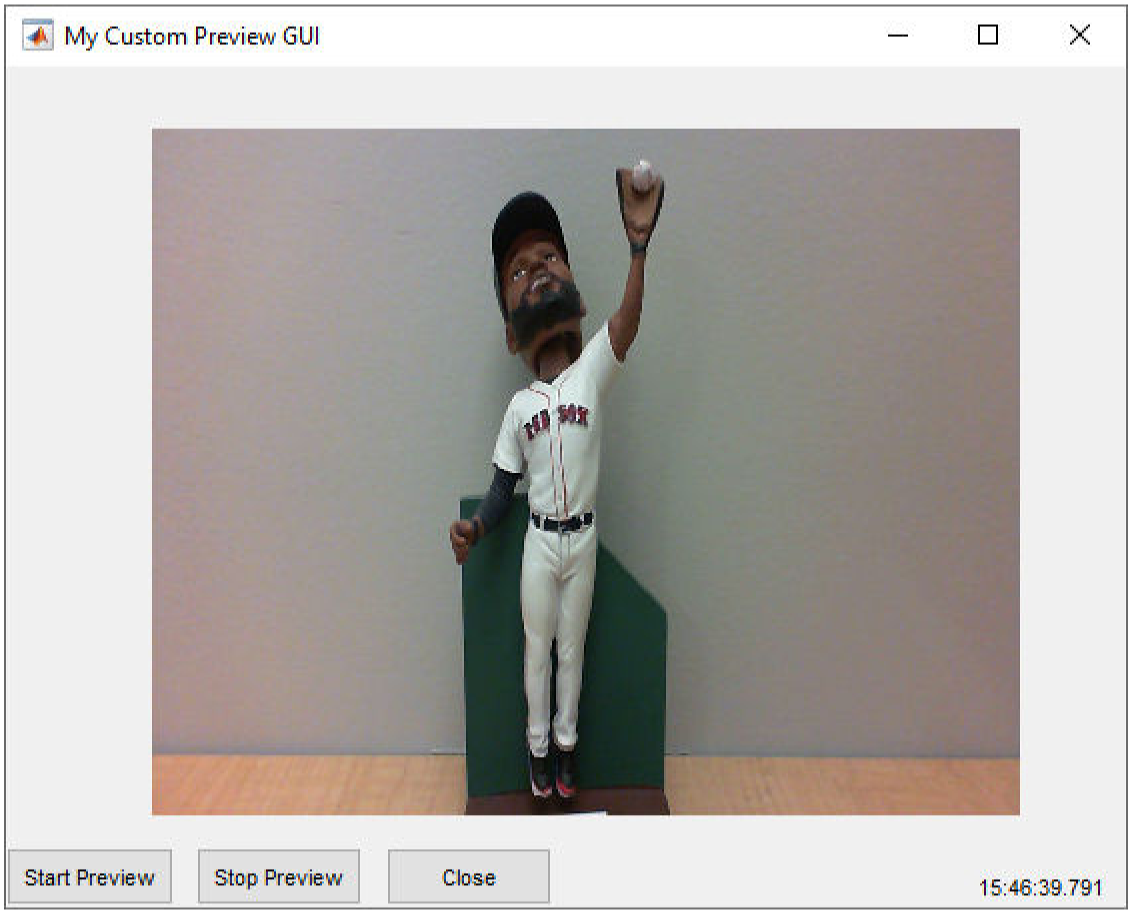
To use an update preview window function, store a function handle to your function in the 'UpdatePreviewWindowFcn' application-defined data of the image object. The following example uses the setappdata function to configure this application-defined data to a function handle to the update preview window function described in “Creating the Update Preview Window Function” on page 2-11.

This example extends the simple custom preview window created in “Previewing Data in Custom GUIs” on page 2-9. This example adds three push button uicontrol objects to the GUI: **Start Preview**, **Stop Preview**, and **Close Preview**.

In addition, to illustrate using an update preview window function, the example GUI includes a text label uicontrol object to display the timestamp value. The update preview window function updates this text label each time a framed is received. The example uses setappdata to store a handle to the text label uicontrol object in application-defined data in the image object. The update preview window function retrieves this handle to update the timestamp display.

```
% Create a video input object.
vid = videoinput('winvideo');

% Create a figure window. This example turns off the default
% toolbar and menubar in the figure.
hFig = figure('Toolbar','none',...
       'Menubar', 'none',...
       'NumberTitle','Off',...
       'Name','My Custom Preview GUI');

% Set up the push buttons
uicontrol('String', 'Start Preview',...
    'Callback', 'preview(vid)',...
    'Units','normalized',...
    'Position',[0 0 0.15 .07]);
uicontrol('String', 'Stop Preview',...
    'Callback', 'stoppreview(vid)',...
    'Units','normalized',...
    'Position',[.17 0 .15 .07]);
uicontrol('String', 'Close',...
    'Callback', 'close(gcf)',...
    'Units','normalized',...
    'Position',[0.34 0 .15 .07]);

% Create the text label for the timestamp
hTextLabel = uicontrol('style','text','String','Timestamp', ...
    'Units','normalized',...
    'Position',[0.85 -.04 .15 .08]);

% Create the image object in which you want to
% display the video preview data.
vidRes = vid.VideoResolution;
imWidth = vidRes(1);
imHeight = vidRes(2);
nBands = vid.NumberOfBands;
hImage = image( zeros(imHeight, imWidth, nBands) );

% Specify the size of the axes that contains the image object
% so that it displays the image at the right resolution and
% centers it in the figure window.
figSize = get(hFig,'Position');
figWidth = figSize(3);
figHeight = figSize(4);
gca.unit = 'pixels';
gca.position = [ ((figWidth - imWidth)/2)...
               ((figHeight - imHeight)/2)...
               imWidth imHeight ];

% Set up the update preview window function.
setappdata(hImage,'UpdatePreviewWindowFcn',@mypreview_fcn);

% Make handle to text label available to update function.
setappdata(hImage,'HandleToTimestampLabel',hTextLabel);

preview(vid, hImage);
```

When you run this example, it creates the GUI shown in the following figure. Each time `preview` receives a video frame, it calls the update preview window function that you specified, which updates the timestamp text label in the GUI.



Custom Preview GUI with Timestamp Text Label

Using the Image Acquisition Explorer

Image Acquisition Explorer Overview

Open the App

You can use the Image Acquisition Toolbox functionalities in the **Image Acquisition Explorer** app. Connect directly to your hardware from the app to preview and acquire image and video data. You can log the data to MATLAB as a workspace variable or to your computer as an image or video file.

The **Image Acquisition Explorer** provides an interface that integrates a preview area with acquisition parameters, so that you can change settings and see the changes dynamically applied to your image data.

There are two ways to launch the app:

- MATLAB Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click the app icon.
- MATLAB command prompt: Enter `imageAcquisitionExplorer`.

Parts of the App

The **Image Acquisition Explorer** consists of the following components.

- **Preview** — See a live preview from your selected image acquisition device. You can toggle the preview on and off by clicking the switch above the preview. The preview updates in real-time as you change device properties. The area below the preview displays the current frame rate and timestamp. The status bar below the preview displays status messages when you capture an image snapshot or record a video. If you select `grayscale` as the **Color Space**, you can set the colormap and color limits above the preview.

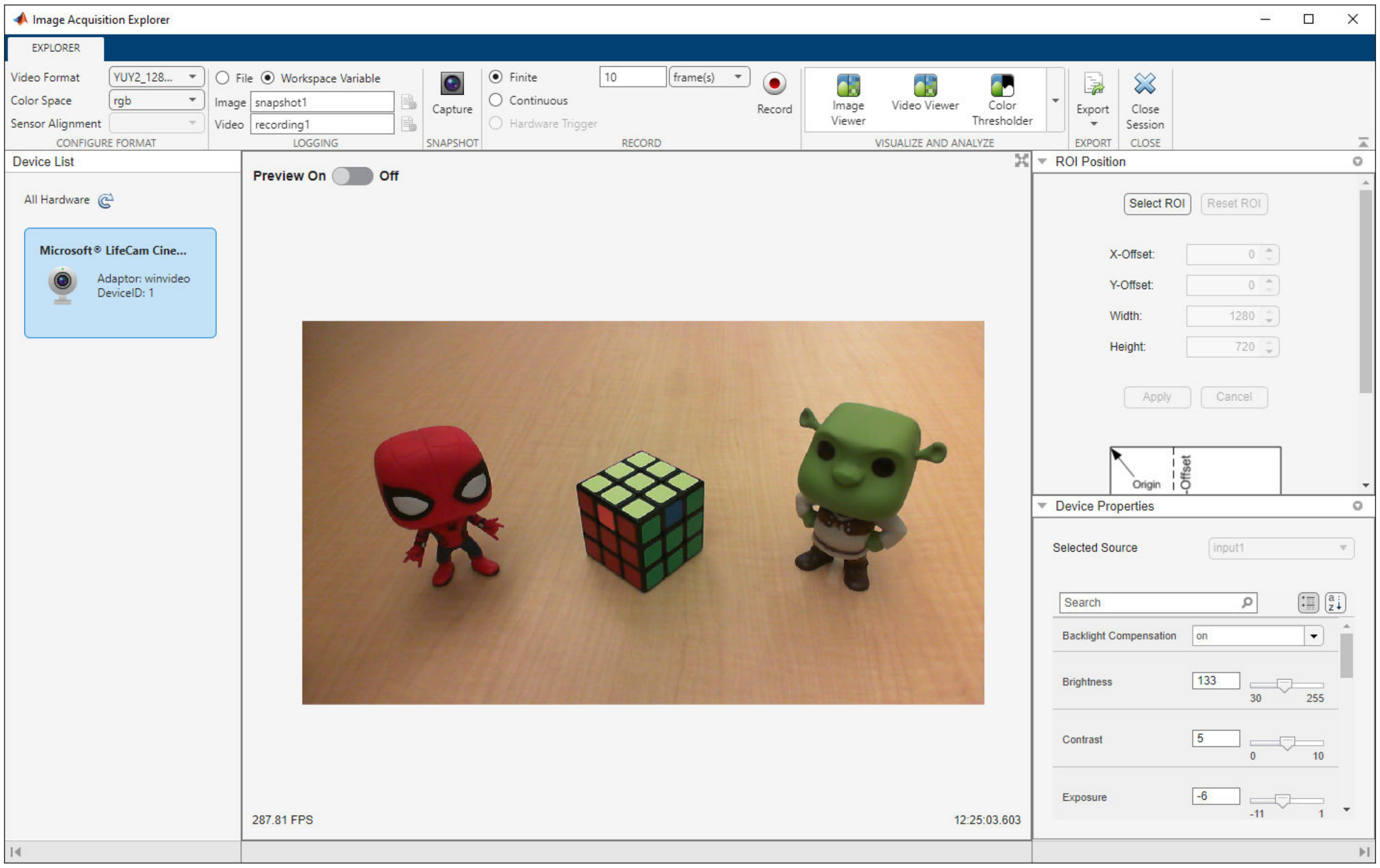
Panels

- **Device List** — View a list of the image acquisition devices on your computer and available to connect to from the app. If you don't see your device, make sure you have the appropriate support package installed, make sure your device is plugged in, and click the refresh button. You can click on a different device to close your current device connection and switch to another device in the app.
- **ROI Position** — Set your desired region of interest (ROI) for image acquisition. You can interactively select an ROI in the preview by clicking **Select ROI**. You can also manually specify ROI values for X-Offset, Y-Offset, Width, and Height in pixels. Click **Apply** after you set your desired ROI.
- **Device Properties** — View and edit device properties. The specific list of properties that appear depends on your selected device. Changing properties here updates the preview in real-time.
- **Hardware Trigger** — Set up hardware triggering options. This is only available if your device supports hardware triggering and you selected **Hardware Trigger** in the toolstrip. Specify the number of triggers and frames per trigger, as well as the trigger source and condition.

Toolstrip Sections

- **Configure Format** — Specify the format of the images to acquire from your selected device. The options available for **Video Format** and **Color Space** depend on your device. These parameters define different resolutions and color spaces that your device supports, or different video standards or camera configurations for your device. If your device supports Bayer sensor alignment, you can set the **Sensor Alignment**. If your device supports camera files, the **Select Camera File** button appears here.

- **Logging** — Choose whether you want to save your image data as a file or as a workspace variable. Your selection here determines how image data is logged when you click **Capture** or **Record**. After selecting the **File** or **Workspace Variable** option, you can edit the names and file locations for data logging. If you select **File**, you can specify additional file configuration settings by clicking the icon next to the file name.
- **Snapshot** — Click the **Capture** button to immediately acquire and save an image snapshot using the current device properties and configuration.
- **Record** — Select finite, continuous, or hardware triggered recording and click the **Record** button to start recording a video using the current device properties and configuration.
- **Visualize and Analyze** — Open one of the Image Processing Toolbox apps and send it the most recently saved image or video data.
- **Export** — Click the **Export** button and select one of the options to generate a MATLAB live script for capturing a snapshot or recording a video and open it in the Live Editor. The live script contains code for the current device configuration and code for saving data as a file or workspace variable.



Video Format: YUY2_128...
Color Space: rgb
Sensor Alignment:
File [✓ Workspace Variable]
Image: snapshot1
Video: recording1
[✓ Finite] Continuous Hardware Trigger
10 frame(s)

Microsoft® LifeCam Cine...
Adaptor: winvideo
DeviceID: 1

287.81 FPS
12:25:03.603

X-Offset: 0
Y-Offset: 0
Width: 1280
Height: 720

Selected Source: input1
Backlight Compensation: on
Brightness: 133 (30–255)
Contrast: 5 (0–10)
Exposure: -6 (-11–1)

See Also

Image Acquisition Explorer

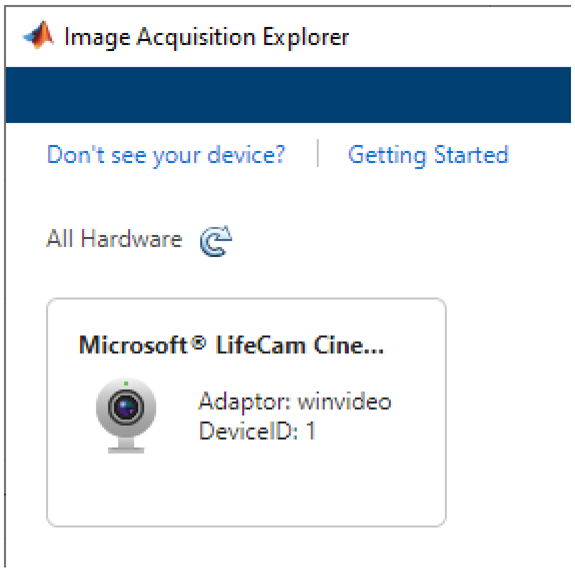
Related Examples

- “Get Started with Image Acquisition Explorer” on page 3-5
- “Select Your Device and Configure Format in Image Acquisition Explorer” on page 3-10
- “Set Acquisition Parameters in Image Acquisition Explorer” on page 3-13
- “Log Data in Image Acquisition Explorer” on page 3-18
- “Preview and Acquire Data in Image Acquisition Explorer” on page 3-22
- “Export Code from Image Acquisition Explorer” on page 3-27

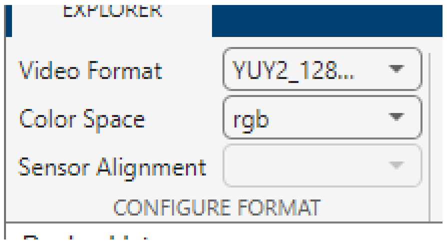
Get Started with Image Acquisition Explorer

The basic workflow of using the **Image Acquisition Explorer** is to preview, configure, acquire, and save image data. For information on opening the app or the parts of the app, see “Image Acquisition Explorer Overview” on page 3-2.

1. After opening the **Image Acquisition Explorer** app, decide which device you want to work with. The image acquisition devices currently connected to your computer are shown. If you do not see your device, make sure you have the appropriate Image Acquisition Toolbox support package installed. For a list of supported hardware and their respective support packages, see “Image Acquisition Toolbox Supported Hardware”. If your device is not connected yet, plug it in to your computer and refresh the list of hardware in the app.

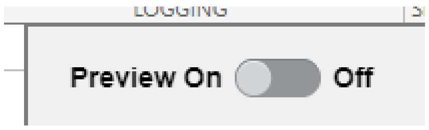


2. Choose the format to work with by selecting an option from **Video Format** in the **Configure Format** section of the app toolstrip. The formats might correspond to the different resolutions and color spaces that your device supports, or to different video standards or camera configurations. This information comes from your device adaptor.

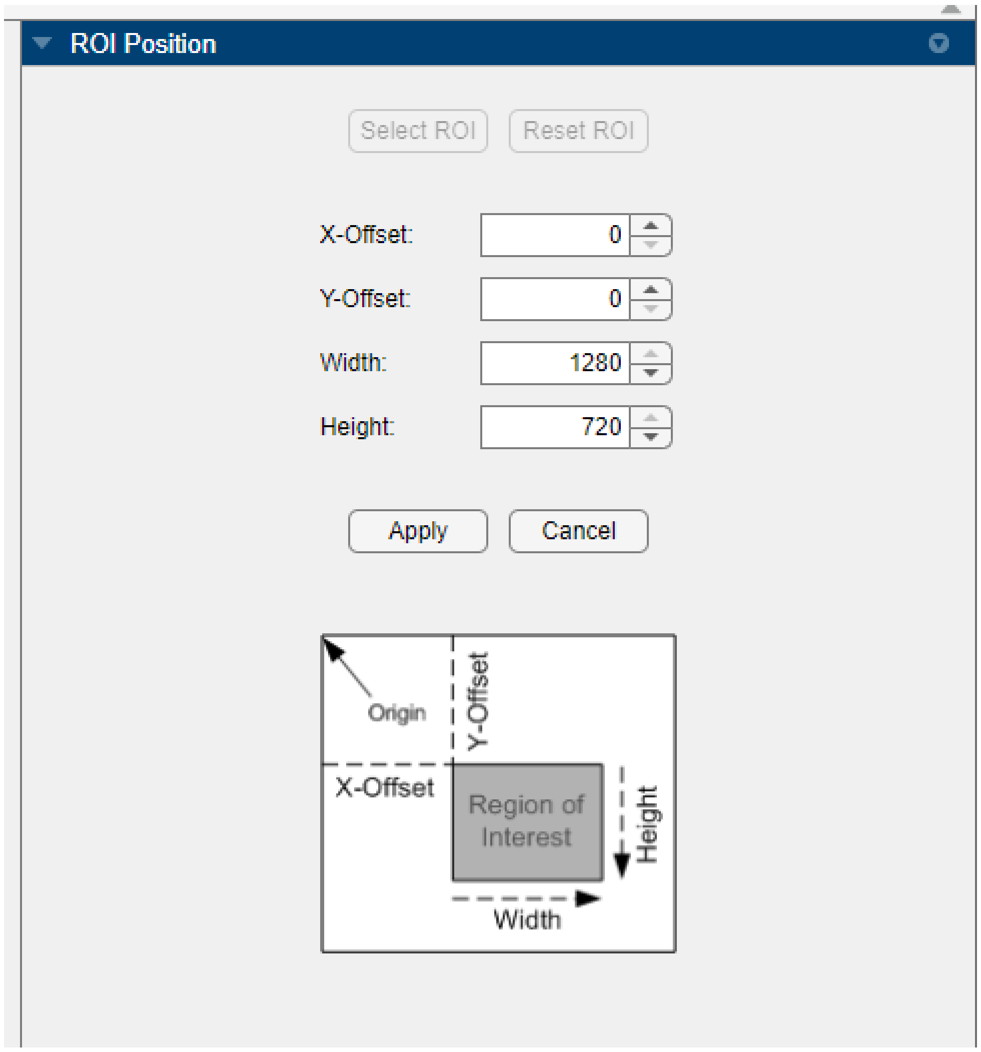


   If your device supports camera files, the **Select Camera File** button is available instead of **Video Format**.

3. Look at the preview in the app to check that the device is working and the image is what you expect. If **Preview** is **Off**, you can toggle the switch to **On**.

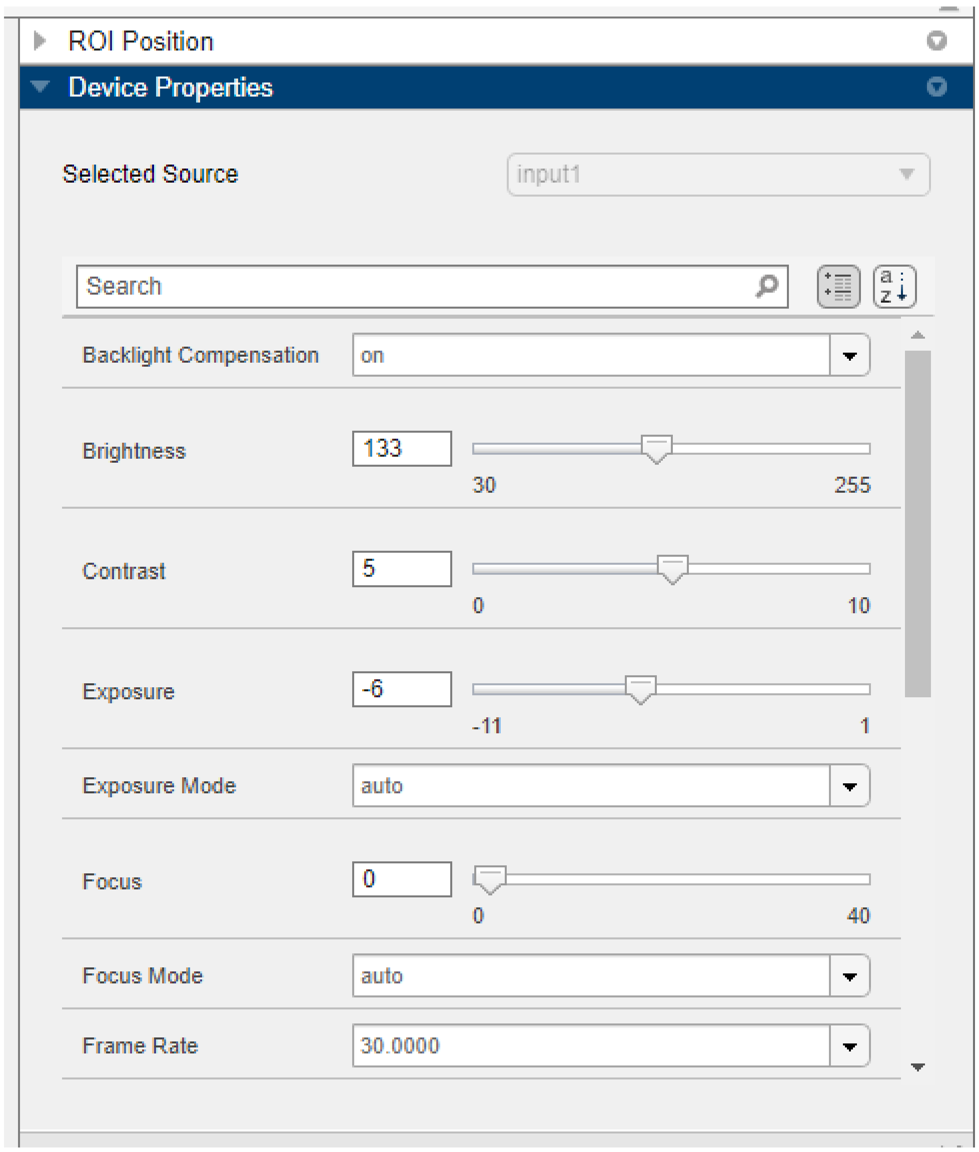


4 If necessary, you can physically adjust the device to achieve the desired image area. Optionally, you can define the acquisition region by using the settings in the **ROI Position** panel next to the preview.



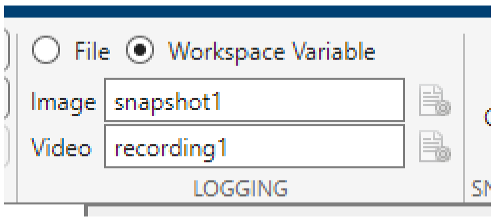
X-Offset: 0
Y-Offset: 0
Width: 1280
Height: 720

5 Set any general or device-specific parameters from the **Device Properties** panel or use the default settings. The specific list of properties that appear depends on your selected device. You can preview the property changes as you update them.



Selected Source	input1
Backlight Compensation	on
Brightness	133 (30–255)
Contrast	5 (0–10)
Exposure	-6 (-11–1)
Exposure Mode	auto
Focus	0 (0–40)
Focus Mode	auto
Frame Rate	30.0000

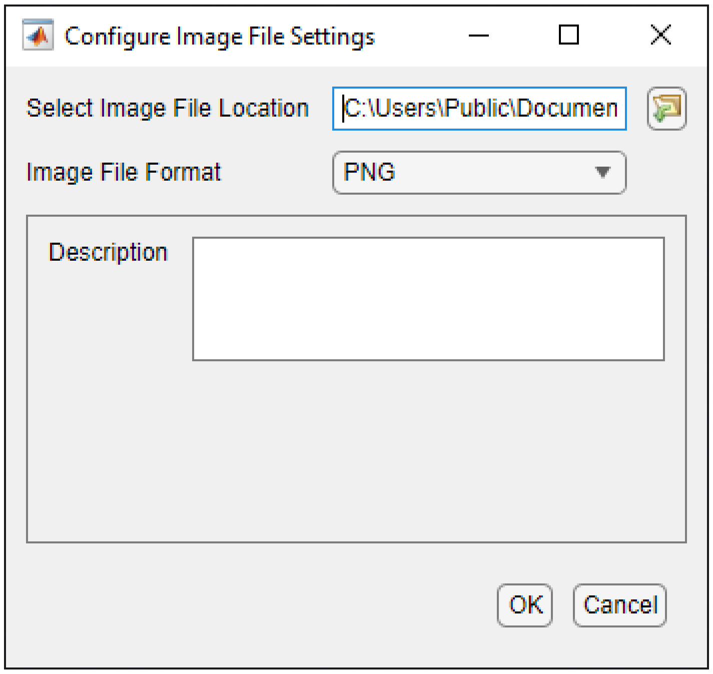
6 Choose your logging mode, which determines whether you save the acquisition data as a file or as a workspace variable. Select either the **File** or **Workspace Variable** option in the **Logging** section of the app toolstrip. Specify the file name or workspace variable name.



File [✓ Workspace Variable]

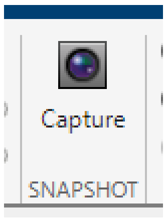
Image	snapshot1
Video	recording1

If you select **File**, you can click the configuration icon next to the file name for additional settings, including file location to save to and file format to save as.

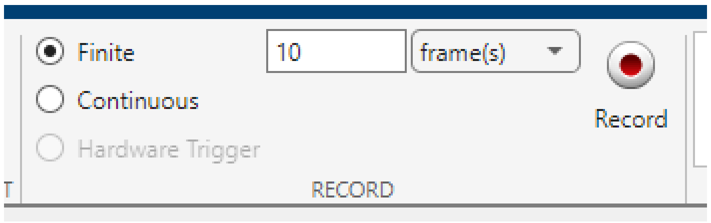


7 Decide whether you want to capture a snapshot of a single frame or record a video of multiple frames.

- If you want to immediately capture a single frame, click the **Capture** button in the **Snapshot** section of the app toolstrip. The image data is saved as an image file or workspace variable, depending on your prior selection.



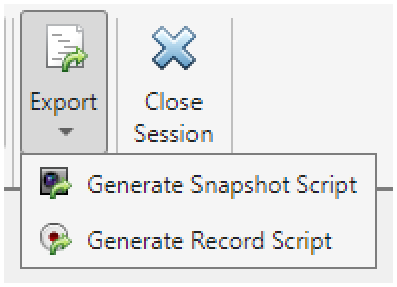
- If you want to record a video or acquire a sequence of multiple frames, you can select a recording mode of **Finite** or **Continuous** in the **Record** section of the app toolstrip. If your device supports hardware triggered acquisition, you also have the **Hardware Trigger** option available. For finite recording, specify the number of frames or seconds to record for and click the **Record** button. For continuous recording, you can start by clicking the **Record** button. It becomes a **Stop** button, which you can click to end recording. The recorded data is saved as a video file or workspace variable, depending on your prior selection.



8 You can generate a MATLAB live script that includes the device and acquisition configurations that you currently have in the app. Click the **Export** button in the app toolstrip to select

`Generate Snapshot Script` or `Generate Record Script`. Both these options create and open a live script that contains code for the current device configuration, as specified in the **Configure Format** section, and code for saving data as a file or workspace variable, as specified in the **Logging** section.

- The `Generate Snapshot Script` option creates and opens a live script that contains code for connecting to your device, configuring its properties, capturing a single frame, and viewing a snapshot of the captured image.
- The `Generate Record Script` option creates and opens a live script that contains code for connecting to your device, configuring its properties, recording a specified number of frames, and viewing the recorded video.



See Also

Image Acquisition Explorer

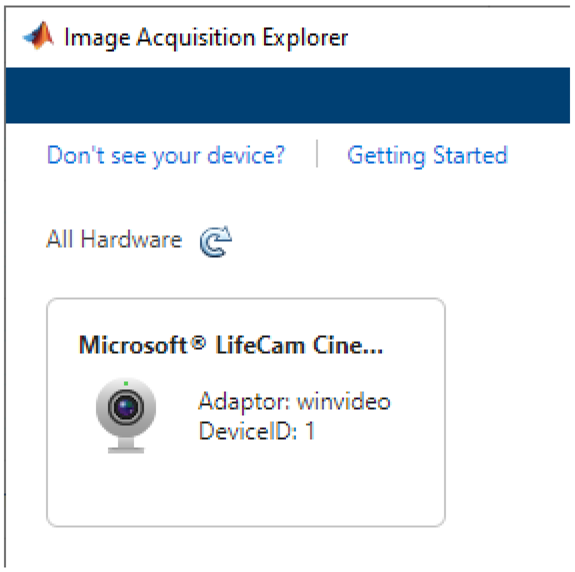
Related Examples

- “Image Acquisition Explorer Overview” on page 3-2
- “Select Your Device and Configure Format in Image Acquisition Explorer” on page 3-10
- “Set Acquisition Parameters in Image Acquisition Explorer” on page 3-13
- “Log Data in Image Acquisition Explorer” on page 3-18
- “Preview and Acquire Data in Image Acquisition Explorer” on page 3-22
- “Export Code from Image Acquisition Explorer” on page 3-27

Select Your Device and Configure Format in Image Acquisition Explorer

Select Device

When you open the **Image Acquisition Explorer** app, you can select an image acquisition device that is currently connected to your computer. Each device card has the name of the device, name of the adaptor it is using, and the device ID for the adaptor.



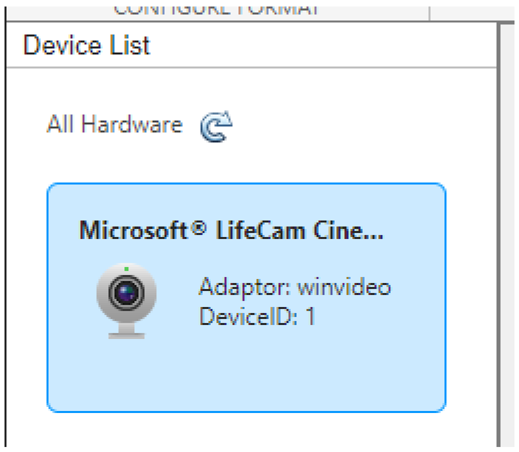
If you do not see your device listed here, make sure that you have the appropriate Image Acquisition Toolbox support package installed for your device by clicking the **Don't see your device?** link. For a list of supported hardware and their respective support packages, see “Image Acquisition Toolbox Supported Hardware”. Check that it is physically plugged in to your computer. You can also try clicking the refresh button next to **All Hardware**.

Add New Hardware

When you open the **Image Acquisition Explorer**, the app automatically shows the image acquisition devices supported by the toolbox that are currently connected to your computer. If you plug a new device in while the **Image Acquisition Explorer** is open, click the refresh button next to **All Hardware** to display the new device.

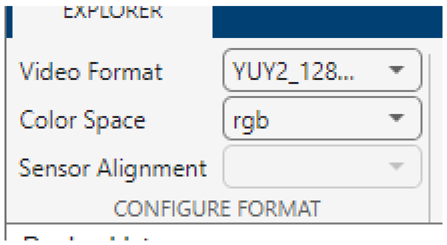
View Device List

After you select a device, the main app space opens. The **Device List** panel shows your selected device as well as the other devices you can connect to from the **Image Acquisition Explorer** app. You can refresh the list of devices and switch to another device at any time. However, doing so will discard your current device configuration.



Configure Device Format

Use the parameters in the **Configure Format** section of the **Image Acquisition Explorer** app toolstrip to define the **Video Format** and **Color Space**. The available values for both of these parameters depend on the selected device. The video format defines different resolutions and color spaces that your device supports, or different video standards or camera configurations for your device.



Video Format: YUY2_128...
Color Space: rgb
Sensor Alignment:

Video Format

Use this parameter to set the video format used by the device to capture images and video. The list of values for this parameter depends on the video formats supported by your device. The format selected when you open the app is the device's default format.

Color Space

Use this parameter to set the color space for the selected video format. Possible values for **Color Space** are `grayscale`, `rgb`, `YCbCr`, and `bayer`, but the list of values that you see depends on the **Video Format** you selected. Your device format's default color space is shown as the default.

If you select `grayscale` for the **Color Space**, you can set the **Colormap** and **Color Limits** parameters for the preview.

- **Color Limit** — Toggle this switch to **Manual** to set the minimum and maximum values on the specified colormap. The default values are 0 and 255. All values in the preview that are less than or equal to the minimum value map to the lowest value of the colormap. All values in the preview that are greater than or equal to the maximum value map to the highest value of the colormap.
- **Colormap** — Colormap applied to the preview. For a full list of options, see `map`.

Sensor Alignment

If your device supports Bayer sensor alignment and you select `bayer` for the **Color Space**, you can set the **Sensor Alignment**. Select the 2-by-2 pixel Bayer color filter array pattern of the Bayer color filter array, as `gbrg`, `grbg`, `bggr`, or `rggb`. The specified pattern is used to convert the Bayer pattern image to RGB.

Use Camera File

If your device supports the use of a camera file, also known as a device configuration file, you can select it in the **Image Acquisition Explorer**. For example, some frame grabbers support these files. The camera file is provided by the device manufacturer. See your device documentation for more information.

After you select your device, there is a **Select Camera File** button in the **Configure Format** section of the app toolstrip if the device supports the use of camera files.

To use a camera file:

1. In the **Configure Format** section of the app toolstrip, click **Select Camera File** to open a file browser window.
2. In the file browser window, navigate to the file location and click on it. Then click **Open**.

   The **Camera File** name appears in the toolstrip. You can hover your cursor over it to see the full file path. You can then set the **Color Space**, specify device properties, preview, and acquire data.

Note The app ignores hardware trigger configurations included in a camera file. To configure hardware triggering, select the **Hardware Trigger** option in the **Record** section of the app toolstrip.

See Also
Image Acquisition Explorer

Related Examples
- "Set Acquisition Parameters in Image Acquisition Explorer" on page 3-13
- "Log Data in Image Acquisition Explorer" on page 3-18
- "Preview and Acquire Data in Image Acquisition Explorer" on page 3-22
- "Export Code from Image Acquisition Explorer" on page 3-27

Set Acquisition Parameters in Image Acquisition Explorer

Before acquiring images or video in the **Image Acquisition Explorer** app, set acquisition parameters such as region of interest (ROI) and device-specific properties. The preview updates in real-time as you change these parameters.

If your device supports hardware triggering, you can also set hardware triggering options before acquiring images or video. These options are only available when you select **Hardware Trigger** in the toolstrip.

After specifying acquisition parameters, you can specify where to save your captured image or video data. For more information, see “Log Data in Image Acquisition Explorer” on page 3-18. Then, you can capture image snapshots and record video. For more information, see “Preview and Acquire Data in Image Acquisition Explorer” on page 3-22.

Acquisition Parameters Panels

Set acquisition parameters in **Image Acquisition Explorer** from the following panels next to the preview.

- **ROI Position** — Define the region of interest.
- **Device Properties** — Change device-specific properties.

If your device supports hardware triggering and you select **Hardware Trigger** in the toolstrip, the following panel is also available.

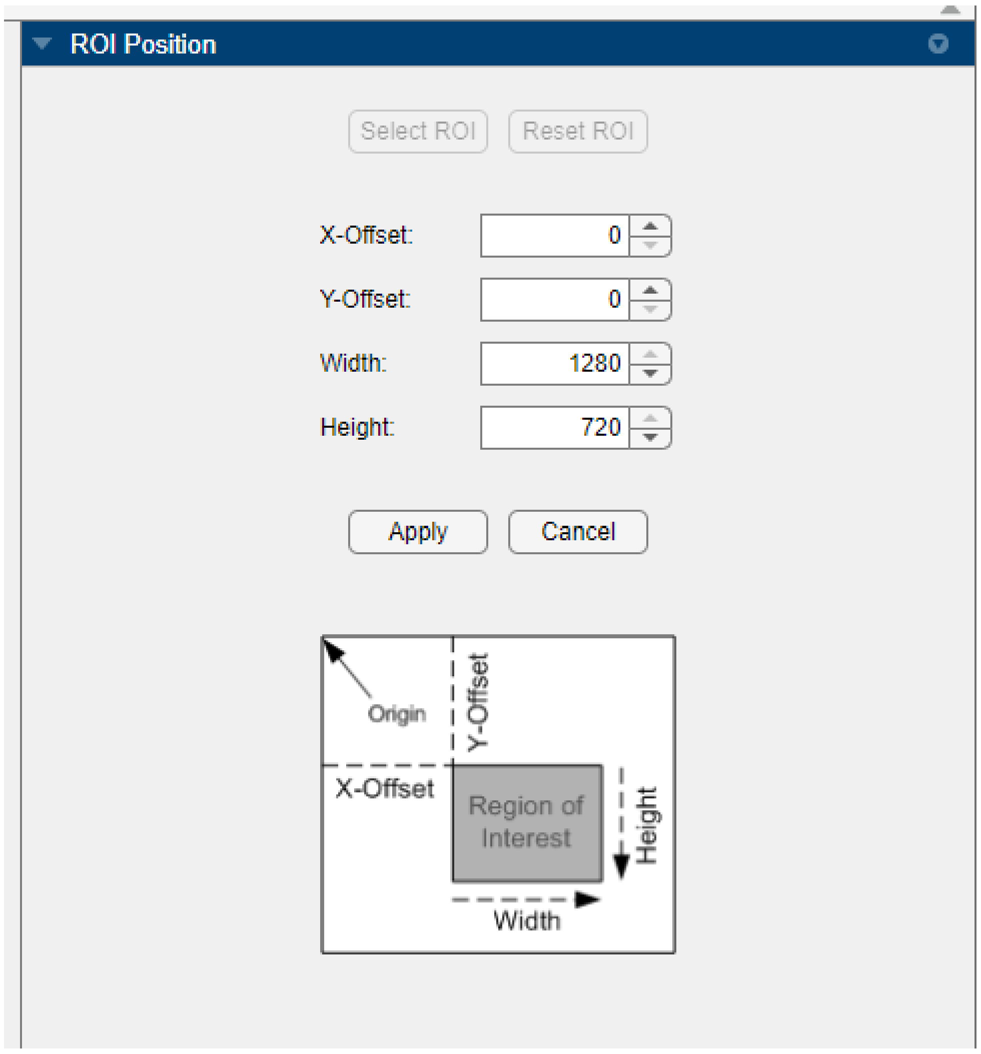
- **Hardware Trigger** — Set up hardware triggering options.

Set Region of Interest

By default, your acquisition consists of the entire frame that the device acquires, which is equal to the selected format's default resolution. If you want to acquire a portion of the frame, use the **ROI Position** panel to set the desired region. The ROI window defines the actual size of the frame logged by the app, measured with respect to the top-left corner of an image frame.

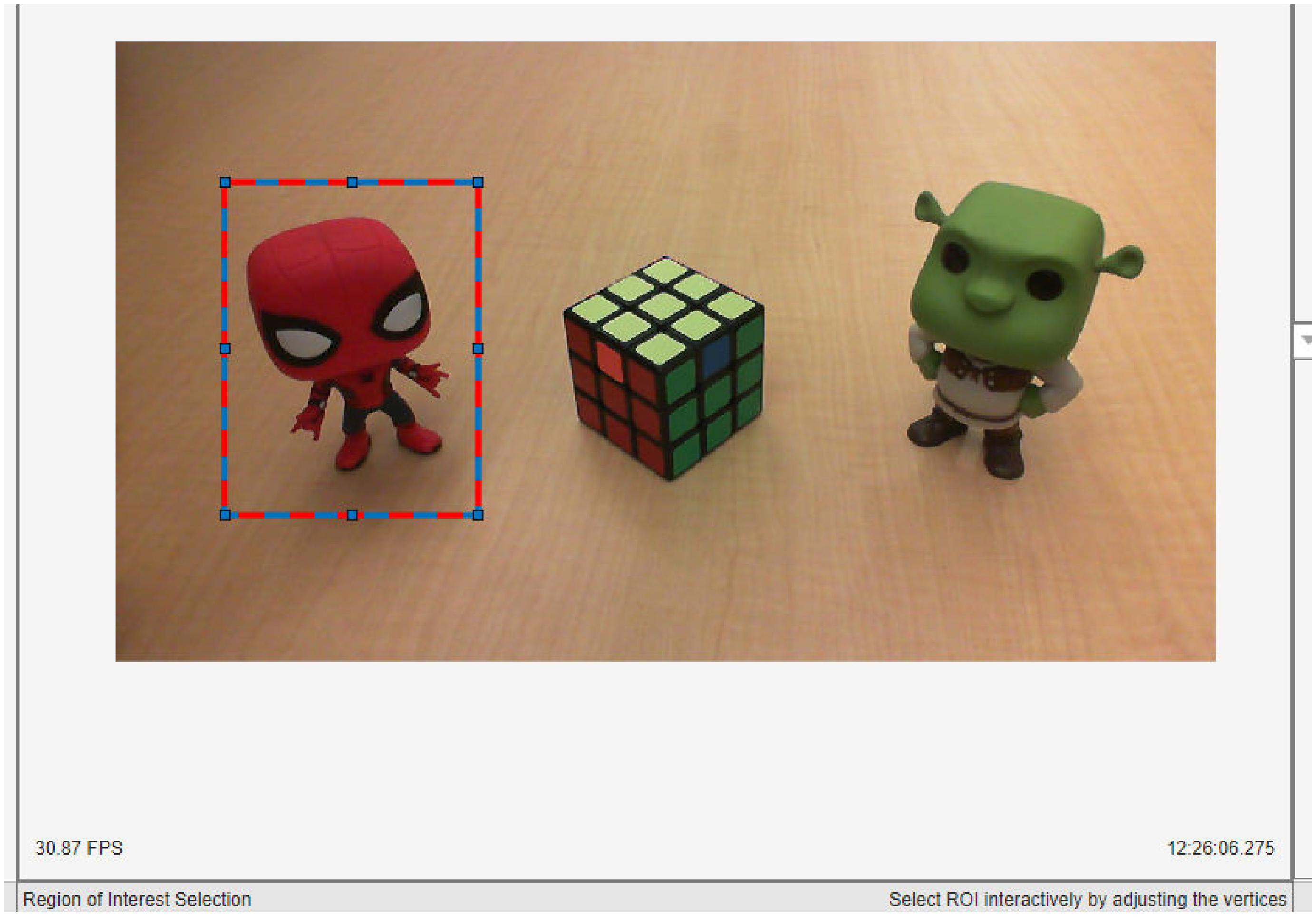
Click the **Select ROI** button to adjust the ROI selection window manually or interactively.

- Set the ROI manually by specifying one or more of the **X-Offset**, **Y-Offset**, **Width**, or **Height** values until you reach the desired region. The ROI selection window in the preview resizes as you make changes.



X-Offset: 0
Y-Offset: 0
Width: 1280
Height: 720

- Set the ROI interactively by clicking and dragging the edges of the selection window that appears in the preview. You can resize and move the selection window to outline the region you want to capture. The values for **X-Offset**, **Y-Offset**, **Width**, and **Height** update as you adjust the selection window. You can manually specify these values as well while you interactively change the selection window.


30.87 FPS
12:26:06.275
Region of Interest Selection
Select ROI interactively by adjusting the vertices

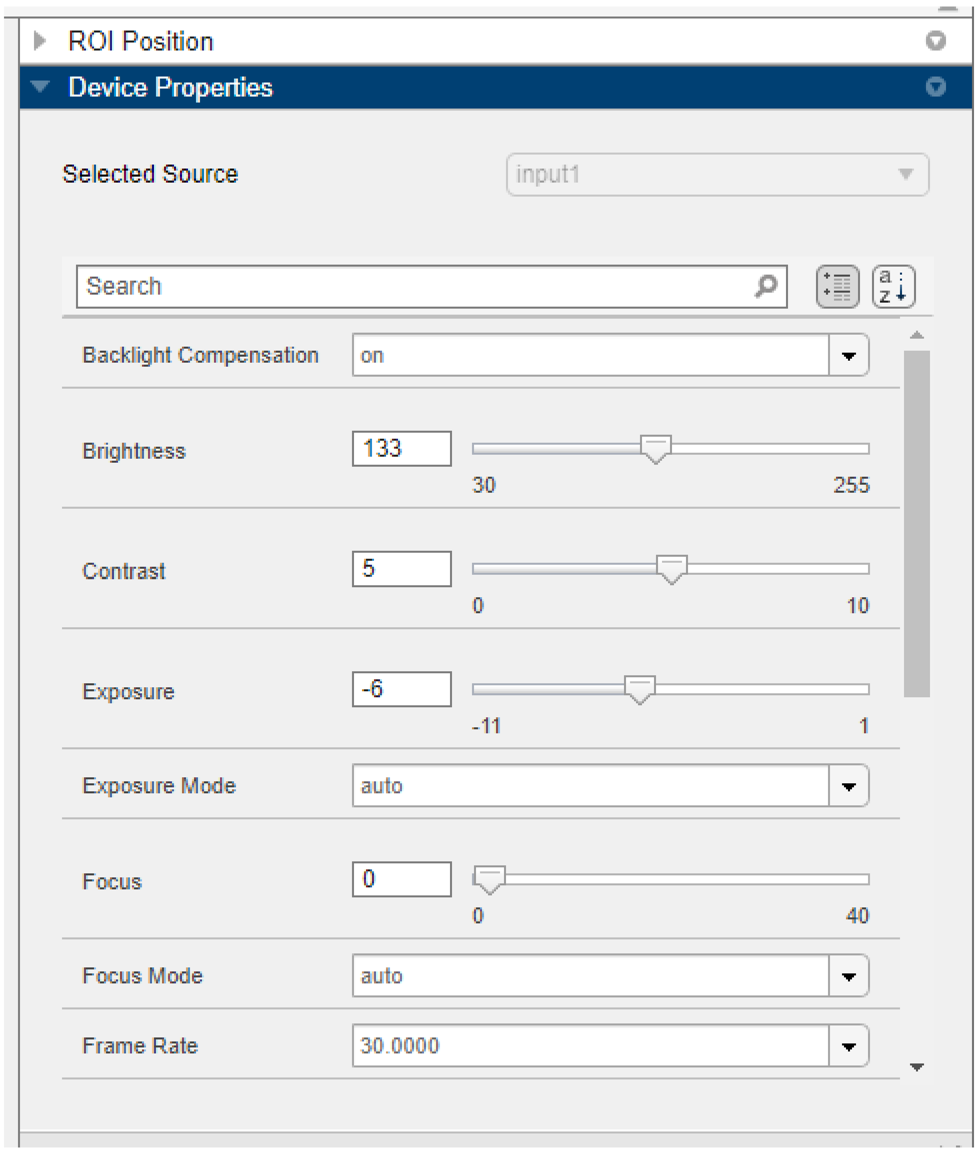
When you are satisfied with the selected ROI, click **Apply** to confirm your selection. After applying your changes, you can reset the ROI to your device's default at any time by clicking **Reset ROI**.

Set Device-Specific Parameters

View or change device-specific properties using the **Device Properties** panel. The specific properties that appear depend on your device.

For example, if **FrameRate** appears in the **Device Properties**, that means your device has a `FrameRate` property. The information in the **Device Properties** panel comes from your device. The value set there is the frame rate that your device uses, in frames per second. If `FrameRate` does not appear in the panel, your device does not support that property.

The **Selected Source** specifies the name of the selected source for the current device. Many device adaptors have only one input source. If your device supports multiple source names, they appear in the drop-down list.



Selected Source: input1

Backlight Compensation: on

Brightness: 133 (30–255)

Contrast: 5 (0–10)

Exposure: -6 (-11–1)

Exposure Mode: auto

Focus: 0 (0–40)

Focus Mode: auto

Frame Rate: 30.0000

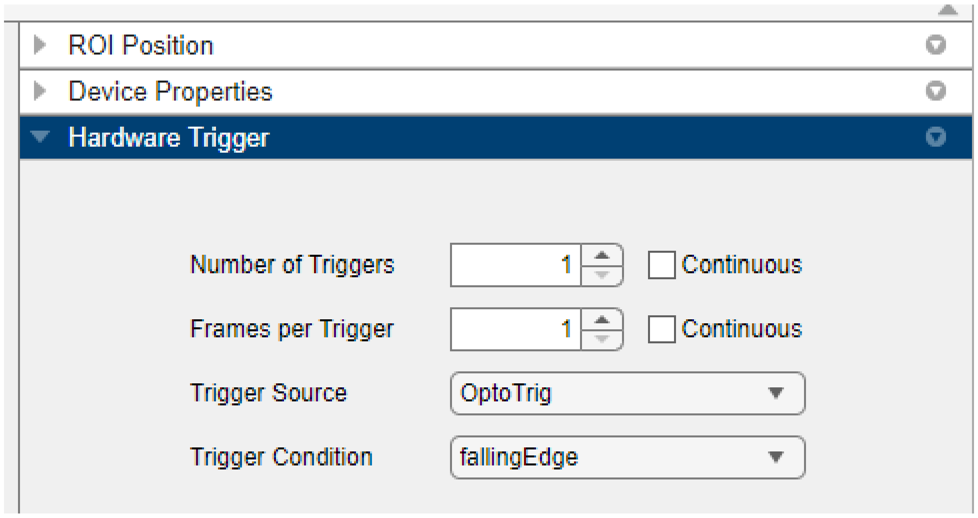
Set Up Hardware Triggering

The **Hardware Trigger** panel is available only for devices that support hardware triggering. It appears when you select the **Hardware Trigger** option in the **Record** section of the app toolstrip. You can set the following parameters in this panel.

- **Number of Triggers** — Number of triggers before acquisition is completed.
- **Frames per Trigger** — Number of frames to acquire per trigger.
- **Trigger Source** — Hardware source that is monitored for trigger conditions. When the condition specified in **Trigger Condition** is met, the trigger is executed and the acquisition starts. **Trigger Source** is device-specific. The drop-down list shows the mechanisms your particular device can use to receive triggers from the hardware source.

- **Trigger Condition** — Condition that must be met from the **Trigger Source** before a trigger event occurs. **Trigger Condition** is device-specific. The drop-down list shows the conditions that your particular device can wait for from the hardware source.

The total number of frames that are acquired when you start an acquisition depends on **Number of Triggers** and **Frames Per Trigger**. For example, if you set **Number of Triggers** to 2 and **Frames Per Trigger** to 4, the total number of frames in the acquisition is 8.



Number of Triggers 1 ☐ Continuous
Frames per Trigger 1 ☐ Continuous
Trigger Source OptoTrig
Trigger Condition fallingEdge

For more information about hardware triggers and how to use them with a `videoinput` object, see “Using a Hardware Trigger” on page 6-12.

See Also

Image Acquisition Explorer

Related Examples

- “Select Your Device and Configure Format in Image Acquisition Explorer” on page 3-10
- “Log Data in Image Acquisition Explorer” on page 3-18
- “Preview and Acquire Data in Image Acquisition Explorer” on page 3-22
- “Export Code from Image Acquisition Explorer” on page 3-27

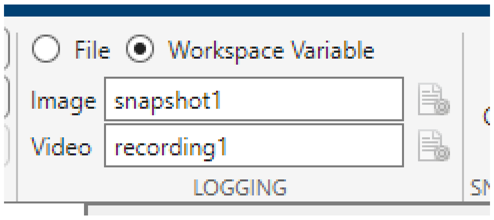
Log Data in Image Acquisition Explorer

You can use the **Image Acquisition Explorer** to save acquired image data to a file or as a workspace variable. If you want to save your image data to a file, you can specify the file format and configure other file settings.

After you specify your data logging preferences, you can capture image snapshots and record video. For more information, see “Preview and Acquire Data in Image Acquisition Explorer” on page 3-22.

Log Data to Workspace

To save image data as a workspace variable, select the **Workspace Variable** option in the **Logging** section of the app toolstrip. You can specify the variable name as a valid MATLAB variable name that does not already exist in the workspace.



File [✓ Workspace Variable]

Image: snapshot1

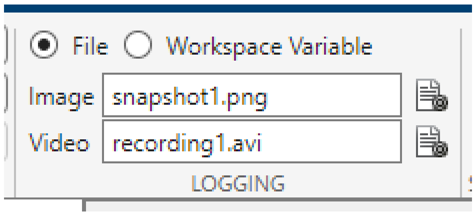
Video: recording1

The default values are `snapshot1` and `recording1` for **Image** and **Video**, respectively. After you capture a snapshot or record a video using those variables, the default variable names update to `snapshot2` or `recording2`, then `snapshot3` or `recording3`, and so on.

After you set your variable names, you can click the **Capture** button to save image data to the workspace or the **Record** button to save video data to the workspace. For more information, see “Preview and Acquire Data in Image Acquisition Explorer” on page 3-22.

Log Data to File

To save image data to file, select the **File** option in the **Logging** section of the app toolstrip. You can then specify file names and additional configuration settings for **Image** and **Video** logging.



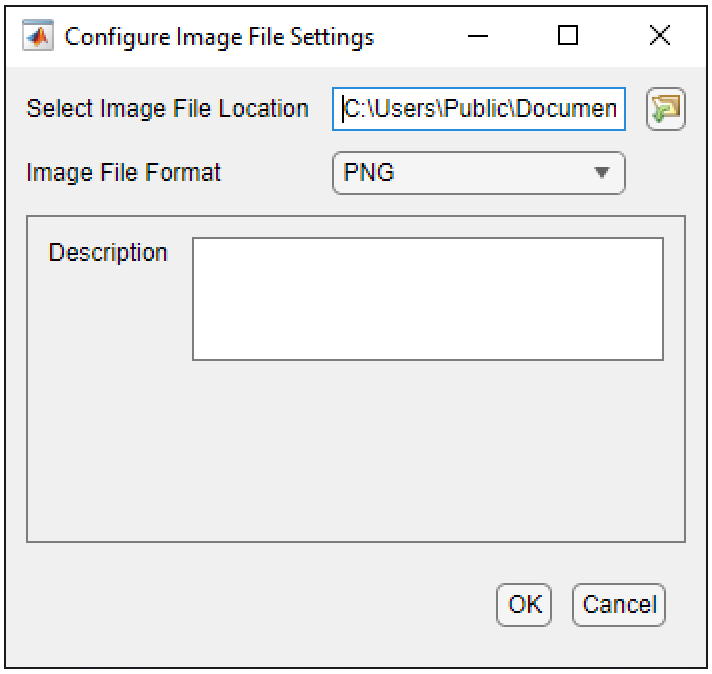
[✓ File] Workspace Variable

Image: snapshot1.png

Video: recording1.avi

Image Data

In the **Image** field, specify the name you want to give the file. The default value is `snapshot1.png`. Click the settings icon next to the file name to configure additional file settings. You can **Select Image File Location** and select an **Image File Format**. Available settings depend on the specified file format.



Select Image File Location: C:\Users\Public\Documen

Image File Format: PNG

Description

If you select PNG, you can specify the following.

- **Description** — Add description to image.

If you select TIFF, you can specify the following.

- **Compression** — Select compression scheme as packbits, none, lzw, or deflate.
- **Description** — Add description to image.

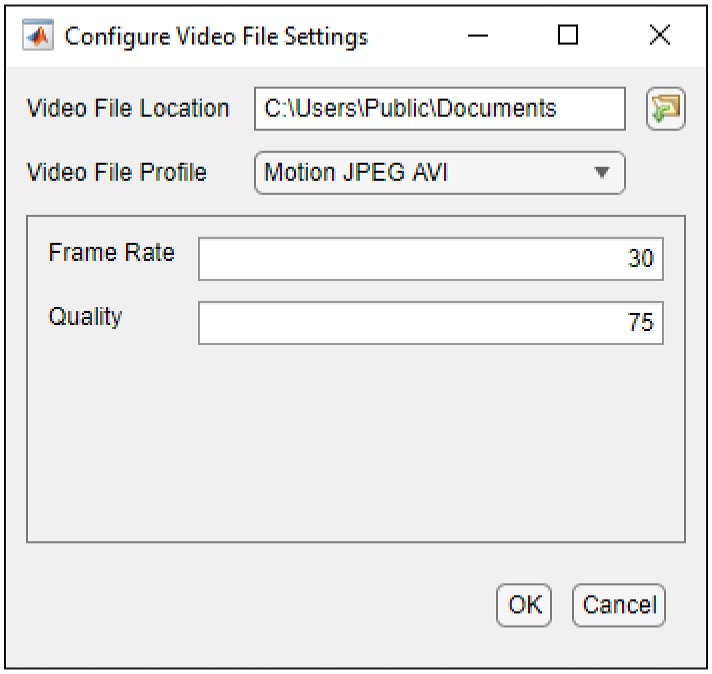
If you select JPEG, you can specify the following.

- **Quality** — Specify quality of output file from 0 to 100, where 0 is lower quality and higher compression and 100 is higher quality and lower compression. The default value is 75.
- **Bit Depth** — Select number of bits per pixel as 8 or 12.
- **Comment** — Add comment to image.

After you specify the file name, location, and other settings, you can click the **Capture** button to save image data to file. For more information, see “Preview and Acquire Data in Image Acquisition Explorer” on page 3-22.

Video Data

In the **Video** field, specify the name you want to give the file. The default value is recording1.avi. Click the settings icon next to the file name to configure additional file settings. You can select a **Video File Location** and a **Video File Profile**. Possible values for **Video File Profile** are Archival, Motion JPEG AVI, MPEG-4, Motion JPEG 2000, and Uncompressed AVI. Available settings depend on the specified file profile.



Video File Location C:\Users\Public\Documents

Video File Profile Motion JPEG AVI

Frame Rate 30

Quality 75

If you select `Archival` or `Motion JPEG 2000`, you can specify the following.

- **Lossless Compression** — Turn on to make decompressed data identical to input data. The default is off for the `Motion JPEG 2000` profile, and on for the `Archival` profile.
- **Compression Ratio** — Specify the target ratio between number of bytes in the input image and number of bytes in the compressed image as a number greater than 1. The default value is `10`. You can modify this setting only if **Lossless Compression** is turned off.
- **Frame Rate** — Specify the rate of video playback in frames per second. The default value is `30`.
- **MJ2 Bit Depth** — Select the number of least-significant bits in input image data as a number from 1 to 16.

If you select `Motion JPEG AVI` or `MPEG-4`, you can specify the following.

- **Frame Rate** — Specify rate of video playback in frames per second. The default value is `30`.
- **Quality** — Specify quality of output file from 0 to 100, where 0 is lower quality and higher compression and 100 is higher quality and lower compression. The default value is `75`.

If you select `Uncompressed AVI`, you can specify the following.

- **Frame Rate** — Specify rate of video playback in frames per second. The default value is `30`.

After you specify the file name, location, and other settings, you can click the **Record** button to save video data to file. For more information, see “Preview and Acquire Data in Image Acquisition Explorer” on page 3-22.

Note If the value of the frame rate specified in the video file settings is different from the frame rate of the preview, the length of the saved video will be different from the specified number of seconds for finite recording. The number of frames saved in the recording is calculated from the frame rate of the preview multiplied by the specified number of seconds to record. (The specified amount of time to record also includes the time required for the acquisition to start.) The length of the saved video is the number of frames divided by the frame rate specified in the video file settings. For example, if the frame rate of the preview is 15 frames per second, the number of seconds to record is 20 seconds, and the frame rate of video playback is 30 frames per second, the length of the saved recording is approximately 10 seconds.

See Also

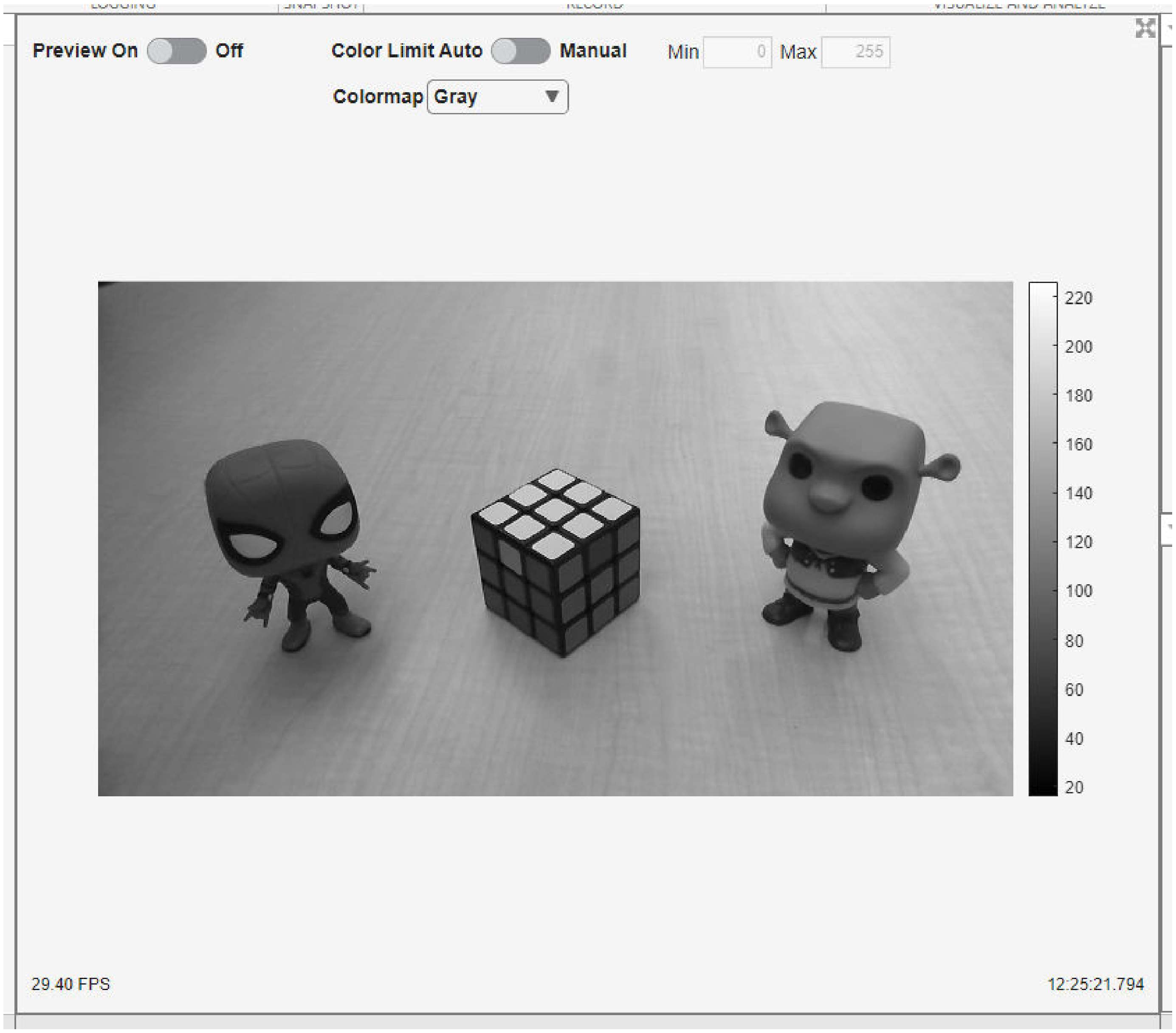
Image Acquisition Explorer

Related Examples

- “Select Your Device and Configure Format in Image Acquisition Explorer” on page 3-10
- “Set Acquisition Parameters in Image Acquisition Explorer” on page 3-13
- “Preview and Acquire Data in Image Acquisition Explorer” on page 3-22
- “Export Code from Image Acquisition Explorer” on page 3-27

Preview and Acquire Data in Image Acquisition Explorer

The **Image Acquisition Explorer** has a preview that displays the image data from your image acquisition device when you preview or acquire data.



Turn off the preview by toggling the **Preview** switch to **Off**.

If you set the **Color Space** to `grayscale` in the toolstrip, you can set the colormap and color limits.

- **Color Limit** — Toggle this switch to **Manual** to set the minimum and maximum values on the specified colormap. The default values are 0 and 255. All values in the preview that are less than or equal to the minimum value map to the lowest value of the colormap. All values in the preview that are greater than or equal to the maximum value map to the highest value of the colormap.

- **Colormap** — Colormap applied to the preview. For a full list of options, see `map`.

The area below the preview displays the current frame rate and the timestamp of the last frame in the preview. The status bar below the preview displays status messages when you capture an image snapshot or record a video. The messages provide information about the current acquisition mode, the number of frames or seconds captured, and the file name or workspace variable name that the data is saved as. The status message also indicates when the app has completed capturing a snapshot or recording a video.

Note Previewing data using software OpenGL® instead of graphics hardware that supports a hardware-accelerated implementation of OpenGL can cause performance issues for the app and is not recommended. For more information, see “System Requirements for Graphics”.

Set Up Preview for Acquisition

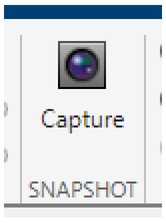
Before acquiring data by taking a snapshot or recording a video, you can preview the image data in the app. Modify acquisition parameters and device properties and see the changes update in real-time in the preview.

1. Make sure that the correct device is selected in the **Device List**. The **Device List** shows the image acquisition devices currently connected to your system. If the device you want to use is not connected to your system, plug it in and refresh the list. Then, select the new hardware. For more information, see “Select Device” on page 3-10.
2. Select the **Video Format** and **Color Space** from the **Configure Format** section of the app toolstrip. The video formats might correspond to the different resolutions and color spaces that your device supports, or to different video standards or camera configurations. This information comes from your device adaptor. Select the format you want to use. For more information, see “Configure Device Format” on page 3-11.
3. Look at the preview to test and set up your device. If necessary, physically adjust the device to achieve the desired image area, or use the settings in the **ROI Position** panel to constrain the image. For more information, see “Set Region of Interest” on page 3-13.
4. Set additional properties in the **Device Properties** panel to adjust the quality of the image or other acquisition factors. For more information, see “Set Device-Specific Parameters” on page 3-15.

For more information about logging, see “Log Data in Image Acquisition Explorer” on page 3-18.

Capture Image Snapshot

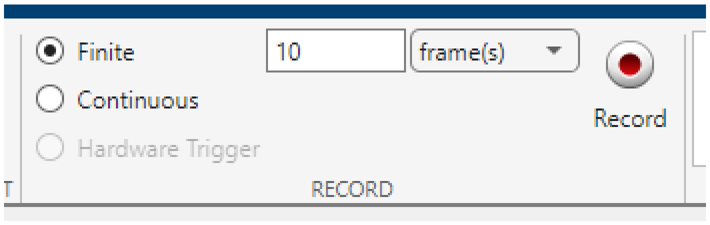
After you are satisfied with the image that you see in the preview, you can capture and save image data. Click the **Capture** button in the **Snapshot** section of the app toolstrip. Clicking this button immediately captures a single image frame and saves it as an image file or as a workspace variable, depending on your selection of **File** or **Workspace Variable** in the **Logging** section.



Record Video

You can save image data from your image acquisition device as a video recording. There are four workflows for recording video in the **Image Acquisition Explorer** app.

- "Record Finite Number of Frames" on page 3-24
- "Record for Finite Duration" on page 3-24
- "Record Continuously" on page 3-25
- "Record with Hardware Trigger" on page 3-25



Record Finite Number of Frames

Capture a finite number of frames of image data from your image acquisition device.

1. Select the **Finite** option in the **Record** section of the app toolstrip. Selecting this option allows you to specify the number of frames you want to record.
2. Select the `frame(s)` option from the dropdown next to **Finite**.
3. Enter an integer value for the number of frames to capture in the field to the left of `frame(s)`.
4. Click the **Record** button to acquire the specified number of frames and save them as a video file or as a workspace variable, depending on your selection of **File** or **Workspace Variable** in the **Logging** section. The **Record** button becomes a **Stop** button after you click it.
5. Wait for the specified number of frames to be recorded. You can also end recording at any time before the specified number of frames is captured by clicking **Stop**. Clicking **Stop** saves the data that has already been captured.

While you are recording, the app toolstrip and all property panels are disabled. You cannot change the value of any parameters during recording.

Record for Finite Duration

Capture a specified number of seconds of image data from your image acquisition device.

1. Select the **Finite** option in the **Record** section of the app toolstrip. Selecting this option allows you to specify the amount of time you want to record.
2. Select the `second(s)` option from the dropdown next to **Finite**.
3. Enter a numeric value for the number of seconds to capture in the field to the left of `second(s)`.

Note The length of the saved recording might be less than the amount of time specified here since this value includes the time required for the acquisition to start. After you click the **Record** button, your device might require some time to start the acquisition. If your recording does not capture all the frames you want it to, try increasing the recording duration and recording again.

4. Click the **Record** button to acquire frames for the specified number of seconds and save them as a video file or as a workspace variable, depending on your selection of **File** or **Workspace Variable** in the **Logging** section. The **Record** button becomes a **Stop** button after you click it.
5. Wait for the specified number of seconds to be recorded. You can also end recording at any time before the specified number of seconds is captured by clicking **Stop**. Clicking **Stop** saves the data that has already been captured.

While you are recording, the app toolstrip and all property panels are disabled. You cannot change the value of any parameters during recording.

Record Continuously

Capture image data for an infinite duration from your image acquisition device.

1. Select the **Continuous** option in the **Record** section of the app toolstrip.
2. Click the **Record** button to continuously acquire frames and save them as a video file or as a workspace variable, depending on your selection of **File** or **Workspace Variable** in the **Logging** section. The **Record** button becomes a **Stop** button after you click it.
3. You can end recording at any time by clicking **Stop** and saving the data that has already been captured.

While you are recording, the app toolstrip and all property panels are disabled. You cannot change the value of any parameters during recording.

Record with Hardware Trigger

Capture frames with a hardware trigger.

1. Make sure that the **Trigger Mode** property is set to `On` and other **Trigger Selector** properties are configured correctly in the **Device Properties** panel.
2. Select the **Hardware Trigger** option in the **Record** section of the app toolstrip.
3. Configure the hardware trigger properties in the **Hardware Trigger** panel.
   - **Number of Triggers**
   - **Frames per Trigger**
   - **Trigger Source**
   - **Trigger Condition**
4. Click the **Record** button to start acquiring frames when the trigger condition is met. The image data is saved as a video file or as a workspace variable, depending on your selection of **File** or **Workspace Variable** in the **Logging** section. The **Record** button becomes a **Stop** button after you click it.
5. Wait for the specified **Number of Triggers** to be reached for the recording to complete. You can also end recording at any time by clicking **Stop** and saving the data that has already been captured.

While you are recording, the app toolstrip and all property panels are disabled. You cannot change the value of any parameters during recording.

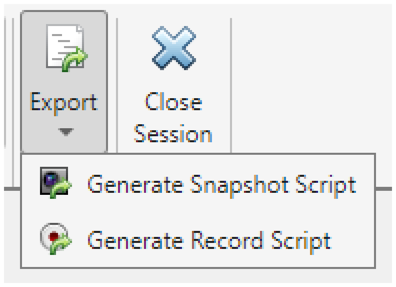
See Also
Image Acquisition Explorer

Related Examples

- “Select Your Device and Configure Format in Image Acquisition Explorer” on page 3-10
- “Set Acquisition Parameters in Image Acquisition Explorer” on page 3-13
- “Log Data in Image Acquisition Explorer” on page 3-18
- “Export Code from Image Acquisition Explorer” on page 3-27

Export Code from Image Acquisition Explorer

The **Image Acquisition Explorer** app allows you to generate a MATLAB live script that includes the device and acquisition configurations that you currently have in the app. Click the **Export** button in the app toolstrip to select `Generate Snapshot Script` or `Generate Record Script`. Both these options create and open a live script that contains code for the current device configuration in the app. You can edit and save the generated live script as necessary for your application.



- The `Generate Snapshot Script` option creates and opens a live script that contains code for connecting to your device, configuring its properties, capturing a single frame, and viewing a snapshot of the captured image.
- The `Generate Record Script` option creates and opens a live script that contains code for connecting to your device, configuring its properties, recording frames, and viewing the recorded video.

Connect and Configure

The generated live script contains the following sections for connection and configuration, with code similar to the following examples.

- **Connect to Device** — Creates a connection to the selected device and specified **Video Format** using the `videoinput` function.

  ```
  v = videoinput("winvideo", 1, "YUY2_1280x720");
  ```
- **Configure Device Properties** — Defines device properties that you select in the app, including **Color Space**, **Sensor Alignment**, and **Region of Interest**. If you do not edit any of these parameters, this section is not in the live script.

  ```
  v.ReturnedColorspace = "rgb";
  ```
- **Configure Device-Specific Properties** — Defines device-specific properties that you specified in the **Device Properties** panel of the app. If you do not make any changes to these parameters, this section is not in the live script.

  ```
  src = getselectedsource(v);
  src.Exposure = 1;
  ```

Generate Snapshot Script

If you select `Generate Snapshot Script`, the live script contains the following additional sections, with examples of code.

- **Capture Image** — Captures a single frame and saves it to the workspace as the variable specified in the **Logging** section. If you selected **File** in the **Logging** section, this section also does the following:

  - Defines the file name to save as and location to save to.
  - Saves the captured frame to a file using the `imwrite` function.
  - Specifies the image file configuration settings that you set in **Logging**, such as file format and quality, as name-value arguments in `imwrite`.

  ```
  image1 = getsnapshot(v);

  % Set the desired file location and name.
  filelocation = "C:\Users\user";
  filename = "snapshot1.jpg";
  fullFilename = fullfile(filelocation, filename);

  % Write image data to file.
  imwrite(image1, fullFilename, "jpg", "Quality", 25, "BitDepth", 12);
  ```
- **View Snapshot** — Displays the captured image using the `imshow` function.

  ```
  imageData = imread(fullFilename);
  f = figure;
  ax = axes(f);
  imshow(imageData, "Parent", ax);
  ```

Generate Record Script

If you select `Generate Record Script` the live script contains the following additional sections, with examples of code.

- **Configure File Logging** — Specifies video file logging settings by doing the following:

  - Defines the file name to save as and location to save to.
  - Specifies the video file configuration settings that you set in **Logging**, such as file format and quality, using `VideoWriter`.
  - Configures the `videoinput` object to log to disk.

  If you do not select **File** in **Logging**, this section is not in the live script.

  ```
  filelocation = "C:\Users\user";
  filename = "recording1.mp4";
  fullFilename = fullfile(filelocation, filename);

  % Create and configure the video writer
  logfile = VideoWriter(fullFilename, "MPEG-4");
  logfile.FrameRate = 15;
  logfile.Quality = 25;

  % Configure the device to log to disk using the video writer
  v.LoggingMode = "disk";
  v.DiskLogger = logfile;
  ```
- **Configure Triggering** — Specifies the hardware trigger settings **Number of Triggers**, **Frames per Trigger**, **Trigger Source**, and **Trigger Condition** that you set in the **Hardware Trigger** panel. If you do not select **Hardware Trigger** in **Record**, this section is not in the live script.

```
framesPerTrigger = 4;
numTriggers = 2;
triggerCondition = "risingEdge";
triggerSource = "TTL";

triggerconfig(v, "hardware", triggerCondition, triggerSource);
v.FramesPerTrigger = framesPerTrigger;
v.TriggerRepeat = numTriggers - 1;
```

- **Record** — Records image data based on the recording mode selected in the **Record** section and saves it using the settings specified in the **Logging** section.

  - If you select **Finite** and `frame(s)`, this section is called **Record Video for Set Number of Frames** and records image data for the specified number of frames.

    ```
    numFrames = 10;
    v.FramesPerTrigger = numFrames;

    start(v);
    wait(v);
    stop(v);
    recording1 = getdata(v, numFrames);
    ```

  - If you select **Finite** and `second(s)`, this section is called **Record Video for Set Number of Seconds** and records image data for the specified number of seconds.

    ```
    numSeconds = 10;
    v.FramesPerTrigger = Inf;

    start(v);
    pause(numSeconds);
    stop(v);
    recording1 = getdata(v, v.FramesAvailable);
    ```

  - If you select **Continuous**, this section is called **Record Continuous Video Data** and records image data continuously and requires you to press Enter to stop recording.

    ```
    v.FramesPerTrigger = Inf;
    start(v);

    % Use INPUT to pause before ending acquisition.
    input("Press ENTER to end acquisition.");
    stop(v);
    recording1 = getdata(v, v.FramesAvailable);
    ```

  - If you select **Hardware Trigger**, this section is called **Record with Hardware Trigger** and records image data using a hardware trigger.

    ```
    start(v);
    wait(v);
    stop(v);
    recording1 = getdata(v, framesPerTrigger * numTriggers);
    ```

- **Show Recording** — Displays the recorded video using the `implay` function.

```
reader = VideoReader(fullFilename);
videoData = read(reader);
implay(videoData);
```

Clean Up

The generated live script contains the following section related to cleaning up the workspace.

- **Clean Up** — Disconnects from the device and clears it from the workspace by using the `delete` and `clear` functions.

  ```
  delete(v)
  clear src v
  ```

See Also

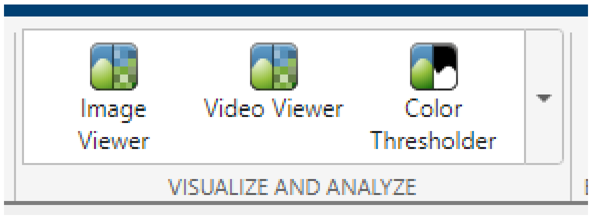
Image Acquisition Explorer

Related Examples

- “Select Your Device and Configure Format in Image Acquisition Explorer” on page 3-10
- “Set Acquisition Parameters in Image Acquisition Explorer” on page 3-13
- “Log Data in Image Acquisition Explorer” on page 3-18
- “Preview and Acquire Data in Image Acquisition Explorer” on page 3-22

Visualize and Analyze Data from Image Acquisition Explorer

You can launch Image Processing Toolbox apps from **Image Acquisition Explorer** and send your image or video data to them for visualization and analysis. Click the **Image Viewer**, **Video Viewer**, or **Color Thresholder** button in the **Visualize and Analyze** section of the app toolstrip to launch the respective app.



The selected app launches with the latest saved image or video data. If you have not yet saved any data, the app still launches.

See Also

Image Acquisition Explorer | **Image Viewer** | **Video Viewer** | **Color Thresholder**

Related Examples

- “Select Your Device and Configure Format in Image Acquisition Explorer” on page 3-10
- “Set Acquisition Parameters in Image Acquisition Explorer” on page 3-13
- “Log Data in Image Acquisition Explorer” on page 3-18
- “Preview and Acquire Data in Image Acquisition Explorer” on page 3-22
- “Export Code from Image Acquisition Explorer” on page 3-27

Saving Image Acquisition Tool Configurations

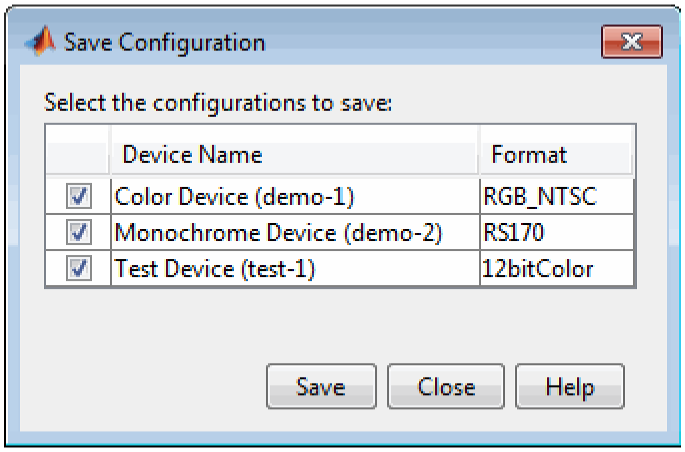
You can save the configuration information about any of your device formats. This includes any parameters you set on any of the tabs in the **Acquisition Parameters** pane. Then when you return to the tool, you can load the configuration so that you do not have to reset those parameters.

To save a configuration:

1. Select **File > Save Configuration**.

   The Save Configuration dialog box opens.
2. Decide what configuration(s) to save.

   The Save Configuration dialog box lists the currently selected device format, as well as any others you selected in the **Hardware Browser** during the tool session. All formats are selected by default, meaning their configurations will be saved. If you do not want to save a configuration, clear it from the list.



	Device Name	Format
✓	Color Device (demo-1)	RGB_NTSC
✓	Monochrome Device (demo-2)	RS170
✓	Test Device (test-1)	12bitColor

3. Click **Save**.

   The Save File dialog box opens.
4. Enter a file name and click **Save**.

The configuration is saved to an Image Acquisition Tool (IAT) file in the location you specified.

You can then open the saved configuration file in a future tool session by selecting **File > Open Configuration**. In the Open Configuration dialog box, browse to an IAT file and click **Open**.

Note You can also export hardware configuration information to other formats such as a MATLAB code file or a MAT-file that can be accessed from MATLAB. See "Exporting Image Acquisition Tool Hardware Configurations to MATLAB" on page 3-33.

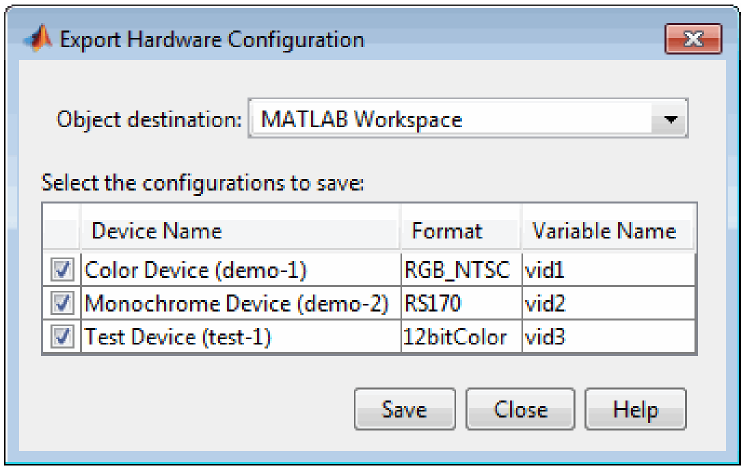
Exporting Image Acquisition Tool Hardware Configurations to MATLAB

You can export the video input objects and their configured parameters from the tool to a choice of multiple formats. You can then access the video object in MATLAB.

To export a hardware configuration:

1 Select **File > Export Hardware Configuration**.

   The Export Hardware Configuration dialog box opens.



Object destination: MATLAB Workspace

Select the configurations to save:

	Device Name	Format	Variable Name
✓	Color Device (demo-1)	RGB_NTSC	vid1
✓	Monochrome Device (demo-2)	RS170	vid2
✓	Test Device (test-1)	12bitColor	vid3

2 Select the file format from the **Object destination** list.

   - `MATLAB Workspace` saves the video input object to the MATLAB Workspace for the duration of the MATLAB session. (You can then save it before exiting MATLAB if you want to retain it.)
   - `MATLAB Code File` is the same as the **File > Generate MATLAB Code File** command. It generates a MATLAB code file containing the video input object and its configured parameters. You could then incorporate the MATLAB code file into other MATLAB code or projects.
   - `MAT-File` saves the video input object and its parameters to a MAT-file.

3 Decide what object configuration(s) to export.

   The Object Exporter dialog box lists the currently selected device format, as well as any others you selected in the **Hardware Browser** during the tool session. All formats are selected by default, meaning their configurations will be saved. If you do not want to save a configuration, clear it from the list.

4 Click **Save**.

   If you exported to the MATLAB Workspace, the dialog box closes and the data is saved to the MATLAB Workspace.

5 If you export to a MAT-file or MATLAB code file, an Export dialog box opens. Select the save location and type a file name, and then click **Save**.

Note You can also save configuration information to an Image Acquisition Tool (IAT) file that can then be loaded in the tool in a future session. See “Saving Image Acquisition Tool Configurations” on page 3-32.

Getting Started with the Image Acquisition Tool

This section describes an example of the basic workflow of using the Image Acquisition Tool to preview, configure, acquire, and save image data. You don't need to do every step shown here, and you can change the order of some steps.

1 Decide which device you want to work with.

   The **Hardware Browser** shows the image acquisition devices currently connected to your system. If the device you want to use is not connected to your system, plug it in and then select **Tools > Refresh Image Acquisition Hardware** to display the new device in the **Hardware Browser**.

2 Choose the format to work with.

   The nodes listed under the device name are the formats the device supports. They may correspond to the different resolutions and color spaces that your device supports, or to different video standards or camera configurations. This information comes from your device adaptor. Select the format you want to use.

3 Preview to check that the device is working and the image is what you expect.

   Click the **Start Preview** button.

   If necessary, physically adjust the device to achieve the desired image area, or use the **Region of Interest** tab to define the acquisition region.

4 Decide how many frames you want to acquire.

   The number of frames that will be acquired when you start the acquisition is dependent on what is set in the **Frames Per Trigger** field on the **General** tab and the **Number of Triggers** field on the **Triggering** tab. For example, if you set **Frames Per Trigger** to 4 and **Number of Triggers** to `2`, the total number of frames acquired will be 8.

   If you just want a snapshot of one frame, leave the default settings of `1` in both of those fields. If you want a specific number of frames, use the fields to set it.

   Alternatively, you can set the tool to acquire continuously and use the buttons in the **Preview Window** to manually start and stop the acquisition. This is discussed in a later step.

5 Set any general or device-specific parameters you need to set, on those tabs of the **Acquisition Parameters** pane, or use the default settings.

6 Choose your log mode, which determines where the acquisition data is stored.

   On the **Logging** tab, use the **Log To** field to choose to log to memory, disk, or both. Disk logging results in a saved VideoWriter file. If you choose memory logging, you can export your data after the acquisition using the **Export Data** button on the **Preview Window**.

   For more information on logging, see the Help for the **Logging** tab in the **Desktop Help** pane in the tool.

7 Start the acquisition by clicking the **Start Acquisition** button.

   – If you set **Trigger Type** (on the **Triggering** tab) to `Immediate`, the tool will immediately start logging data.

– If you set **Trigger Type** to `Manual`, click the **Trigger** button when you want to start logging data.

8 Stop the acquisition.

– If you set **Frames Per Trigger** (on the **General** tab) to `1` or any other number, your acquisition will stop when that number of frames is reached.

– If you set **Frames Per Trigger** to `Infinite`, click the **Stop Acquisition** button to stop the acquisition.

Note that you can also click **Stop Acquisition** to abort an acquisition if number of frames was specified.

9 Optionally you can export data that was saved to memory.

You can export the data that has been acquired in memory to a MAT-file, the MATLAB Workspace, VideoWriter, or to the Image Tool, Image File, or Movie Player tools that are provided by the Image Processing Toolbox software using the **Export Data** button. For more information, see the “Exporting Data” section of the **Desktop Help** on the **Preview Window** in the **Desktop Help** pane in the tool.

10 Optionally you can save your configuration(s), using the **File > Save Configuration** or **File > Export Hardware Configuration** menus. For more information about these commands, see the “Image Acquisition Tool Menus” section of the **Help** on the **Hardware Browser** in the **Desktop Help** pane in the tool.

Image Acquisition Support Packages

Image Acquisition Support Packages for Hardware Adaptors

The existing support for all supported hardware, such as GigE Vision and Windows Video, is now available via Hardware Support Packages. This is the same functionality for acquiring images using all supported cameras and frame grabbers that has always been part of the Image Acquisition Toolbox.

With previous versions of the Image Acquisition Toolbox, the files for all of the adaptors were included in your installation. Starting with version R2014a, each adaptor is available separately in support packages via MATLAB Add-Ons. All of the support packages contain the necessary MATLAB files to use the toolbox with your adaptor. Some also contain third-party files, such as drivers or camera set-up utilities. Offering the adaptor files via Add-Ons allows us to provide the most up to date versions of files.

To install a support package, on the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. In the Add-On Explorer, scroll to the **Hardware Support Packages** section, and click **Load more** to find your support package. You can refine the list by selecting **Imaging/Cameras** in the **Filter by Hardware Type** section on the left side of the Add-On Explorer. Select your adaptor, for example Image Acquisition Toolbox Support Package for GigE Vision Hardware or Image Acquisition Toolbox Support Package for OS Generic Video Interface, from the list.

Note For any cameras that use the Windows Video (`winvideo`), Macintosh Video (`macvideo`), or Linux Video (`linuxvideo`) adaptors, use the support package called Image Acquisition Toolbox Support Package for OS Generic Video Interface. The correct files will be installed, depending on your operating system.

The following table shows the support package name for each adaptor. In Add-On Explorer, select your adaptor using the name listed in the table.

Adaptor Name	Support Package Name in Add-Ons	Contents
Windows Video (`winvideo` adaptor on `videoinput` object)	Image Acquisition Toolbox Support Package for OS Generic Video Interface	MATLAB files to use Windows Video, Macintosh Video, or Linux Video hardware with the toolbox. The correct OS files will be installed, depending on your system.
Kinect for Windows (`kinect` adaptor on `videoinput` object)	Image Acquisition Toolbox Support Package for Kinect® For Windows Sensor	MATLAB files to use Kinect for Windows V1 and V2 hardware with the toolbox Third party files – Kinect for Windows Runtime
DALSA Sapera (`dalsasapera` adaptor on `videoinput` object)	Image Acquisition Toolbox Support Package for DALSA® Sapera Hardware	MATLAB files to use DALSA Sapera hardware with the toolbox

Adaptor Name	Support Package Name in Add-Ons	Contents
GigE Vision (`gige` adaptor on `videoinput` object and `gigecam` object)	Image Acquisition Toolbox Support Package for GigE Vision Hardware	MATLAB files to use GigE Vision hardware with the toolbox
Matrox (`matrox` adaptor on `videoinput` object and `matroxcam` object)	Image Acquisition Toolbox Support Package for Matrox® Hardware	MATLAB files to use Matrox hardware with the toolbox
DCAM (`dcam` adaptor on `videoinput` object)	Image Acquisition Toolbox Support Package for DCAM Hardware	MATLAB files to use DCAM hardware with the toolbox Third party files – CMU DCAM on Windows driver file
GenICam GenTL (`gentl` adaptor on `videoinput` object)	Image Acquisition Toolbox Support Package for GenICam Interface	MATLAB files to use GenTL hardware with the toolbox
Point Grey (`pointgrey` adaptor on `videoinput` object)	Image Acquisition Toolbox Support Package for Point Grey Hardware	MATLAB files to use Point Grey hardware with the toolbox Third party files – Point Grey FlyCapture
Linux Video (`linuxvideo` adaptor on `videoinput` object)	Image Acquisition Toolbox Support Package for OS Generic Video Interface	MATLAB files to use Windows Video, Macintosh Video, or Linux Video hardware with the toolbox. The correct OS files will be installed, depending on your system.
Macintosh Video (`macvideo` adaptor on `videoinput` object)	Image Acquisition Toolbox Support Package for OS Generic Video Interface	MATLAB files to use Windows Video, Macintosh Video, or Linux Video hardware with the toolbox. The correct OS files will be installed, depending on your system.
National Instruments (`ni` adaptor on `videoinput` object)	Image Acquisition Toolbox Support Package for National Instruments™ Frame Grabbers	MATLAB files to use NI hardware with the toolbox Third party files – NI-IMAQ files

To use the cameras or frame grabbers you have been using with the toolbox, you must install the support package for the adaptor that your camera uses. If you use multiple adaptors, you need to install the support package for each one you use. For example, if you have a webcam on a Windows system and a Matrox camera, you would need to install the Image Acquisition Toolbox Support Package for OS Generic Video Interface for the `winvideo` adaptor for the webcam and the Image Acquisition Toolbox Support Package for Matrox Hardware for the `matrox` adaptor.

Go to MATLAB Add-Ons and use the adaptor name in the table to install the correct package(s) that you need. To install more than one package, select the support packages in the Add-On Explorer one at a time.

“Installing the Support Packages for Image Acquisition Toolbox Adaptors” on page 4-5 describes how to install the Image Acquisition Toolbox support packages.

Installing the Support Packages for Image Acquisition Toolbox Adaptors

With previous versions of the Image Acquisition Toolbox, the files for all of the adaptors were included in your installation. Starting with version R2014a, each adaptor is available separately in support packages via MATLAB Add-Ons. All of the support packages contain the necessary MATLAB files to use the toolbox with your adaptor. Some also contain third-party files, such as drivers or camera set-up utilities.

To use the cameras or frame grabbers you have been using with the toolbox, you must install the support package for the adaptor that your camera uses. If you use multiple adaptors, you need to install the support package for each one you use. For example, if you have a Webcam on a Windows system and a Matrox camera, you would need to install the Image Acquisition Toolbox Support Package for OS Generic Video Interface for the `winvideo` adaptor for the Webcam and the Image Acquisition Toolbox Support Package for Matrox Hardware for the `matrox` adaptor.

To use the Image Acquisition Toolbox for acquisition from any generic video interface, you need to install the Image Acquisition Toolbox Support Package for OS Generic Video Interface. This includes any cameras that use the Windows Video (`winvideo`), Macintosh Video (`macvideo`), or Linux Video (`linuxvideo`) adaptors. The correct files will be installed, depending on your operating system.

All video interface adaptors are available through the Hardware Support Packages. Using this installation process, you download and install the following file(s) on your host computer:

- Image Acquisition Toolbox adaptor files for your selected adaptor
- Third-party files if your support package includes them, depending on the adaptor (see the table in "Image Acquisition Support Packages for Hardware Adaptors" on page 4-2)

Install a Support Package

To install a support package:

1. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.
2. In the Add-On Explorer, scroll to the **Hardware Support Packages** section, and click **show all** to find your support package.
3. You can refine the list by selecting **Imaging/Cameras** in the **Refine by Hardware Type** section on the left side of the Explorer.
4. Select the support package for your adaptor. The table in "Image Acquisition Support Packages for Hardware Adaptors" on page 4-2 shows the names of the support packages for each adaptor type.

Uninstall or Update a Support Package

You can also use Add-Ons to uninstall or update support packages.

To uninstall support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Manage Add-Ons**.

To update existing support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

For more information about using Add-On Explorer, see “Get and Manage Add-Ons”.

Install the MATLAB Support Package for USB Webcams

You can use 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 USB Webcams support package.

Webcam support is available through MATLAB Add-Ons. Using this installation process, you download and install the following files on your host computer:

- MATLAB files for Webcam support
- An example that shows how to acquire images using a Webcam
- The USB Webcams support package documentation

To install the support package:

1. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.
2. In the Add-On Explorer, scroll to the **Hardware Support Packages** section, and click **show all** to find your support package.
3. Refine the list by selecting **Imaging/Cameras** in the **Refine by Hardware Type** section on the left side of the Explorer.
4. In the **Imaging/Cameras** list, select the MATLAB Support Package for USB Webcams.

Connecting to Hardware

To connect to an image acquisition device from within MATLAB, you must create a video input object. This object represents the connection between MATLAB and the device. You can use object properties to control various aspects of the acquisition. Before you can create the object, you need several pieces of information about the device that you want to connect to.

Getting Hardware Information

In this section...

Getting Hardware Information

To connect to an image acquisition device from within MATLAB, you must create a video input object. This object represents the connection between MATLAB and the device. You can use object properties to control various aspects of the acquisition. Before you can create the object, you need several pieces of information about the device that you want to connect to.

To access an image acquisition device, the toolbox needs several pieces of information:

- The name of the adaptor the toolbox uses to connect to the image acquisition device
- The device ID of the device you want to access
- The video format of the video stream or, optionally, a device configuration file (camera file)

You use the `imaqhwinfo` function to retrieve this information, as described in the following subsections.

Note When using `imaqhwinfo` to get information about a device, especially devices that use a Video for Windows (VFW) driver, you might encounter dialog boxes reporting an assertion error. Make sure that the software drivers are installed correctly and that the acquisition device is connected to the computer.

Determining the Device Adaptor Name

An adaptor is the software the toolbox uses to communicate with an image acquisition device via its device driver. The toolbox includes adaptors for some vendors of image acquisition equipment and for particular classes of image acquisition devices. For the latest information about supported hardware, visit the Image Acquisition Toolbox product page at the MathWorks Web site (`www.mathworks.com/products/image-acquisition`).

To determine which adaptors are available on your system, call the `imaqhwinfo` function. The `imaqhwinfo` function returns information about the toolbox software and lists the adaptors available on the system in the `InstalledAdaptors` field. In this example, there are two adaptors available on the system.

```
imaqhwinfo
ans = 

    InstalledAdaptors: {'matrox'  'winvideo'}
        MATLABVersion: '7.4 (R2007a)'
          ToolboxName: 'Image Acquisition Toolbox'
       ToolboxVersion: '2.1 (R2007a)'
```

Note While every adaptor supported by the Image Acquisition Toolbox software is installed with the toolbox, `imaqhwinfo` lists only adaptors in the `InstalledAdaptors` field that are loadable. That is, the device drivers required by the vendor are installed on the system. Note, however, that inclusion in the `InstalledAdaptors` field does not necessarily mean that an adaptor is connected to a device.

Determining the Device ID

The adaptor assigns a unique number to each device with which it can communicate. The adaptor assigns the first device it detects the device ID 1, the second it detects the device ID 2, and so on.

To find the device ID of a particular image acquisition device, call the `imaqhwinfo` function, specifying the name of the adaptor as the only argument. When called with this syntax, `imaqhwinfo` returns a structure containing information about all the devices available through the specified adaptor.

In this example, the `imaqhwinfo` function returns information about all the devices available through the Matrox adaptor.

```
info = imaqhwinfo('matrox');
info =

       AdaptorDllName: [1x73 char]
    AdaptorDllVersion: '2.1 (R2007a)'
          AdaptorName: 'matrox'
            DeviceIDs: {[1]}
           DeviceInfo: [1x1 struct]
```

The fields in the structure returned by `imaqhwinfo` provide the following information.

Field	Description
`AdaptorDllName`	Character vector that identifies the name of the adaptor dynamic link library (DLL)
`AdaptorDllVersion`	Information about the version of the adaptor DLL
`AdaptorName`	Name of the adaptor
`DeviceIDs`	Cell array containing the device IDs of all the devices accessible through this adaptor
`DeviceInfo`	Array of device information structures. See “Getting More Information About a Particular Device” on page 5-3 for more information.

Getting More Information About a Particular Device

If an adaptor provides access to multiple devices, you might need to find out more information about the devices before you can select a device ID. The `DeviceInfo` field is an array of device information structures. Each device information structure contains detailed information about a particular device available through the adaptor.

To view the information for a particular device, you can use the device ID as a reference into the `DeviceInfo` structure array. Call `imaqhwinfo` again, this time specifying a device ID as an argument.

```
dev_info = imaqhwinfo('matrox',1)
dev_info = 

          DefaultFormat: 'M_RS170'
    DeviceFileSupported: 1
             DeviceName: 'Orion'
               DeviceID: 1
  VideoInputConstructor: 'videoinput('matrox', 1)'
 VideoDeviceConstructor: 'imaq.VideoDevice('matrox', 1)'
       SupportedFormats: {1x10 cell}
```

The fields in the device information structure provide the following information about a device.

Field	Description
`DefaultFormat`	Character vector that identifies the video format used by the device if none is specified at object creation time
`DeviceFileSupported`	If set to 1, the device supports device configuration files; otherwise 0. See “Using Device Configuration Files (Camera Files)” on page 5-9 for more information.
`DeviceName`	Descriptive character vector, assigned by the adaptor, that identifies the device
`DeviceID`	ID assigned to the device by the adaptor
`VideoInputConstructor`	Default syntax you can use to create a video input object to represent this device. See “Creating Image Acquisition Objects” on page 5-6 for more information.
`VideoDeviceConstructor`	Default syntax you can use to create a VideoDevice System object to represent this device.
`SupportedFormats`	Cell array of character vectors that identify the video formats supported by the device. See “Determining Supported Video Formats” on page 5-4 for more information.

Determining Supported Video Formats

The video format specifies the characteristics of the images in the video stream, such as the image resolution (width and height), the industry standard used, and the size of the data type used to store pixel information.

Image acquisition devices typically support multiple video formats. You can specify the video format when you create the video input object to represent the connection to the device. See “Creating Image Acquisition Objects” on page 5-6 for more information.

Note Specifying the video format is optional; the toolbox uses one of the supported formats as the default.

To determine which video formats an image acquisition device supports, look in the `SupportedFormats` field of the `DeviceInfo` structure returned by the `imaqhwinfo` function. To view the information for a particular device, call `imaqhwinfo`, specifying the device ID as an argument.

```
dev_info = imaqhwinfo('matrox',1)
dev_info =

          DefaultFormat: 'M_RS170'
    DeviceFileSupported: 1
             DeviceName: 'Orion'
               DeviceID: 1
  VideoInputConstructor: 'videoinput('matrox', 1)'
 VideoDeviceConstructor: 'imaq.VideoDevice('matrox', 1)'
       SupportedFormats: {1x10 cell}
```

The `DefaultFormat` field lists the default format selected by the toolbox. The `SupportedFormats` field is a cell array containing character vectors that identify all the supported video formats. The toolbox assigns names to the formats based on vendor-specific terminology. If you want to specify a video format when you create an image acquisition object, you must use one of the character vectors in this cell array. See "Creating Image Acquisition Objects" on page 5-6 for more information.

```
celldisp(dev_info.SupportedFormats)

ans{1} =

M_RS170

ans{2} =

M_RS170_VIA_RGB

ans{3} =

M_CCIR

ans{4} =

M_CCIR_VIA_RGB

ans{5} =

M_NTSC

ans{6} =

M_NTSC_RGB

ans{7} =

M_NTSC_YC

ans{8} =

M_PAL

ans{9} =

M_PAL_RGB

ans{10} =

M_PAL_YC
```

Creating Image Acquisition Objects

In this section...

Types of Objects

After you get information about your image acquisition hardware, described in “Getting Hardware Information” on page 5-2, you can establish a connection to the device by creating an image acquisition object. The toolbox uses two types of image acquisition objects:

- Video input object
- Video source object

Video Input Objects

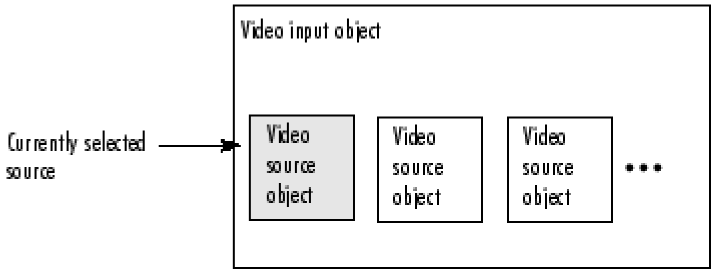
A video input object represents the connection between MATLAB and a video acquisition device at a high level. You must create the video input object using the `videoinput` function. See “Creating a Video Input Object” on page 5-7 for more information.

Video Source Objects

When you create a video input object, the toolbox automatically creates one or more video source objects associated with the video input object. Each video source object represents a collection of one or more physical data sources that are treated as a single entity. The number of video source objects the toolbox creates depends on the device and the video format you specify.

At any one time, only one of the video source objects, called the *selected* source, can be active. This is the source used for acquisition. The toolbox selects one of the video source objects by default, but you can change this selection. See “Specifying the Selected Video Source Object” on page 5-10 for more information.

The following figure illustrates how a video input object acts as a container for one or more video source objects.



Relationship of Video Input Objects and Video Source Objects

For example, a Matrox frame grabber device can support eight physical connections, which Matrox calls channels. These channels can be configured in various ways, depending upon the video format. If you specify a monochrome video format, such as RS170, the toolbox creates eight video source objects, one object for each of the eight channels on the device. If you specify a color video format, such as NTSC RGB, the Matrox device uses three physical channels to represent one RGB connection, where each physical connection provides the red data, green data, and blue data separately. With this format, the toolbox only creates two video source objects for the same device.

Creating a Video Input Object

To create a video input object, call the `videoinput` function specifying the adaptor name, device ID, and video format. You retrieved this information using the `imaqhwinfo` function (described in “Getting Hardware Information” on page 5-2). The only required argument is the adaptor name. The toolbox can use default values for the device ID and video format.

This example creates a video input object to represent the connection to a Matrox image acquisition device. The `imaqhwinfo` function includes the default `videoinput` syntax in the `VideoInputConstructor` field of the device information structure.

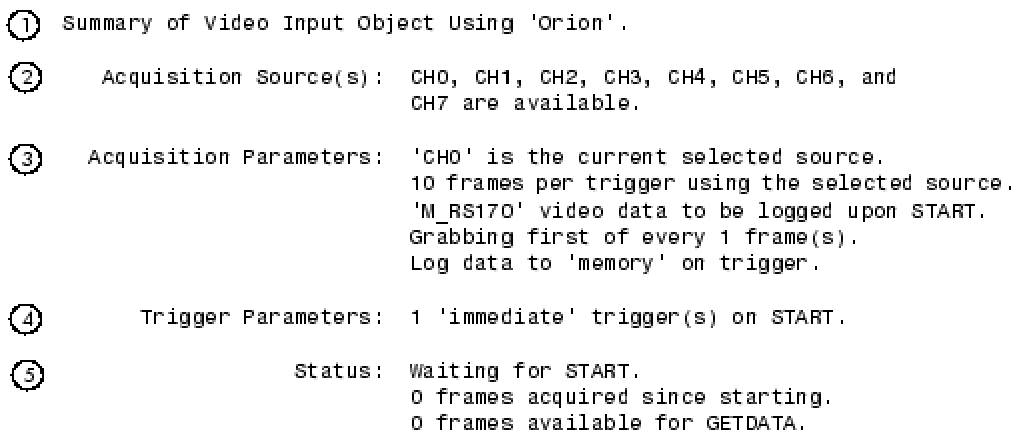
```
vid = videoinput('matrox');
```

This syntax uses the default video format listed in the `DefaultFormat` field of the data returned by `imaqhwinfo`. You can optionally specify the video format. See “Specifying the Video Format” on page 5-8 for more information.

Viewing a Summary of a Video Input Object

To view a summary of the characteristics of the video input object you created, enter the variable name you assigned to the object at the command prompt. For example, this is the summary for the object `vid`.

```
vid
```



```
① Summary of Video Input Object Using 'Orion'.

②    Acquisition Source(s):  CH0, CH1, CH2, CH3, CH4, CH5, CH6, and
                             CH7 are available.

③   Acquisition Parameters:  'CH0' is the current selected source.
                             10 frames per trigger using the selected source.
                             'M_RS170' video data to be logged upon START.
                             Grabbing first of every 1 frame(s).
                             Log data to 'memory' on trigger.

④       Trigger Parameters:  1 'immediate' trigger(s) on START.

⑤                   Status:  Waiting for START.
                             0 frames acquired since starting.
                             0 frames available for GETDATA.
```

The items in this list correspond to the numbered elements in the object summary:

1. The title of the summary includes the name of the image acquisition device this object represents. In the example, this is a Matrox Orion frame grabber.
2. The Acquisition Source section lists the name of all the video source objects associated with this video input object. For many objects, this list might only contain one video source object. In the example, the Matrox device supports eight physical input channels and, with the default video format, the toolbox creates a video source object for each connection. For an example showing the video source objects created with another video format, see "Specifying the Video Format" on page 5-8.
3. The Acquisition Parameters section lists the values of key video input object properties. These properties control various aspects of the acquisition, such as the number of frames to acquire and the location where acquired frames are stored. For information about these properties, see "Acquiring Image Data" on page 6-2.
4. The Trigger Parameters section lists the trigger type configured for the object and the number of times the trigger is to be executed. Trigger execution initiates data logging, and the toolbox supports several types of triggers. The example object is configured by default with an immediate trigger. For more information about configuring triggers, see "Specifying the Trigger Type" on page 6-8.
5. The Status section lists the current state of the object. A video input object can be in one of several states:

   - Running or not running (stopped)
   - Logging or not logging
   - Previewing or not previewing

   In the example, the object describes its state as `Waiting for START`. This indicates it is not running. For more information about the running state, see "Starting and Stopping a Video Input Object" on page 5-25. This section also reports how many frames of data have been acquired and how many frames are available in the buffer where the toolbox stores acquired frames. For more information about these parameters, see "Controlling Logging Parameters" on page 6-19.

Specifying the Video Format

You can optionally specify the format of the video stream when you create a video input object as a third argument to the `videoinput` function. This argument can take two forms:

- A character vector specifying a video format
- A name of a device configuration file, also known as a camera file

The following sections describe these options. If you do not specify a video format, the `videoinput` function uses one of the video formats supported by the device. For Matrox and Data Translation® devices, it chooses the RS170 video format. For Windows devices, it uses the first RGB format in the list of supported formats or, if no RGB formats are supported, the device's default format.

Using a Video Format Character Vector

To specify a video format as a character vector, use the `imaqhwinfo` function to determine the list of supported formats. The `imaqhwinfo` function returns this information in the `SupportedFormats` field of the device information structure. See “Determining Supported Video Formats” on page 5-4 for more information.

In this example, each of the character vectors is a video format supported by a Matrox device.

```
info = imaqhwinfo('matrox');

info.DeviceInfo.SupportedFormats

ans =
  Columns 1 through 4

    'M_RS170'    'M_RS170_VIA_RGB'    'M_CCIR'    'M_CCIR_VIA_RGB'

  Columns 5 through 8

'M_NTSC'    'M_NTSC_RGB'      'M_NTSC_YC'    'M_PAL'

  Columns 9 through 10

'M_PAL_RGB'    'M_PAL_YC'
```

For Matrox devices, the toolbox uses the RS170 format as the default. (To find out which is the default video format, look in the `DefaultFormat` field of the device information structure returned by the `imaqhwinfo` function.)

Note For Matrox devices, the `M_NTSC_RGB` format represents a component video format.

This example creates a video input object, specifying a color video format.

```
vid2 = videoinput('matrox', 1,'M_NTSC_RGB');
```

Using Device Configuration Files (Camera Files)

For some devices, you can use a device configuration file, also known as a camera file, to specify the video format as well as other configuration settings. Image acquisition device vendors supply these device configuration files.

Note The toolbox ignores hardware trigger configurations included in a device configuration file. To configure a hardware trigger, you must use the toolbox `triggerconfig` function. See “Using a Hardware Trigger” on page 6-12 for more information.

For example, with Matrox frame grabbers, you can download digitizer configuration format (DCF) files, in their terminology. These files configure their devices to support particular cameras.

Some image acquisition device vendors provide utility programs you can use to create a device configuration file or edit an existing one. See your hardware vendor's documentation for more information.

To determine if your image acquisition device supports device configuration files, check the value of the `DeviceFileSupported` field of the device information structure returned by `imaqhwinfo`. See “Getting More Information About a Particular Device” on page 5-3 for more information.

When you use a device configuration file, the value of the `VideoFormat` property of the video input object is the name of the file, not a video format character vector.

This example creates a video input object specifying a Matrox device configuration file as an argument.

```
vid = videoinput('matrox',1,'pulnix.dcf')

Summary of Video Input Object Using 'Orion'.

   Acquisition Source(s): CH0 and CH1 are available.

  Acquisition Parameters: 'CH0' is the current selected source.
                          10 frames per trigger using the selected source.
                          'C:\pulnix.dcf' video data to be logged upon START.
                          Grabbing first of every 1 frame(s).
                          Log data to 'memory' on trigger.

      Trigger Parameters: 1 'immediate' trigger(s) on START.

                  Status: Waiting for START.
                          0 frames acquired since starting.
                          0 frames available for GETDATA.
```

Specifying the Selected Video Source Object

When you create a video input object, the toolbox creates one or more video source objects associated with the video input object. The number of video source objects created depends on the device and the video format. The `Source` property of the video input object lists these video source objects.

To illustrate, this example lists the video source objects associated with the video input object `vid`.

```
vid.Source
     Display Summary for Video Source Object Array:

      Index:   SourceName:   Selected:
      1        'CH0'         'on'
      2        'CH1'         'off'
      3        'CH2'         'off'
      4        'CH3'         'off'
      5        'CH4'         'off'
      6        'CH5'         'off'
      7        'CH6'         'off'
      8        'CH7'         'off'
```

By default, the video input object makes the first video source object in the array the selected source. To use another video source, change the value of the `SelectedSourceName` property.

This example changes the currently selected video source object from `CH0` to `CH1` by setting the value of the `SelectedSourceName` property.

```
vid.SelectedSourceName = 'CH1';
```

Note The `getselectedsource` function returns the video source object that is currently selected at the time the function is called. If you change the value of the `SelectedSourceName` property, you must call the `getselectedsource` function again to retrieve the new selected video source object.

Getting Information About a Video Input Object

After creating a video input object, you can get information about the device it represents using the `imaqhwinfo` function. When called with a video input object as an argument, `imaqhwinfo` returns a structure containing information about the object such as the name of the adaptor, name of the device, video resolution, and details of the vendor's device driver and version.

```
out = imaqhwinfo(vid)
out = 

                AdaptorName: 'winvideo'
                 DeviceName: 'IBM PC Camera'
                  MaxHeight: 96
                   MaxWidth: 128
             NativeDataType: 'uint8'
               TotalSources: 1
    VendorDriverDescription: 'Windows WDM Compatible Driver'
        VendorDriverVersion: 'DirectX 9.0'
```

Configuring Image Acquisition Object Properties

In this section...

About Image Acquisition Object Properties

The video input object and the video source object both support properties that enable you to control characteristics of the video image and how it is acquired.

The video input object properties control aspects of an acquisition that are common to all image acquisition devices. For example, you can use the `FramesPerTrigger` property to specify the amount of data you want to acquire.

The video source object properties control aspects of the acquisition associated with a particular source. The set of properties supported by a video source object varies with each device. For example, some image acquisition devices support properties that enable you to control the quality of the image being produced, such as `Brightness`, `Hue`, and `Saturation`.

With either type of object, you can use the same toolbox functions to

- View a list of all the properties supported by the object, with their current values
- View the value of a particular property
- Get information about a property
- Set the value of a property

Note Three video input object trigger properties require the use of a special configuration function. For more information, see "Setting Trigger Properties" on page 5-16.

Viewing the Values of Object Properties

To view all the properties of an image acquisition object, with their current values, use the `get` function. You can also use the `inspect` function to view a list of object properties in the Property Inspector window, where you can also edit their values.

This example uses the `get` function to display a list of all the properties of the video input object `vid`. "Viewing the Properties of a Video Source Object" on page 5-13 describes how to do this for video source objects.

If you do not specify a return value, the `get` function displays the object properties in four categories: General Settings, Callback Function Settings, Trigger Settings, and Acquisition Sources.

```
get(vid)
  General Settings:
    DeviceID = 1
    DiskLogger = []
    DiskLoggerFrameCount = 0
    EventLog = [1x0 struct]
    FrameGrabInterval = 1
    FramesAcquired = 0
    FramesAvailable = 0
    FramesPerTrigger = 10
    Logging = off
    LoggingMode = memory
    Name = M_RS170-matrox-1
    NumberOfBands = 1
    Previewing = off
    ReturnedColorSpace = grayscale
    ROIPosition = [0 0 640 480]
    Running = off
    Tag =
    Timeout = 10
    Type = videoinput
    UserData = []
    VideoFormat = M_RS170
    VideoResolution = [640 480]

  Callback Function Settings:
    ErrorFcn = @imaqcallback
    FramesAcquiredFcn = []
    FramesAcquiredFcnCount = 0
    StartFcn = []
    StopFcn = []
    TimerFcn = []
    TimerPeriod = 1
    TriggerFcn = []

  Trigger Settings:
    InitialTriggerTime = [0 0 0 0 0 0]
    TriggerCondition = none
    TriggerFrameDelay = 0
    TriggerRepeat = 0
    TriggersExecuted = 0
    TriggerSource = none
    TriggerType = immediate

  Acquisition Sources:
    SelectedSourceName = CH0
    Source = [1x8 videosource]
```

Viewing the Properties of a Video Source Object

To view the properties supported by the video source object (or objects) associated with a video input object, use the `getselectedsource` function to retrieve the currently selected video source object. This example lists the properties supported by the video source object associated with the video input object `vid`. Note the device-specific properties that are included.

Note The video source object for your device might not include device-specific properties. For example, devices accessed with the `'winvideo'` adaptor, such as webcams, that use a Video for

Windows (VFW) driver, may not provide a way for the toolbox to programmatically query for device properties. Use the configuration tools provided by the manufacturer to configure these devices.

```
get(getselectedsource(vid))
  General Settings:
    Parent = [1x1 videoinput]
    Selected = on
    SourceName = CH0
    Tag = 
    Type = videosource

  Device Specific Properties:
    InputFilter = lowpass
    UserOutputBit3 = off
    UserOutputBit4 = off
    XScaleFactor = 1
    YScaleFactor = 1
```

Viewing the Value of a Particular Property

To view the value of a particular property of an image acquisition object, access the value of the property as you would a field in a MATLAB structure.

This example illustrates how to access a property by referencing the object as if it were a MATLAB structure using dot notation.

```
vid.Previewing

ans =

off
```

Getting Information About Object Properties

To get information about a particular property, see “Image Acquisition Toolbox Properties” on page 5-31. You can also get information about a particular property at the command line by using the `propinfo` or `imaqhelp` functions.

The `propinfo` function returns a structure that contains information about the property such as its data type, default value, and a list of all possible values, if the property supports such a list. This example uses `propinfo` to get information about the `LoggingMode` property.

```
propinfo(vid,'LoggingMode')

ans = 

               Type: 'character vector'
         Constraint: 'enum'
    ConstraintValue: {'memory'  'disk'  'disk&memory'}
       DefaultValue: 'memory'
           ReadOnly: 'whileRunning'
     DeviceSpecific: 0
```

The `imaqhelp` function returns reference information about the property with a complete description. This example uses `imaqhelp` to get information about the `LoggingMode` property.

```
imaqhelp(vid,'LoggingMode')
```

Setting the Value of an Object Property

To set the value of a particular property of an image acquisition object, you assign the value to the property as you would a field in a MATLAB structure, using dot notation.

Note Because some properties are read-only, only a subset of all video input and video source properties can be set.

This example sets the value of a property by assigning the value to the object as if it were a MATLAB structure.

```
vid.LoggingMode = 'disk';

% Verify the property setting.
vid.LoggingMode

ans =

disk
```

Viewing a List of All Settable Object Properties

To view a list of all the properties of a video input object or video source object that can be set, use the `set` function.

```
set(vid)
```

```
 General Settings:
  DiskLogger
  FrameGrabInterval
  FramesPerTrigger
  LoggingMode: [ {memory} | disk | disk&memory ]
  Name
  ReturnedColorSpace: [ {rgb} | grayscale | YCbCr ]
  ROIPosition
  Tag
  Timeout
  UserData

Callback Function Settings:
  ErrorFcn: string -or- function handle -or- cell array
  FramesAcquiredFcn: string -or- function handle -or- cell array
  FramesAcquiredFcnCount
  StartFcn: string -or- function handle -or- cell array
  StopFcn: string -or- function handle -or- cell array
  TimerFcn: string -or- function handle -or- cell array
  TimerPeriod
  TriggerFcn: string -or- function handle -or- cell array

Trigger Settings:
  TriggerFrameDelay
  TriggerRepeat

Acquisition Sources:
  SelectedSourceName: [ {CH0} | CH1 | CH2 | CH3 | CH4 ]
```

Setting Trigger Properties

The values of certain trigger properties, `TriggerType`, `TriggerCondition`, and `TriggerSource`, are interrelated. For example, some `TriggerCondition` values are only valid with specific values of the `TriggerType` property.

To ensure that you specify only valid combinations for the values of these properties, you must use two functions:

- The `triggerinfo` function returns all the valid combinations of values for the specified video input object.
- The `triggerconfig` function sets the values of these properties.

For more information, see “Specifying Trigger Type, Source, and Condition” on page 6-6.

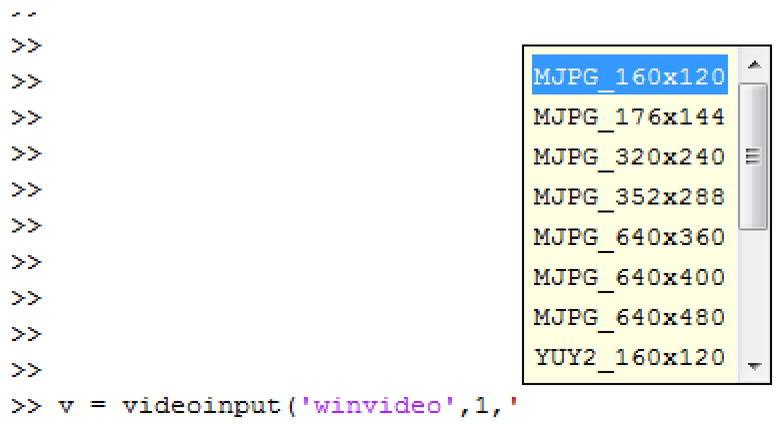
Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Using Tab Completion for Functions

To get a list of options you can use on the function, press the **Tab** button after entering a function on the command line. The list expands and you can scroll to choose a property or value. For example, when you create the `videoinput` object, you can get a list of supported formats:

```
v = videoinput('winvideo',1,
```

When you press **Tab** after the Device ID (1 in this example), the list of formats displays, as shown here.



```
>>
>>
>>
>>
>>
>>
>>
>>
>>
>>
>> v = videoinput('winvideo',1,'
```

If you pressed **Tab** just after the first parentheses, a list of available adaptors would appear. The format for the object constructor function is:

```
v = videoinput('adaptorName',deviceID,'VideoFormat')
```

When you press **Tab** where a field should appear, you get the list of options for that field.

You can also get the values for property-value pairs, which can be used on the `videoinput` function after the first three required fields. For example,

```
v = videoinput('winvideo',1,'RGB1024x768','LoggingMode',
```

Press **Tab** after typing `LoggingMode` to get the possible values for that property, which are `'memory'`, `'disk'`, and `'disk&memory'`.

Many of the other toolbox functions also have tab completion. For example, when using the `getdata` function you can specify the data type using tab completion.

```
data = getdata(obj,10,
```

Press **Tab** after typing the number of frames (10 in this example) to get the possible values for the data types, such as `'uint8'` or `'double'`, since data type is the next input `getdata` takes after number of frames.

Use Advanced Property Support in the GigE Vision and GenICam GenTL Interfaces

In this section...

Advanced Property Support

The Image Acquisition Toolbox has added GenICam property enhancements for the GigE Vision (`gige`) and GenICam GenTL (`gentl`) adaptors used with the `videoinput` object in R2016a. These features were already included in the `gigecam` object.

- Ability to change properties while the acquisition is running
- Dynamic accessibility and readability
- Dynamic constraints
- Grouped selector properties

Change Properties While the Acquisition Is Running

This ability is useful for properties that you want to change dynamically, such as exposure time. For example, you can now do this:

```
% Create the videoinput object using the GigE adaptor
vid = videoinput('gige')

% Get the video source
src = getselectedsource(vid);

% Set the frames per trigger on the source
vid.FramesPerTrigger = Inf;

% Start acquiring frames
start(vid)

% Change the exposure time during the acquisition
src.ExposureTime = 4;
```

Previously, changing the exposure time after starting the acquisition resulted in an error.

Note This workflow is not supported in the **Image Acquisition Explorer**. While the acquisition is running, you can not change a property on the **Device Properties** tab.

Dynamic Accessibility and Readability

Device-specific properties, or camera GenICam properties, are now dynamically accessible. In previous releases, camera GenICam properties that were not accessible were hidden. If you display the device-specific properties using the `disp`, `get` or `propinfo` functions, properties that previously did not show up now show up with labels.

The `propinfo` function includes a new field called `Accessible`, which is a read-only boolean property. A `disp` on a property that has `Accessible` set to `0` results in “Currently not accessible.” To enable accessibility, set `Accessible` to `1`. For example, if you have the `ReverseY` property set to `Accessible`, the following:

```
propinfo(src,'ReverseY')
```

would result in a disp showing:

```
Accessible: 1
```

The same is true for the `ReadOnly` property. Readability is now dynamic and the `propinfo` function shows a `ReadOnly` property as either `'notCurrently'`, if it is writable, or `'currently'`, if it is read-only. The example in the Dynamic Constraints section demonstrates the dynamic use of this property.

You can view the source properties to see if any properties are currently not accessible. In this example, for the part of the disp shown below, `AcquisitionFrameCount` and `BalanceRatioRaw` are currently not accessible.

```
>> src = vid.Source

src =

   Display Summary for Video Source Object:

      General Settings:
        Parent = [1x1 videoinput]
        Selected = on
        SourceName = input1
        Tag = [0x0 character vector]
        Type = videosource

      Device Specific Properties:
        AcquisitionFrameCount = (Currently not accessible)
        AcquisitionFrameRate = 4.5
        AcquisitionFrameRateAuto = Off
        AcquisitionFrameRateEnabled = True
        BalanceRatioRaw = (Currently not accessible)
        BinningHorizontal = 1
        BinningVertical = 1
        BlackLevel = 1.001
        ...
```

Dynamic Constraints

If you change a property that results in a change of possible values, or constraint change, for another property, the other property’s constraint values are updated dynamically. Consider a camera that has an automatic sharpness setting that you can set to `Continuous` to automatically adjust the sharpness or set to `Off`. The automatic sharpness property then affects the related `Sharpness` property. In this example, when `SharpnessAuto` is set to `Continuous`, a disp of the `Sharpness` property shows the constrained values and that it is not able to be set.

```
>> propinfo(src, 'SharpnessAuto')

ans =

               Type: 'character vector'
         Constraint: 'enum'
    ConstraintValue: {'Continuous'  'Off'}
       DefaultValue: 'Continuous'
           ReadOnly: 'notCurrently'
     DeviceSpecific: 1
         Accessible: 1

>> propinfo(src, 'Sharpness')

ans =

               Type: 'integer'
         Constraint: 'bounded'
    ConstraintValue: [1532 1532]
       DefaultValue: 1532
           ReadOnly: 'currently'
     DeviceSpecific: 1
         Accessible: 1
```

If you then set the `SharpnessAuto` property to `Off`, a second disp of the `Sharpness` property shows that the constrained values have dynamically updated, and that it is now able to be set (no longer read-only).

```
>> src.SharpnessAuto = 'Off'
>> propinfo(src, 'Sharpness')

ans =

               Type: 'integer'
         Constraint: 'bounded'
    ConstraintValue: [0 4095]
       DefaultValue: 1532
           ReadOnly: 'notCurrently'
     DeviceSpecific: 1
         Accessible: 1
```

Grouped Selector Properties

In both the **Image Acquisition Explorer** and the command line, selector properties are now grouped. In the tool, you can see the groupings in the **Device Properties** tab. In the property display on the command line, the related properties are grouped – the selector property is listed, with its possible values appearing below it.

For example, in previous versions of the toolbox, for a `GainSelector` with possible values of `Red`, `Blue`, and `Green` and a `Gain` property, the gain properties displayed as follows:

```
>> vid = videoinput('gige')
>> src = getselectedsource(vid)
...
...
RedGain = 0.4
BlueGain = 0.2
GreenGain = 0.1
...
```

They now display as separate values on one selector property instead:

```
>> vid = videoinput('gige')
>> src = getselectedsource(vid)
...
...
GainSelector = 'Red'
Gain = 0.2
...
```

Compatibility Considerations

The grouping of selector properties results in a compatibility consideration starting in R2016a because of the change in how selector properties are displayed, read, or written. There are now fewer properties since some are shown as a single selector property with separate values, whereas they used to be separate properties.

If you have any MATLAB code written prior to R2016a which references the previous, separate properties, you need to change the code to reflect them as values on the selector property. Setting and getting properties that belong to a selector using the previous composite-name style is no longer supported. For example, `RedGain` no longer works. Instead use `GainSelector` set to `Red`, as shown in the example.

To set a property value, first set the selector value, then set the property value:

```
src.GainSelector = 'Green';
src.Gain = 0.1;
```

Use Advanced Property Support with Point Grey Camera

Use advanced property support with Point Grey cameras to change properties while the acquisition is running. Additionally, when changing the value of a property, you also dynamically update the constraint values of other properties that depend on it.

Change Properties While the Acquisition Is Running

You can change the value of the video source property of a Point Grey camera while image acquisition is running. This ability is useful for device-specific properties that you want to change dynamically, such as brightness, exposure, or frame rate. In this example, start acquisition from the `videoinput` object and then set the `Exposure` property.

Create the `videoinput` object using the Point Grey adaptor and get the video source.

```
vid = videoinput(“pointgrey”);
src = vid.Source;
```

Set the number of frames per trigger on the source.

```
vid.FramesPerTrigger = Inf;
```

Start acquiring frames.

```
start(vid)
```

View the `Exposure` property information to determine whether the property can be changed while acquisition is running.

```
propinfo(src,"Exposure")


ans =

  struct with fields:

               Type: 'double'
         Constraint: 'bounded'
    ConstraintValue: [-7.5850 2.4136]
       DefaultValue: -0.0614
           ReadOnly: 'never'
     DeviceSpecific: 1
         Accessible: 1
```

Since `ReadOnly` is `'never'`, you can change this property during acquisition. The current value is `-0.0614` and the maximum and minimum constraints are `[-7.5850 2.4136]`.

Change the value of the `Exposure` property during the acquisition.

```
src.Exposure = 2;
```

Previously, changing the exposure after starting the acquisition resulted in an error.

Stop the image acquisition when you are done.

```
stop(vid)
```

Note This workflow is not supported in the **Image Acquisition Explorer**. While the acquisition is running, you can not change a property on the **Device Properties** tab.

Update Property Constraints Dynamically

If you change a property that results in a change of possible values, or constraint change, for another property, the constraint values of the other property are updated dynamically. Consider a Point Grey camera that has a region of interest that is already set to `[0 0 612 512]`. The values limit the `FrameRate` property to a specific minimum and maximum value, depending on the `ROIPosition` value. Changing the region of interest to a lower value increases the `FrameRate` property constraints. In this example, you set `ROIPosition` to `[0 0 320 240]`, and you call `propinfo` on the `FrameRate` property to show the updated property constraint values.

Create the `videoinput` object using the Point Grey adaptor and get the video source.

```
vid = videoinput(“pointgrey”);
src = vid.Source;
```

View the region of interest.

```
vid.ROIPosition
```

```
ans =

     0     0   612   512
```

View the `FrameRate` property information.

```
propinfo(src,"FrameRate")
```

```
ans =

  struct with fields:

               Type: 'double'
         Constraint: 'bounded'
    ConstraintValue: [1 29]
       DefaultValue: 2.5000
           ReadOnly: 'never'
     DeviceSpecific: 1
         Accessible: 1
```

The minimum and maximum values for this property are `[1 29]`.

Set the `ROIPosition` property to `[0 0 320 240]` and view the `FrameRate` property again to see the updated values.

```
vid.ROIPosition = [0 0 320 240];
propinfo(src,"FrameRate")
```

```
ans =

  struct with fields:
```

```
               Type: 'double'
         Constraint: 'bounded'
    ConstraintValue: [1 34]
       DefaultValue: 2.5000
           ReadOnly: 'never'
     DeviceSpecific: 1
         Accessible: 1
```

The minimum and maximum values are now `[1 34]` because the region of interest is lowered.

See Also

`videoinput`

More About

- "Use Advanced Property Support in the GigE Vision and GenICam GenTL Interfaces" on page 5-18

Starting and Stopping a Video Input Object

When you create a video input object, you establish a connection between MATLAB and an image acquisition device. However, before you can acquire data from the device, you must start the object, using the `start` function.

```
start(vid);
```

When you start an object, you reserve the device for your exclusive use and lock the configuration. Thus, certain properties become read only while running.

An image acquisition object stops running when any of the following conditions is met:

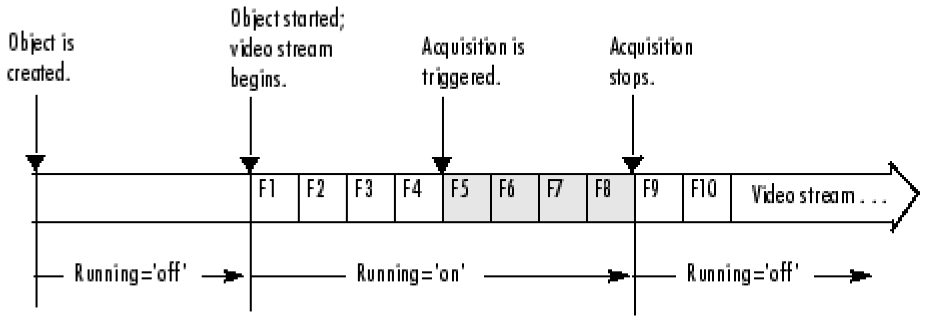
- The requested number of frames is acquired. This occurs when

  ```
  FramesAcquired = FramesPerTrigger * (TriggerRepeat + 1)
  ```

  where `FramesAcquired`, `FramesPerTrigger`, and `TriggerRepeat` are properties of the video input object. For information about these properties, see “Acquiring Image Data” on page 6-2.
- A run-time error occurs.
- The object's `Timeout` value is reached.
- You issue the `stop` function.

When an object is started, the toolbox sets the object's `Running` property to `'on'`. When an object is not running, the toolbox sets the object's `Running` property to `'off'`; this state is called stopped.

The following figure illustrates how an object moves from a running to a stopped state.



Transitions from Running to Stopped States

The following example illustrates starting and stopping an object:

1. **Create an image acquisition object** — This example creates a video input object for a webcam image acquisition device. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

   ```
   vid = videoinput('winvideo',1);
   ```
2. **Verify that the image is in a stopped state** — Use the `isrunning` function to determine the current state of the video input object.

   ```
   isrunning(vid)
   ```

```
ans =

     0
```

3 **Configure properties** To illustrate object states, set the video input object's `TriggerType` property to `'Manual'`. To set the value of certain trigger properties, including the `TriggerType` property, you must use the `triggerconfig` function. See “Setting the Values of Trigger Properties” on page 6-6 for more information.

```
triggerconfig(vid, 'Manual')
```

Configure an acquisition that takes several seconds so that you can see the video input in logging state.

```
vid.FramesPerTrigger = 100;
```

4 **Start the image acquisition object** — Call the `start` function to start the image acquisition object.

```
start(vid)
```

5 **Verify that the image is running but not logging** — Use the `isrunning` and `islogging` functions to determine the current state of the video input object. With manual triggers, the video input object is in running state after being started but does not start logging data until a trigger executes.

```
isrunning(vid)

ans =

     1

islogging(vid)

ans =

     0
```

6 **Execute the manual trigger** — Call the `trigger` function to execute the manual trigger.

```
trigger(vid)
```

While the acquisition is underway, check the logging state of the video input object.

```
islogging(vid)

ans =

     1
```

After it acquires the specified number of frames, the video input object stops running.

```
isrunning(vid)

ans =

     0
```

7 **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

```
delete(vid)
clear vid
```

Deleting Image Acquisition Objects

When you finish using your image acquisition objects, use the `delete` function to remove them from memory. After deleting them, clear the variables that reference the objects from the MATLAB workspace by using the `clear` function.

Note When you delete a video input object, all the video source objects associated with the video input object are also deleted.

To illustrate, this example creates several video input objects and then deletes them.

1. **Create several image acquisition objects** — This example creates several video input objects for a single webcam image acquisition device, specifying several different video formats. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

   ```
   vid = videoinput('winvideo',1);
   vid2 = videoinput('winvideo',1,'RGB24_176x144');
   vid3 = videoinput('winvideo',1,'YV12_352x288');
   ```
2. **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

   You can delete image acquisition objects one at a time, using the delete function.

   ```
   delete(vid)
   ```

   You can also delete all the video input objects that currently exist in memory in one call to `delete` by using the `imaqfind` function. The `imaqfind` function returns an array of all the video input objects in memory.

   ```
   imaqfind

      Video Input Object Array:

      Index:   Type:          Name:
      1        videoinput     RGB555_128x96-winvideo-1
      2        videoinput     RGB24_176x144-winvideo-1
      3        videoinput     YV12_352x288-winvideo-1
   ```

   Nest a call to the `imaqfind` function within the `delete` function to delete all these objects from memory.

   ```
   delete(imaqfind)
   ```

   Note that the variables associated with the objects remain in the workspace.

   ```
   whos
     Name       Size                  Bytes  Class

     vid        1x1                    1120  videoinput object
     vid2       1x1                    1120  videoinput object
     vid3       1x1                    1120  videoinput object
     vids       1x3                    1280  videoinput object
   ```

   These variables are not valid image acquisition objects.

```
isvalid(vid)

ans =
    0
```

To remove these variables from the workspace, use the `clear` command.

Saving Image Acquisition Objects

In this section...

Using the save Command

You can save a video input object to a MAT-file just as you would any workspace variable by using the `save` command. This example saves the video input object `vid` to the MAT-file `myvid.mat`.

```
save myvid vid
```

When you save a video input object, all the video source objects associated with the video input object are also saved.

To load an image acquisition object that was saved to a MAT-file into the MATLAB workspace, use the `load` command. For example, to load `vid` from MAT-file `myvid.mat`, use

```
load myvid
```

Note The values of read-only properties are not saved. When you load an image acquisition object into the MATLAB workspace, read-only properties revert to their default values. To determine if a property is read only, use the `propinfo` function or read the property reference page.

Using the obj2mfile Command

Another way to save a video input object is to create an M-file that contains the set of commands used to create the video input object and configure its properties. You can use the `obj2mfile` function to create such an M-file. When you execute the M-file, it can create a new video input object or reuses an existing video input object, if one exists that has the same video format and adaptor.

Image Acquisition Toolbox Properties

The following properties are available in the toolbox.

- `BayerSensorAlignment`
- `DeviceID`
- `DiskLogger`
- `DiskLoggerFrameCount`
- `ErrorFcn`
- `EventLog`
- `FrameGrabInterval`
- `FramesAcquired`
- `FramesAcquiredFcn`
- `FramesAcquiredFcnCount`
- `FramesAvailable`
- `FramesPerTrigger`
- `IgnoreDroppedFrames`
- `InitialTriggerTime`
- `Logging`
- `LoggingMode`
- `Name`
- `NumberOfBands`
- `NumDroppedFrames`
- `Parent`
- `PreviewFullBitDepth`
- `Previewing`
- `ReturnedColorSpace`
- `ROIPosition`
- `Running`
- `Selected`
- `SelectedSourceName`
- `Source`
- `SourceName`
- `StartFcn`
- `StopFcn`
- `Tag`
- `Timeout`
- `TimerFcn`
- `TimerPeriod`

- `TriggerCondition`
- `TriggerFcn`
- `TriggerFrameDelay`
- `TriggerRepeat`
- `TriggersExecuted`
- `TriggerSource`
- `TriggerType`
- `Type`
- `UserData`
- `VideoFormat`
- `VideoResolution`

6

Acquiring Image Data

Acquiring Image Data

The core of any image acquisition application is the data acquired from the input device. A *trigger* is the event that initiates the acquisition of image frames, a process called *logging*. A trigger event occurs when a certain condition is met. For some types of triggers, the condition can be the execution of a toolbox function. For other types of triggers, the condition can be a signal from an external source that is monitored by the image acquisition hardware.

The following topics describe how to configure and use the various triggering options supported by the Image Acquisition Toolbox software and control other acquisition parameters.

- "Data Logging" on page 6-3
- "Setting the Values of Trigger Properties" on page 6-6
- "Specifying the Trigger Type" on page 6-8
- "Controlling Logging Parameters" on page 6-19
- "Waiting for an Acquisition to Finish" on page 6-27
- "Managing Memory Usage" on page 6-30
- "Logging Image Data to Disk" on page 6-32

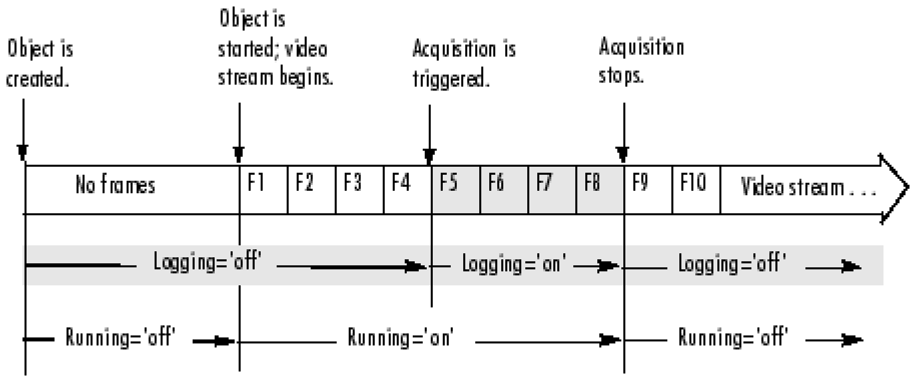
Data Logging

In this section...
"Overview" on page 6-3
"Trigger Properties" on page 6-4

Overview

When a trigger occurs, the toolbox sets the object's `Logging` property to `'on'` and starts storing the acquired frames in a buffer in memory, a disk file, or both. When the acquisition stops, the toolbox sets the object's `Logging` property to `'off'`.

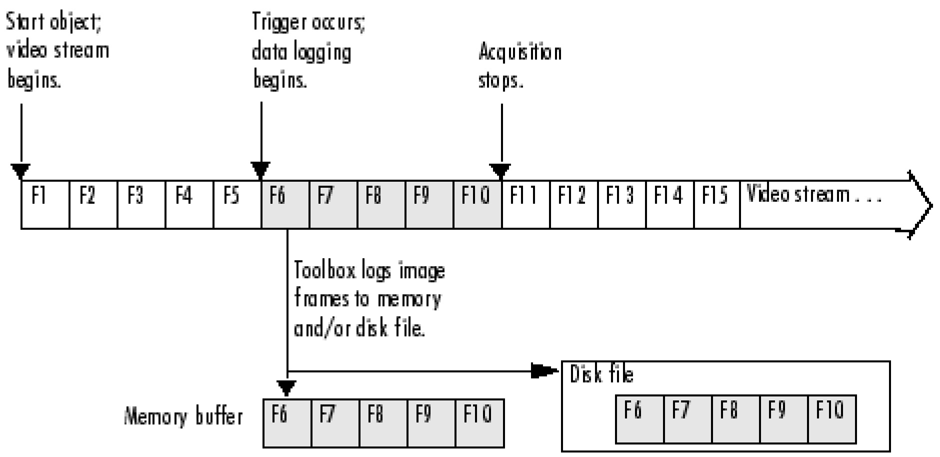
The following figure illustrates when an object moves into a logging state and the relation between running and logging states.



Logging State Transitions

Note After `Logging` is set to `'off'`, it is possible that the object might still be logging data to disk. To determine when disk logging is complete, check the value of the `DiskLoggerFrameCount` property. For more information, see "Logging Image Data to Disk" on page 6-32.

The following figure illustrates a group of frames being acquired from the video stream and being logged to memory and disk.


Start object; video stream begins.
Trigger occurs; data logging begins.
Acquisition stops.
F1
F2
F3
F4
F5
F6
F7
F8
F9
F10
F11
F12
F13
F14
F15
Video stream . . .
Toolbox logs image frames to memory and/or disk file.
Memory buffer
Disk file

Overview of Data Logging

Trigger Properties

The video input object supports several properties that you can use to configure aspects of trigger execution. Some of these properties return information about triggers. For example, to find out when the first trigger occurred, look at the value of the `InitialTriggerTime` property. Other properties enable you to control trigger behavior. For example, you use the `TriggerRepeat` property to specify how many additional times an object should execute a trigger.

The following table provides a brief description of all the trigger-related properties supported by the video input object. For information about how to set these properties, see “Setting the Values of Trigger Properties” on page 6-6.

Property	**Description**
`InitialTriggerTime`	Reports the absolute time when the first trigger executed.
`TriggerCondition`	Specifies the condition that must be met for a trigger to be executed. This property is always set to `'none'` for immediate and manual triggers.
`TriggerFcn`	Specifies the callback function to execute when a trigger occurs. For more information about callbacks, see “Using Events and Callbacks” on page 8-2.
`TriggerFrameDelay`	Specifies the number of frames to skip before logging data to memory, disk, or both. For more information, see “Delaying Data Logging After a Trigger” on page 6-24.
`TriggerRepeat`	Specifies the number of additional times to execute a trigger. If the value of `TriggerRepeat` is 0 (zero), the trigger executes but is not repeated any additional times. For more information, see “Specifying Multiple Triggers” on page 6-25.
`TriggersExecuted`	Reports the number of triggers that have been executed.
`TriggerSource`	Specifies the source to monitor for a trigger condition to be met. This property is always set to `'none'` for immediate and manual triggers.

Property	Description
`TriggerType`	Specifies the type of trigger: `'immediate'`, `'manual'`, or `'hardware'`. Use the `triggerinfo` function to determine whether your image acquisition device supports hardware triggers.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Setting the Values of Trigger Properties

In this section...
"About Trigger Properties" on page 6-6
"Specifying Trigger Type, Source, and Condition" on page 6-6

About Trigger Properties

Most trigger properties can be set in the same way you set any other image acquisition object property: referencing the property as you would a field in a structure using dot notation. For example, you can specify the value of the `TriggerRepeat` property, where `vid` is a video input object created using the `videoinput` function.

```
vid.TriggerRepeat = Inf
```

For more information, see "Configuring Image Acquisition Object Properties" on page 5-12.

Some trigger properties, however, are interrelated and require the use of the `triggerconfig` function to set their values. These properties are the `TriggerType`, `TriggerCondition`, and `TriggerSource` properties. For example, some `TriggerCondition` values are only valid when the value of the `TriggerType` property is `'hardware'`.

Specifying Trigger Type, Source, and Condition

Setting the values of the `TriggerType`, `TriggerSource`, and `TriggerCondition` properties can be a two-step process:

1. Determine valid configurations on page 6-6 of these properties by calling the `triggerinfo` function.
2. Set the values of these properties by calling the `triggerconfig` function.

For an example of using these functions, see "Using a Hardware Trigger" on page 6-12.

Determining Valid Configurations

To find all the valid configurations of the `TriggerType`, `TriggerSource`, and `TriggerCondition` properties, use the `triggerinfo` function, specifying a video input object as an argument.

```
config = triggerinfo(vid);
```

This function returns an array of structures, one structure for each valid combination of property values. Each structure in the array is made up of three fields that contain the values of each of these trigger properties. For example, the structure returned for an immediate trigger always has these values:

```
         TriggerType: 'immediate'
    TriggerCondition: 'none'
       TriggerSource: 'none'
```

A device that supports hardware configurations might return the following structure.

```
         TriggerType: 'hardware'
    TriggerCondition: 'risingEdge'
       TriggerSource: 'TTL'
```

Note The character vectors used as the values of the `TriggerCondition` and `TriggerSource` properties are device specific. Your device, if it supports hardware triggers, might support different condition and source values.

Configuring Trigger Type, Source, and Condition Properties

To set the values of the `TriggerType`, `TriggerSource`, and `TriggerCondition` properties, you must use the `triggerconfig` function. You specify the value of the property as an argument to the function.

For example, this code sets the values of these properties for a hardware trigger.

```
triggerconfig(vid,'hardware','risingEdge','TTL')
```

If you are specifying a manual trigger, you only need to specify the trigger type value as an argument.

```
triggerconfig(vid,'manual')
```

You can also pass one of the structures returned by the `triggerinfo` function to the `triggerconfig` function and set all three properties at once.

```
triggerconfig(vid, config(1))
```

See the `triggerconfig` function documentation for more information.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Specifying the Trigger Type

In this section...

Comparison of Trigger Types

To specify the type of trigger you want to execute, set the value of the `TriggerType` property of the video input object. You must use the `triggerconfig` function to set the value of this property. The following table lists all the trigger types supported by the toolbox, with information about when to use each type of trigger.

Comparison of Trigger Types

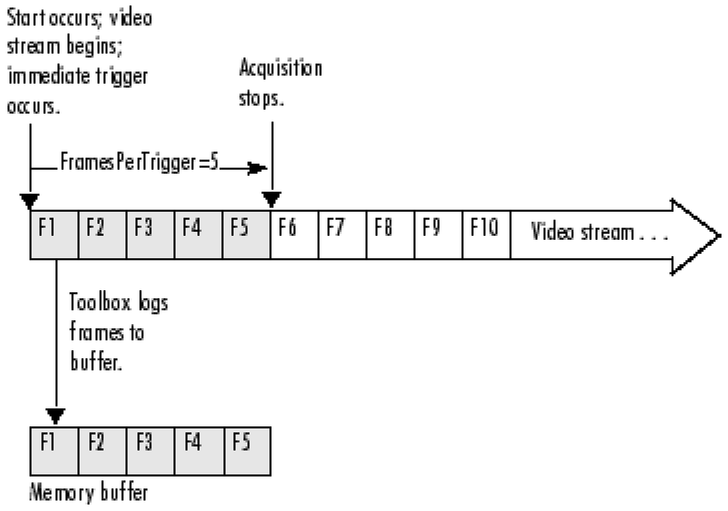
TriggerType Value	TriggerSource and TriggerCondition Values	Description
`'immediate'`	Always `'none'`	The trigger occurs automatically, immediately after the `start` function is issued. This is the default trigger type. For more information, see "Using an Immediate Trigger" on page 6-9.
`'manual'`	Always `'none'`	The trigger occurs when you issue the `trigger` function. A manual trigger can provide more control over image acquisition. For example, you can monitor the video stream being acquired, using the `preview` function, and manually execute the trigger when you observe a particular condition in the scene. For more information, see "Using a Manual Trigger" on page 6-10.
`'hardware'`	Device-specific	Hardware triggers are external signals that are processed directly by the hardware. This type of trigger is used when synchronization with another device is part of the image acquisition setup or when speed is required. A hardware device can process an input signal much faster than software. For more information, see "Using a Hardware Trigger" on page 6-12. **Note** Only a subset of image acquisition devices supports hardware triggers. To determine the trigger types supported by your device, see "Determining Valid Configurations" on page 6-6.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For

information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Using an Immediate Trigger

To use an immediate trigger, simply create a video input object. Immediate triggering is the default trigger type for all video input objects. With an immediate trigger, the object executes the trigger immediately after you start the object running with the `start` command. The following figure illustrates an immediate trigger.



Immediate Trigger

The following example illustrates how to use an immediate trigger:

1 **Create an image acquisition object** — This example creates a video input object for a Matrox image acquisition device. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

```
vid = videoinput('matrox',1);
```

Verify that the object has not acquired any frames.

```
vid.FramesAcquired
ans =

    0
```

2 **Configure properties** — To use an immediate trigger, you do not have to configure the `TriggerType` property because `'immediate'` is the default trigger type. You can verify this by using the `triggerconfig` function to view the current trigger configuration or by viewing the video input object's properties.

```
triggerconfig(vid)
ans =

             TriggerType: 'immediate'
```

```
    TriggerCondition: 'none'
       TriggerSource: 'none'
```

This example sets the value of the `FramesPerTrigger` property to `5`. (The default is 10 frames per trigger.)

```
vid.FramesPerTrigger = 5
```

3 **Start the image acquisition object** — Call the `start` function to start the image acquisition object. By default, the object executes an immediate trigger and acquires five frames of data, logging the data to a memory buffer. After logging the specified number of frames, the object stops running.

```
start(vid)
```

To verify that the object acquired data, view the value of the `FramesAcquired` property. The object updates the value of this property as it acquires data.

```
vid.FramesAcquired
ans =

   5
```

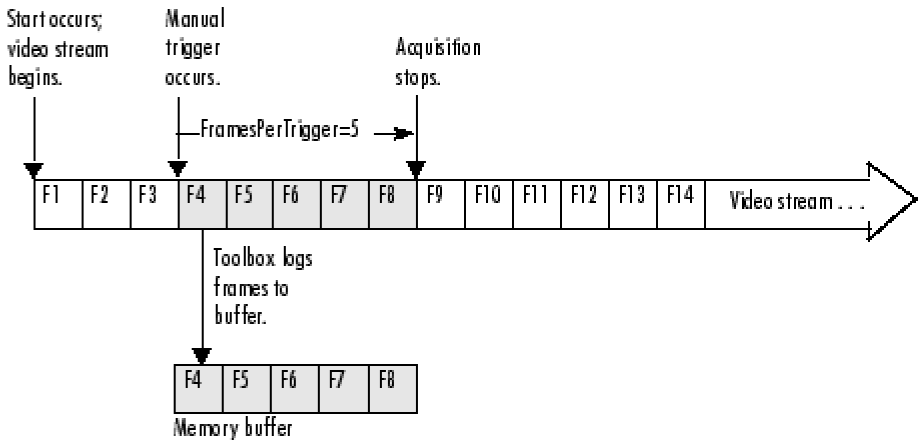
To execute another immediate trigger, you must restart the object. Note, however, that this deletes the data acquired by the first trigger. To execute multiple immediate triggers, specify a value for the `TriggerRepeat` property. See "Specifying Multiple Triggers" on page 6-25 for more information.

4 **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

```
delete(vid)
clear vid
```

Using a Manual Trigger

To use a manual trigger, create a video input object and set the value of the `TriggerType` property to `'manual'`. A video input object executes a manual trigger after you issue the `trigger` function. The following figure illustrates a manual trigger.



Manual Trigger

The following example illustrates how to use a manual trigger:

1 **Create an image acquisition object** — This example creates a video input object for a webcam image acquisition device. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

```
vid = videoinput('winvideo',1);
```

Verify that the object has not acquired any frames.

```
vid.FramesAcquired
ans =
    0
```

2 **Configure properties** — Set the video input object's `TriggerType` property to `'Manual'`. To set the values of certain trigger properties, including the `TriggerType` property, you must use the `triggerconfig` function. See “Setting the Values of Trigger Properties” on page 6-6 for more information.

```
triggerconfig(vid, 'Manual')
```

This example also sets the value of the `FramesPerTrigger` property to `5`. (The default is 10 frames per trigger.)

```
vid.FramesPerTrigger = 5
```

3 **Start the image acquisition object** — Call the `start` function to start the image acquisition object.

```
start(vid);
```

The video object is now running but not logging. With manual triggers, the video stream begins when the object starts but no frames are acquired until the trigger executes.

```
isrunning(vid)

ans =

     1

islogging(vid)

ans =

     0
```

Verify that the object has still not acquired any frames.

```
vid.FramesAcquired
ans =
    0
```

4 **Execute the manual trigger** — Call the `trigger` function to execute the manual trigger.

```
trigger(vid)
```

The object initiates the acquisition of five frames. Check the `FramesAcquired` property again to verify that five frames have been acquired.

```
vid.FramesAcquired
ans =
    5
```

After it acquires the specified number of frames, the video input object stops running.

```
isrunning(vid)

ans =

     0
```

To execute another manual trigger, you must first restart the video input object. Note that this deletes the frames acquired by the first trigger. To execute multiple manual triggers, specify a value for the `TriggerRepeat` property. See "Specifying Multiple Triggers" on page 6-25 for more information.

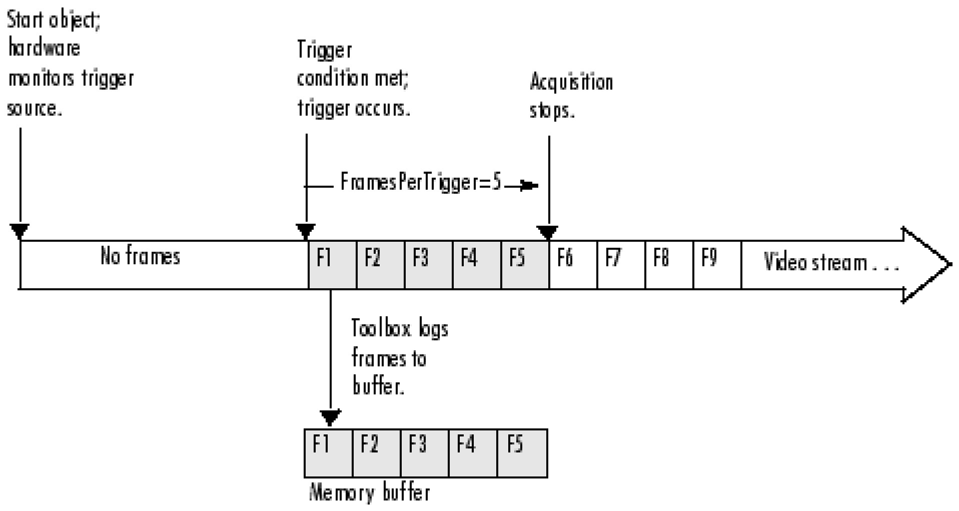
5 **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

```
delete(vid)
clear vid
```

Using a Hardware Trigger

To use a hardware trigger, create a video input object and set the value of the `TriggerType` property to `'hardware'`. You must also specify the source of the hardware trigger and the condition type. The hardware monitors the source you specify for the condition you specify. The following figure illustrates a hardware trigger. For hardware triggers, the video stream does not start until the trigger occurs.

Note Trigger sources and the conditions that control hardware triggers are device specific. Use the `triggerinfo` function to determine whether your image acquisition device supports hardware triggers and, if it does, which conditions you can configure. Refer to the documentation that came with your device for more detailed information about its hardware triggering capabilities.


Start object; hardware monitors trigger source.
Trigger condition met; trigger occurs.
Acquisition stops.
FramesPerTrigger=5
No frames
F1
F2
F3
F4
F5
F6
F7
F8
F9
Video stream . . .
Toolbox logs frames to buffer.
F1
F2
F3
F4
F5
Memory buffer

Hardware Trigger

The following example illustrates how to use a hardware trigger:

1. **Create an image acquisition object** — This example creates a video input object for a Matrox image acquisition device. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code. The device must support hardware triggers.

   ```
   vid = videoinput('matrox',1);
   ```
2. **Determine valid trigger property configurations** — Use the `triggerinfo` function to determine if your image acquisition device supports hardware triggers, and if it does, to find out valid configurations of the `TriggerSource` and `TriggerCondition` properties. See “Determining Valid Configurations” on page 6-6 for more information.

   In this example, `triggerinfo` returns the following valid trigger configurations.

   ```
   triggerinfo(vid)
   Valid Trigger Configurations:

         TriggerType:   TriggerCondition:   TriggerSource:
         'immediate'    'none'              'none'
         'manual'       'none'              'none'
           'hardware'     'risingEdge'        'TTL'
           'hardware'     'fallingEdge'       'TTL'
   ```
3. **Configure properties** — Configure the video input object trigger properties to one of the valid combinations returned by `triggerinfo`. You can specify each property value as an argument to the `triggerconfig` function

   ```
   triggerconfig(vid, 'hardware','risingEdge','TTL')
   ```

   Alternatively, you can set these values by passing one of the structures returned by the `triggerinfo` function to the `triggerconfig` function.

   ```
   configs = triggerinfo(vid);
   triggerconfig(vid,configs(3));
   ```

   This example also sets the value of the `FramesPerTrigger` property to `5`. (The default is 10 frames per trigger.)

   ```
   vid.FramesPerTrigger = 5
   ```
4. **Start the image acquisition object** — Call the `start` function to start the image acquisition object.

   ```
   start(vid)
   ```

   The object is running but not logging any data.

   ```
   isrunning(vid)

   ans =

        1

   islogging(vid)

   ans =

        0
   ```

The hardware begins monitoring the trigger source for the specified condition. When the condition is met, the hardware executes a trigger and begins providing image frames to the object. The object acquires the number of frames specified by the `FramesPerTrigger` property. View the value of the `FramesAcquired` property to see how much data was acquired. The object updates the value of this property as it acquires data.

```
vid.FramesAcquired
ans =

    5
```

After it executes the trigger and acquires the specified number of frames, the video input object stops running.

```
isrunning(vid)

ans =

     0
```

To execute another hardware trigger, you must first restart the video input object. Note that this deletes the frames acquired by the first trigger. To execute multiple triggers, specify a value for the `TriggerRepeat` property. See “Specifying Multiple Triggers” on page 6-25 for more information.

5. **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

```
delete(vid)
clear vid
```

Setting DCAM-Specific Trigger Modes

You can now use all trigger modes and all trigger inputs that DCAM cameras support. Previous toolbox releases supported only trigger mode `0`. Support for additional trigger modes and inputs do not affect any existing code you use.

Control trigger functionality using the `triggerinfo` and `triggerconfig` functions and the `triggersource` property. Before R2010a, one `triggersource` was available, `externalTrigger`. Selecting `externalTrigger` configures the camera to use trigger mode `0` with trigger source `0`.

The `triggersource` property is now composed of the trigger type (internal or external), the trigger source (0, 1, 2, etc.), and the mode number (0 through 5, 14 and 15). The following table summarizes the options.

Trigger Mode	Parameter	External Source	Multiple Frames Per Trigger
0	none	yes	yes
1	none	yes	no
2	(N >= 2)	yes	no
3	(N >= 1)	no	yes
4	(N >= 1)	yes	no

Trigger Mode	Parameter	External Source	Multiple Frames Per Trigger
5	(N >= 1)	yes	no
14	unknown	unknown	unknown
15	unknown	unknown	unknown

For example, the second `triggersource` for trigger mode 1 is called `externalTrigger1-mode1`. To use mode 3, the `triggersource` is `internalTrigger-mode3`.

Note Toolbox versions before R2010a supported DCAM trigger mode 0 with the first available `triggersource` as `externalTrigger`. The existing `externalTrigger` property will be maintained so to prevent backward compatibility issues. In addition, in order to preserve symmetry with the new functionality, `triggersource externalTrigger0-mode0`, which is synonymous, will also be supported. The new trigger modes do not work before R2010a.

Usage Notes

If a trigger mode has multiple trigger sources (modes `0`, `1`, `2`, `4`, and `5`), then `triggersource` has a digit indicating the corresponding camera source, even if only one camera source is available. For example, if the camera has only a single `triggersource` available, the toolbox reports the `triggersource` name as `externalTrigger0-modeX`. If the trigger mode does not have multiple sources (mode `3`), then no source digit appears in the name (i.e, `internalTriggerMode3` instead of `internalTriggerMode3-Source0`).

The DCAM adaptor includes a `TriggerParameter` property that is passed to the camera when you set trigger configurations. The `TriggerParameter` property is validated when you call `START` after selecting a hardware trigger mode.

If the selected trigger mode prohibits multiple frames per trigger, then an error appears when you call `START` without setting `FramesPerTrigger` to `1`.

If the camera supports only trigger mode `0` with source `0`, then the original functionality of having only the `externalTrigger triggersource` is supported.

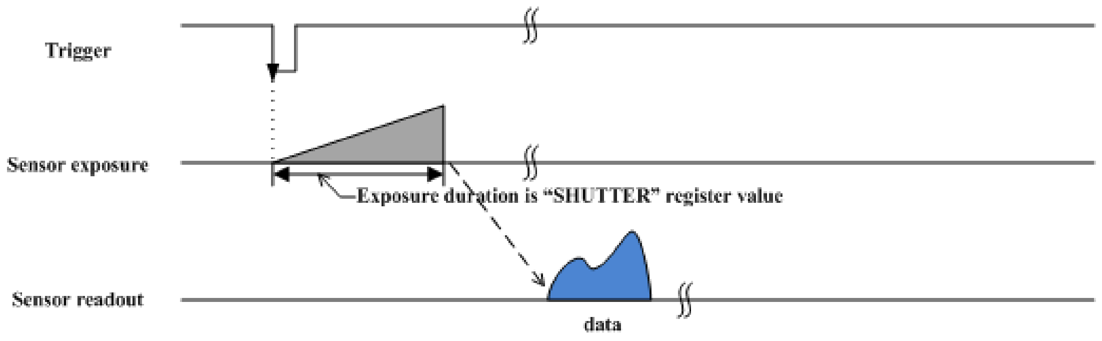
Trigger modes 14 and 15 are vendor-specific and are assumed to be external triggers and have no restrictions on any settings. You must validate any settings you use.

The following sections detail the trigger modes.

Trigger Mode 0

This is the only trigger mode supported before R2010a. When a trigger is received, a frame is acquired. You can acquire multiple frames per trigger by switching the camera for hardware triggered mode to free running mode when a triggered frame is acquired.

No parameter is required.

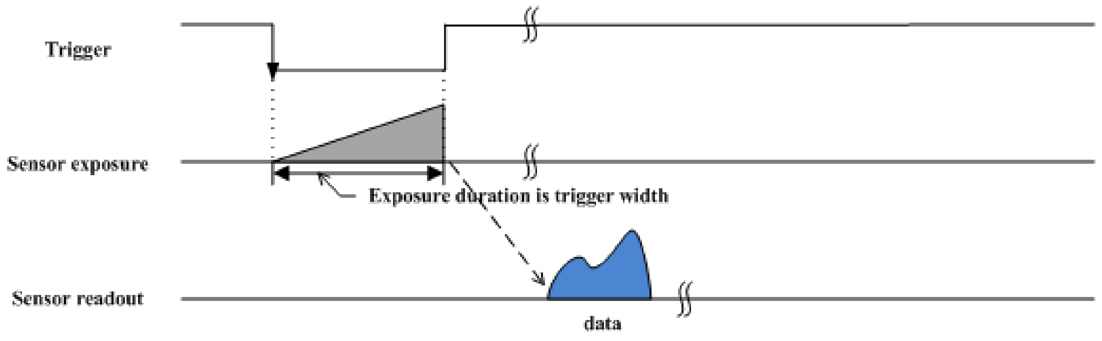


The camera starts the integration of the incoming light from the external trigger input falling edge.

Trigger Mode 1

In this mode, the duration of the trigger signal is used to control the integration time of the incoming light. This mode is used to synchronize the exposure time of the camera to an external event.

No parameter is required.

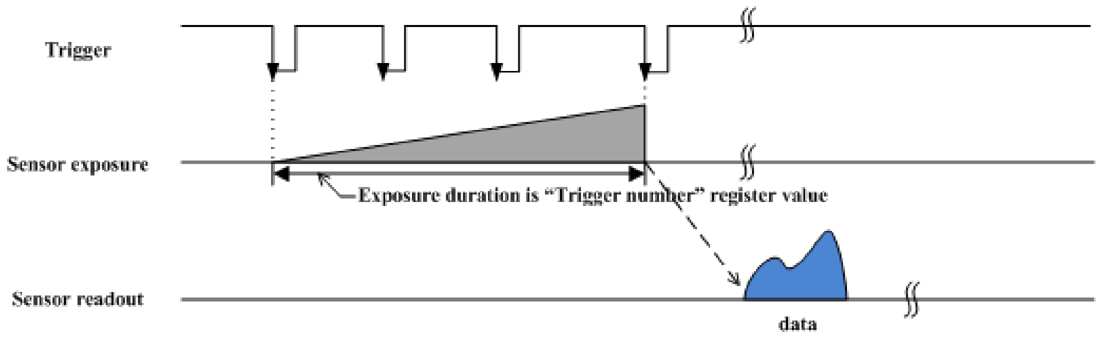


The camera starts the integration of the incoming light from the external trigger input falling edge. Integration time is equal to the low state time of the external trigger input if `triggersource` is `fallingEdge`, otherwise it is equal to the high state time.

Trigger Mode 2

This mode is similar to mode `1`, except the duration of the trigger signal does govern integration time. Instead the number of trigger signals received does. Integration commences upon the start of the first trigger signal and continues until the start of the `N`th trigger signal.

Parameter `N` is required and describes the number of trigger signals in an integration.

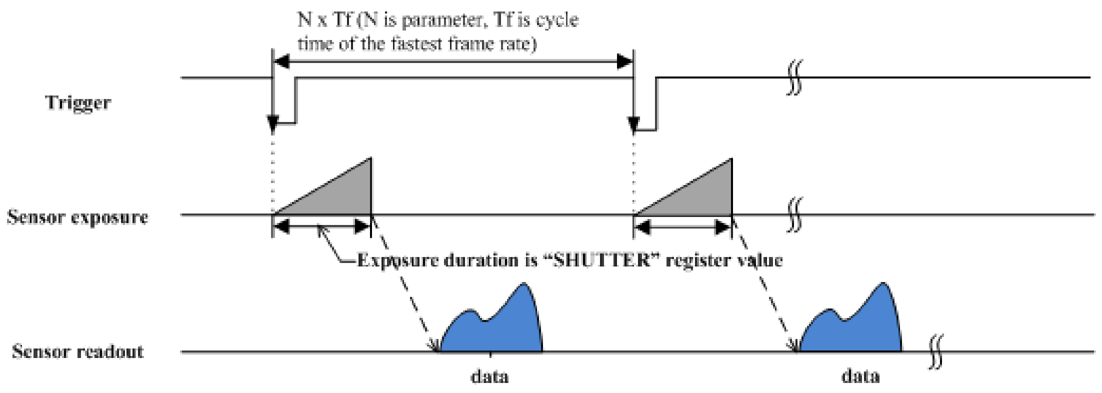


The camera starts the integration of the incoming light from the first external trigger input falling edge. At the Nth external trigger input falling edge, integration stops. Parameter N is required and must be 2 or greater. (N >= 2).

Trigger Mode 3

Use this internal trigger mode to achieve a lower frame rate. When the trigger generates internally, a frame is acquired and returned. A new frame is not acquired for N x Tf when N is the parameter and Tf is the cycle time of the fastest frame rate supported by the camera.

A parameter is required, as described above.

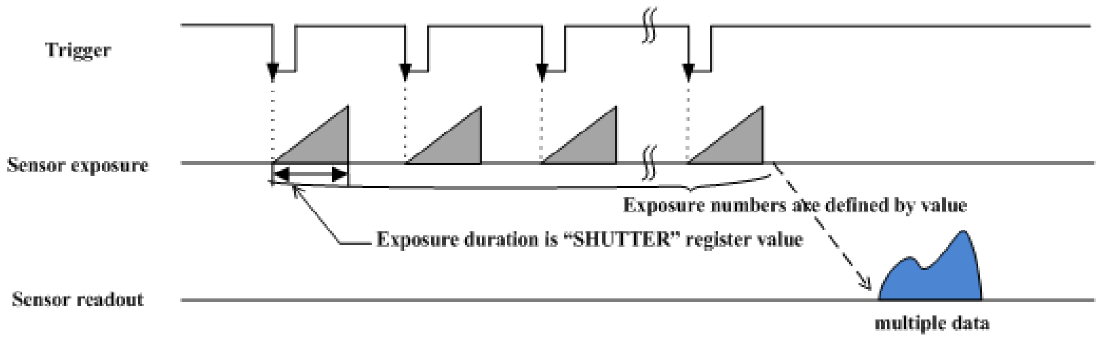

N x Tf (N is parameter, Tf is cycle time of the fastest frame rate)
Trigger
Sensor exposure
Exposure duration is "SHUTTER" register value
Sensor readout
data
data

This is an internal trigger mode. The camera issues the trigger internally and cycle time is N times of the cycle time of the fastest frame rate. Integration time of incoming light is described in the shutter register. Parameter N is required and must be 1 or greater (N >= 1).

Trigger Mode 4

This mode is the "multiple shutter preset mode." It is similar to mode 1, but the exposure time is governed by the shutter property. On each trigger, shutter property defines the exposure duration. When N triggers are received, a frame is acquired.

Parameter N is required and describes the number of triggers.

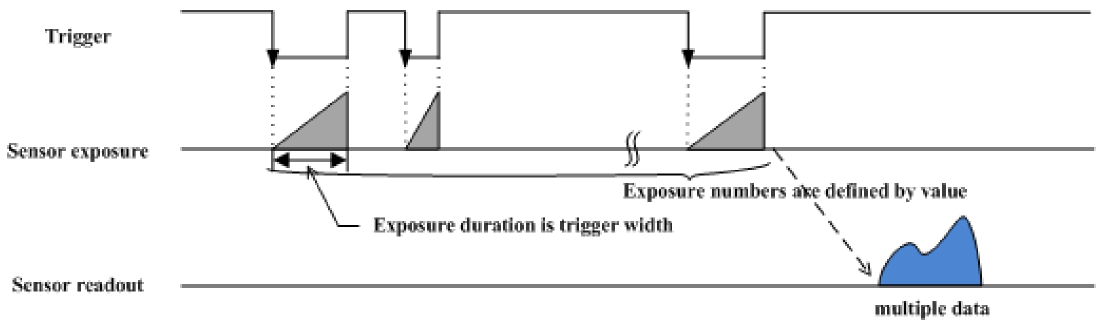

Trigger
Sensor exposure
Exposure numbers are defined by value
Exposure duration is "SHUTTER" register value
Sensor readout
multiple data

The camera starts integration of incoming light from the first external trigger input falling edge and exposes incoming light at shutter time. Repeat this sequence until the Nth external trigger input falling edge, then finish integration. Parameter N is required and must be 1 or greater (N >= 1).

Trigger Mode 5

This mode is the “multiple shutter pulse width mode.” It is a combination of modes 1 and 2. The exposure time is governed by the duration of the trigger signal and a number of trigger signals can be integrated into a single readout. If the trigger parameter is 1, this mode is degenerate with mode 1.

A parameter is required. The parameter describes the number of triggers.


Trigger
Sensor exposure
Exposure numbers are defined by value
Exposure duration is trigger width
Sensor readout
multiple data

The camera starts integration of incoming light from first the external trigger input falling edge and exposes incoming light until the trigger is inactive. Repeat this sequence until the Nth external trigger input falling edge, then finish integration. Parameter N is required and must be 1 or greater (N >= 1).

Trigger Mode 14

This is a vendor-specific mode and no information is available. Consult the documentation for your camera.

Trigger Mode 15

This is a vendor-specific mode and no information is available. Consult the documentation for your camera.

Controlling Logging Parameters

In this section...

Data Logging

The following subsections describe how to control various aspects of data logging.

- Specifying the logging mode on page 6-19
- Specifying the number of frames to log on page 6-20
- Determining how many frames have been logged on page 6-21 since the object was started
- Determining how many frames are currently available on page 6-22 in the memory buffer
- Delaying data logging on page 6-24 after a trigger executes
- Specifying multiple trigger executions on page 6-25

Specifying Logging Mode

Using the video input object `LoggingMode` property, you can control where the toolbox logs acquired frames of data.

The default value for the `LoggingMode` property is `'memory'`. In this mode, the toolbox logs data to a buffer in memory. If you want to bring image data into the MATLAB workspace, you must log frames to memory. The functions provided by the toolbox to move data into the workspace all work with the memory buffer. For more information, see “Bringing Image Data into the MATLAB Workspace” on page 7-3.

You can also log data to a disk file by setting the `LoggingMode` property to `'disk'` or to `'disk&memory'`. By logging frames to a disk file, you create a permanent record of the frames you acquire. For example, this code sets the value of the `LoggingMode` property of the video input object `vid` to `'disk&memory'`.

```
vid.LoggingMode = 'disk&memory';
```

Because the toolbox stores the image frames in Audio Video Interleaved (AVI) format, you can view the logged frames in any standard media player. For more information, see “Logging Image Data to Disk” on page 6-32.

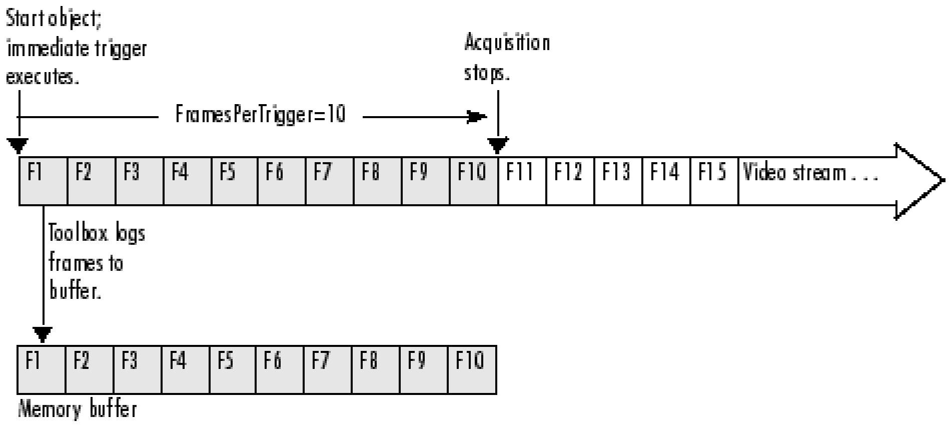
Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For

information about using this advanced tab completion feature, see "Using Tab Completion for Functions" on page 5-17.

Specifying the Number of Frames to Log

In the Image Acquisition Toolbox software, you specify the amount of data you want to acquire as the number of frames per trigger.

You specify the desired size of your acquisition as the value of the video input object `FramesPerTrigger` property. By default, the value of this property is 10 frames per trigger, but you can specify any value. The following figure illustrates an acquisition using the default value for the `FramesPerTrigger` property. To see an example of an acquisition, see "Acquiring 100 Frames" on page 6-21.



Specifying the Amount of Data to Log

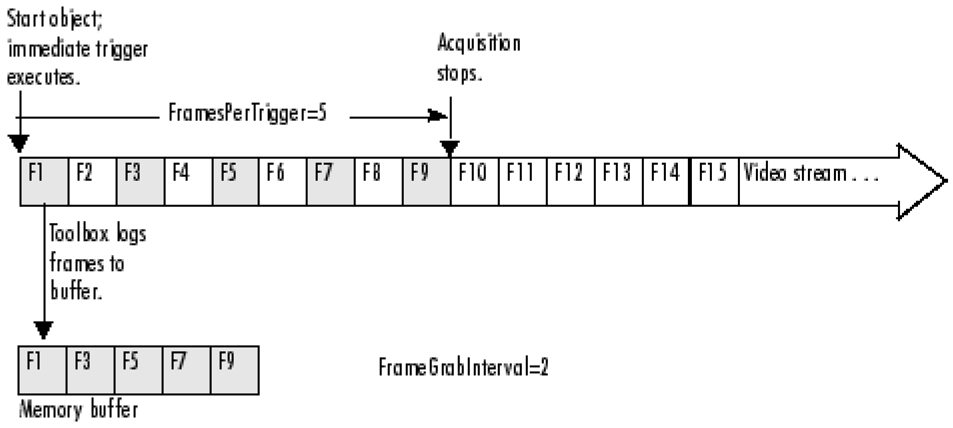
Note While you can specify any size acquisition, the number of frames you can acquire is limited by the amount of memory you have available on your system for image storage. A large acquisition can potentially fill all available system memory. For large acquisitions, you might want to remove frames from the buffer as they are logged. For more information, see "Moving Multiple Frames into the Workspace" on page 7-3. To learn how to empty the memory buffer, see "Freeing Memory" on page 6-30.

Specifying a Noncontiguous Acquisition

Although `FramesPerTrigger` specifies the number of frames to acquire, these frames do not have to be captured contiguously from the video stream. You can specify that the toolbox skip a certain number of frames between frames it acquires. To do this, set the value of the `FrameGrabInterval` property.

Note The `FrameGrabInterval` property controls the interval at which the toolbox acquires frames from the video stream (measured in frames). This property does not control the rate at which frames are provided by the device, otherwise known as the frame rate.

The following figure illustrates how the `FrameGrabInterval` property affects an acquisition.



Impact of FrameGrabInterval on Data Logging

Determining How Much Data Has Been Logged

To determine how many frames have been acquired by a video input object, check the value of the `FramesAcquired` property. This property tells how many frames the object has acquired since it was started. To determine how many frames are currently available in the memory buffer, see "Determining How Many Frames Are Available" on page 6-22.

Acquiring 100 Frames

This example illustrates how you can specify the amount of data to be acquired and determine how much data has been acquired. (For an example of configuring a time-based acquisition, see "Acquiring 10 Seconds of Image Data" on page 7-4.)

1. **Create an image acquisition object** — This example creates a video input object for a Windows image acquisition device. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

   ```
   vid = videoinput('winvideo',1);
   ```
2. **Configure properties** — Specify the amount of data you want to acquire as the number of frames per trigger. By default, a video input object acquires 10 frames per trigger. For this example, set the value of this property to `100`.

   ```
   vid.FramesPerTrigger = 100
   ```
3. **Start the image acquisition object** -— Call the `start` function to start the image acquisition object.

   ```
   start(vid)
   ```

   The object executes an immediate trigger and begins acquiring frames of data. To verify if the video input object is logging data, use the `islogging` function.

   ```
   islogging(vid)
   ans =

        1
   ```

The `start` function returns control to the command line immediately but the object continues logging the data to the memory buffer. After acquiring the specified number of frames, the object stops running and logging.

4 **Check how many frames have been acquired** — To verify that the specified number of frames has been acquired, check the value of the `FramesAcquired` property. Note that the object continuously updates the value of the `FramesAcquired` property as the acquisition progresses. If you view the value of this property several times during an acquisition, you can see the number of frames acquired increase until logging stops.

```
vid.FramesAcquired
ans =

    100
```

5 **Clean up** Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

```
delete(vid)
clear vid
```

Determining How Many Frames Are Available

The `FramesAcquired` property tells how many frames the object has logged since it was started, described in “Determining How Much Data Has Been Logged” on page 6-21. Once you move frames from the memory buffer into the MATLAB workspace, the number of frames stored in the memory buffer will differ from the `FramesAcquired` value. To determine how many frames are currently available in the memory buffer, check the value of the `FramesAvailable` property.

Note The `FramesAvailable` property tells the number of frames in the memory buffer, *not* in the disk log, if `LoggingMode` is configured to `'disk'` or `'disk&memory'`. Because it takes longer to write frames to a disk file than to memory, the number of frames stored in the disk log might lag behind those stored in the memory buffer. To see how many frames are available in the disk log, look at the value of the `DiskLoggerFrameCount` property. See “Logging Image Data to Disk” on page 6-32 for more information.

This example illustrates the distinction between the `FramesAcquired` and the `FramesAvailable` properties:

1 **Create an image acquisition object** — This example creates a video input object for a Windows image acquisition device. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

```
vid = videoinput('winvideo',1);
```

2 **Configure properties** — For this example, configure an acquisition of 15 frames.

```
vid.FramesPerTrigger = 15
```

3 **Start the image acquisition object** — Call the `start` function to start the image acquisition object.

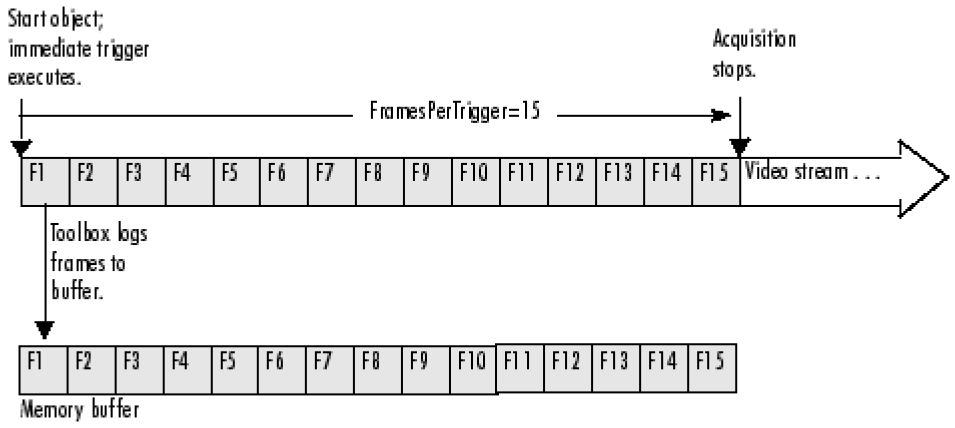
```
start(vid)
```

The object executes an immediate trigger and begins acquiring frames of data. The `start` function returns control to the command line immediately but the object continues logging the data to the memory buffer. After logging the specified number of frames, the object stops running.

4 **Check how many frames have been acquired** — To determine how many frames the object has acquired and how many frames are available in the memory buffer, check the value of the `FramesAcquired` and `FramesAvailable` properties.

```
vid.FramesAcquired
ans =

    15

vid.FramesAvailable

ans =

    15
```

The object updates the value of these properties continuously as it acquires frames of data. The following figure illustrates how the object puts acquired frames in the memory buffer as the acquisition progresses.



Frames Available After Initial Trigger Execution

5 **Remove frames from the memory buffer** — When you remove frames from the memory buffer, the object decrements the value of the `FramesAvailable` property by the number of frames removed.

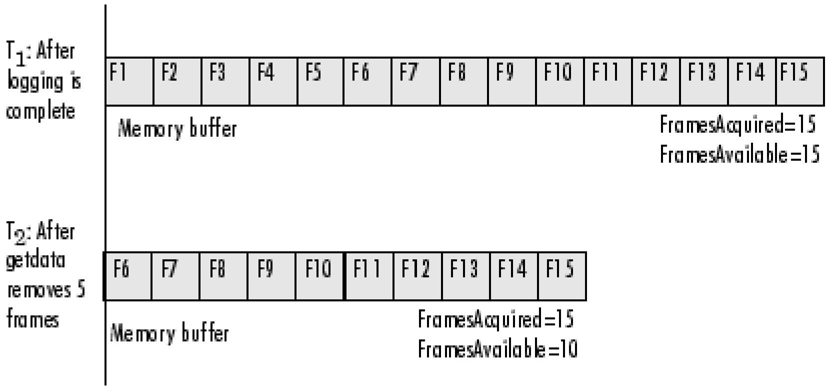
To remove frames from the memory buffer, call the `getdata` function, specifying the number of frames to retrieve. For more information about using `getdata`, see “Bringing Image Data into the MATLAB Workspace” on page 7-3.

```
data = getdata(vid,5);
```

After you execute the `getdata` function, check the values of the `FramesAcquired` and `FramesAvailable` properties again. Notice that the `FramesAcquired` property remains unchanged but the object has decremented the value of the `FramesAvailable` property by the number of frames removed from the memory buffer.

```
vid.FramesAcquired

ans =
    15

vid.FramesAvailable

ans =

    10
```

The following figure illustrates the contents of the memory buffer after frames are removed.



Contents of Memory Buffer Before and After Removing Frames

6 **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

```
delete(vid)
clear vid
```

Delaying Data Logging After a Trigger

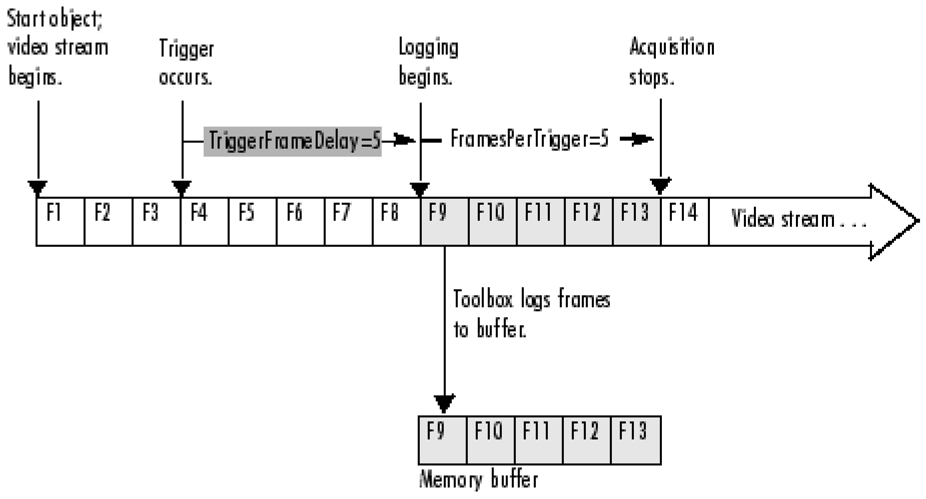
In some image acquisition setups, you might not want to log the first few frames returned from your camera or other imaging device. For example, some cameras require a short warmup time when activated. The quality of the first few images returned by these cameras might be too dark to be useful for your application.

To account for this characteristic of your setup, you can specify that the toolbox skip a specified number of frames after a trigger executes. You use the `TriggerFrameDelay` property to specify the number of frames you want to skip before logging begins.

For example, to specify a delay of five frames before data logging begins after a trigger executes, you would set the value of the `TriggerFrameDelay` property to 5. The number of frames captured is defined by the `FramesPerTrigger` property and is unaffected by the delay.

```
vid.TriggerFrameDelay = 5;
```

This figure illustrates this scenario.



Specifying a Delay Before Data Logging Begins

Specifying Multiple Triggers

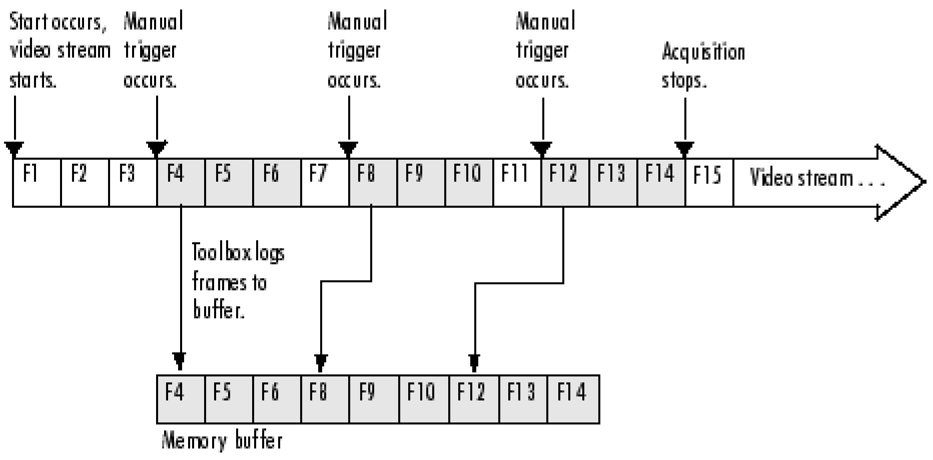
When a trigger occurs, a video input object acquires the number of frames specified by the `FramesPerTrigger` property and logs the data to a memory buffer, a disk file, or both.

When it acquires the specified number of frames, the video input object stops running. To execute another trigger, you must restart the video input object. Restarting an object causes it to delete all the data it has stored in the memory buffer from the previous trigger. To execute multiple triggers, retaining the data from each trigger, you must specify a value for the `TriggerRepeat` property.

Note that the `TriggerRepeat` property specifies the number of *additional* times a trigger executes. For example, to execute a trigger three times, you would set the value of the `TriggerRepeat` property to `2`. In the following, `vid` is a video input object created with the `videoinput` function.

```
vid.TriggerRepeat = 2;
```

This figure illustrates an acquisition with three executions of a manual trigger. In the figure, the `FramesPerTrigger` property is set to 3.



Executing Multiple Triggers

Waiting for an Acquisition to Finish

In this section...

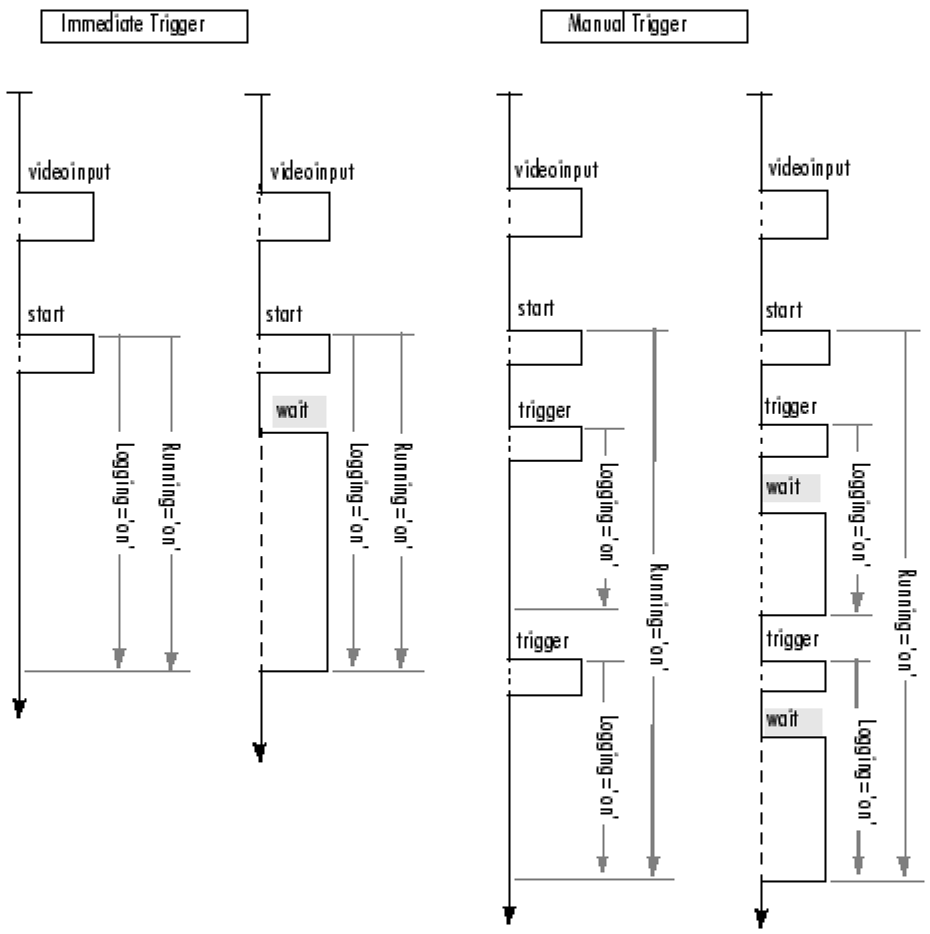
Using the wait Function

The `start` function and the `trigger` function are asynchronous functions. That is, they start the acquisition of frames and return control to the MATLAB command line immediately.

In some scenarios, you might want your application to wait until the acquisition completes before proceeding with other processing. To do this, call the `wait` function immediately after the `start` or `trigger` function returns. The `wait` function blocks the MATLAB command line until an acquisition completes or a timeout value expires, whichever comes first.

By default, `wait` blocks the command line until a video input object stops running. You can optionally specify that `wait` block the command line until the object stops logging. For acquisitions using an immediate trigger, video input objects always stop running and stop logging at the same time. However, with a manual trigger configured for multiple executions (`TriggerRepeat` > 0), you can use `wait` immediately after each call to the `trigger` function to block the command line while logging is in progress, even though the object remains in running state throughout the entire acquisition.

The following figure illustrates the flow of control at the MATLAB command line for a single execution of an immediate trigger and a manual trigger, with and without the `wait` function. A hardware trigger is similar to the manual trigger diagram, except that the acquisition is triggered by an external signal to the camera or frame grabber board, not by the `trigger` function. For an example, see "Blocking the Command Line Until an Acquisition Completes" on page 6-28.



Using wait to Block the MATLAB Command Line

Blocking the Command Line Until an Acquisition Completes

The following example illustrates how to use the `wait` function to put a 60 second time limit on the execution of a hardware trigger. If the hardware trigger does not execute within the time limit, `wait` returns control to the MATLAB command line.

1 **Create an image acquisition object** — This example creates a video input object for a Matrox image acquisition device. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

```
vid = videoinput('matrox',1);
```

2 **Configure a hardware trigger** — Use the `triggerinfo` function to determine valid configurations of the `TriggerSource` and `TriggerCondition` properties. See “Determining Valid Configurations” on page 6-6 for more information. In this example, `triggerinfo` returns the following valid trigger configurations.

```
triggerinfo(vid)
Valid Trigger Configurations:

      TriggerType:   TriggerCondition:   TriggerSource:
```

```
      'immediate'    'none'              'none'
      'manual'       'none'              'none'
        'hardware'     'risingEdge'        'TTL'
        'hardware'     'fallingEdge'       'TTL'
```

Configure the video input object trigger properties to one of the valid combinations returned by `triggerinfo`. You can specify each property value as an argument to the `triggerconfig` function

```
triggerconfig(vid, 'hardware','risingEdge','TTL')
```

Alternatively, you can set these values by passing one of the structures returned by the `triggerinfo` function to the `triggerconfig` function.

```
configs = triggerinfo(vid);
triggerconfig(vid,configs(3));
```

3 **Configure other object properties** — This example also sets the value of the `FramesPerTrigger` property to configure an acquisition large enough to produce a noticeable duration. (The default is 10 frames per trigger.)

```
vid.FramesPerTrigger = 100
```

4 **Start the image acquisition object** — Call the `start` function to start the image acquisition object.

```
start(vid)
```

The `start` function sets the object running and returns control to the command line.

5 **Block the command line until the acquisition finishes** — After the `start` function returns, call the `wait` function.

```
wait(vid,60)
```

The `wait` function blocks the command line until the hardware trigger fires and acquisition completes or until the amount of time specified by the timeout value expires.

6 **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

```
delete(vid)
clear vid
```

Managing Memory Usage

Freeing Memory

At times, while acquiring image data, you might want to delete some or all of the frames that are stored in memory. Using the `flushdata` function, you can delete all the frames currently stored in memory or only those frames associated with the execution of a trigger.

The following example illustrates how to use `flushdata` to delete all the frames in memory or one trigger's worth of frames.

1. **Create an image acquisition object** — This example creates a video input object for a Windows image acquisition device. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

   ```
   vid = videoinput('winvideo',1);
   ```
2. **Configure properties** — For this example, configure an acquisition of five frames per trigger and, to show the effect of `flushdata`, configure multiple triggers using the `TriggerRepeat` property.

   ```
   vid.FramesPerTrigger = 5
   vid.TriggerRepeat = 2;
   ```
3. **Start the image acquisition object** — Call the `start` function to start the image acquisition object.

   ```
   start(vid)
   ```

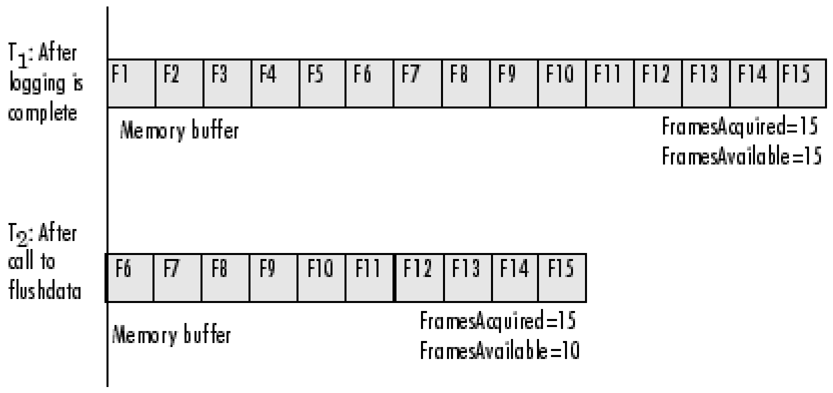
   The object executes an immediate trigger, acquires five frames of data, and repeats this trigger two more times. After logging the specified number of frames, the object stops running.

   To verify that the object acquired data, view the value of the `FramesAvailable` property. This property reports how many frames are currently stored in the memory buffer.

   ```
   vid.FramesAvailable
   ans =

      15
   ```
4. **Delete a trigger's worth of image data** — Call the `flushdata` function, specifying the mode `'triggers'`. This deletes the frames associated with the oldest trigger.

   ```
   flushdata(vid,'triggers');
   ```

   The following figure shows the frames acquired before and after the call to `flushdata`. Note how `flushdata` deletes the frames associated with the oldest trigger.



T_1: After logging is complete
F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 F13 F14 F15
Memory buffer
FramesAcquired=15
FramesAvailable=15
T_2: After call to flushdata
F6 F7 F8 F9 F10 F11 F12 F13 F14 F15
Memory buffer
FramesAcquired=15
FramesAvailable=10

To verify that the object deleted the frames, view the value of the `FramesAvailable` property.

```
vid.FramesAvailable
ans =

   10
```

5 **Empty the entire memory buffer** — Calling `flushdata` without specifying the mode deletes all the frames stored in memory.

```
flushdata(vid);
```

To verify that the object deleted the frames, view the value of the `FramesAvailable` property.

```
vid.FramesAvailable
ans =

   0
```

6 **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

```
delete(vid)
clear vid
```

Logging Image Data to Disk

In this section...

Formats for Logging Data to Disk

While a video input object is running, you can log image data being acquired to a disk file. Logging image data to disk provides a record of your data.

For the best performance, logging to disk requires a MATLAB `VideoWriter` object, which is a MATLAB function, not an Image Acquisition Toolbox function. After you create and configure a `VideoWriter` object, provide it to the `videoinput` object's `DiskLogger` property.

`VideoWriter` provides a number of different profiles that log the data in different formats. The following example uses the Motion JPEG 2000 profile, which can log single-banded (grayscale) data as well as multi-byte data. Supported profiles are:

- `'Motion JPEG 2000'` — Compressed Motion JPEG 2000 file.
- `'Archival'` — Motion JPEG 2000 file with lossless compression.
- `'Motion JPEG AVI'` — Compressed AVI file using Motion JPEG codec.
- `'Uncompressed AVI'` — Uncompressed AVI file with `RGB24` video.
- `'MPEG-4'` — Compressed MPEG-4 file with H.264 encoding (systems with Windows 7 or macOS 10.7 and later).
- `'Grayscale AVI'` — Uncompressed AVI file with grayscale video. Only used for monochrome devices.
- `'Indexed AVI'` — Uncompressed AVI file with indexed video. Only used for monochrome devices.

Logging Data to Disk Using VideoWriter

This example uses a GigE Vision device in a grayscale format (`Mono10`).

1. Create a video input object that accesses a GigE Vision image acquisition device and uses grayscale format at 10 bits per pixel.

   ```
   vidobj = videoinput('gige', 1, 'Mono10');
   ```
2. You can log acquired data to memory, to disk, or both. By default, data is logged to memory. To change the logging mode to disk, configure the video input object's `LoggingMode` property.

   ```
   vidobj.LoggingMode = 'disk'
   ```
3. Create a VideoWriter object with the profile set to Motion JPEG 2000.

   ```
   logfile = VideoWriter('logfile.mj2, 'Motion JPEG 2000')
   ```
4. Configure the video input object to use the `VideoWriter` object.

   ```
   vidobj.DiskLogger = logfile;
   ```

5 Now that the video input object is configured for logging data to a Motion JPEG 2000 file, initiate the acquisition.

```
start(vidobj)
```

6 Wait for the acquisition to finish.

```
wait(vidobj, 5)
```

7 When logging large amounts of data to disk, disk writing occasionally lags behind the acquisition. To determine whether all frames are written to disk, you can optionally use the `DiskLoggerFrameCount` property.

```
while (vidobj.FramesAcquired ~= vidobj.DiskLoggerFrameCount)
    pause(.1)
end
```

8 You can verify that the `FramesAcquired` and `DiskLoggerFrameCount` properties have identical values by using these commands and comparing the output.

```
vidobj.FramesAcquired
vidobj.DiskLoggerFrameCount
```

9 When the video input object is no longer needed, delete it and clear it from the workspace.

```
delete(vidobj)
clear vidobj
```

Guidelines for Using a VideoWriter Object to Log Image Data

Note the following when using VideoWriter.

- You should not delete the video input object until logging has been completed as indicated by the `DiskLoggerFrameCount` property equaling the `FramesAcquired` property. Doing so will cause disk logging to stop without all of the data being logged.
- If START is called multiple times without supplying a new VideoWriter object, the contents of the previous file will be erased when START is called.
- Once the VideoWriter object has been passed to the `DiskLogger` property, you should not modify it.

Working with Acquired Image Data

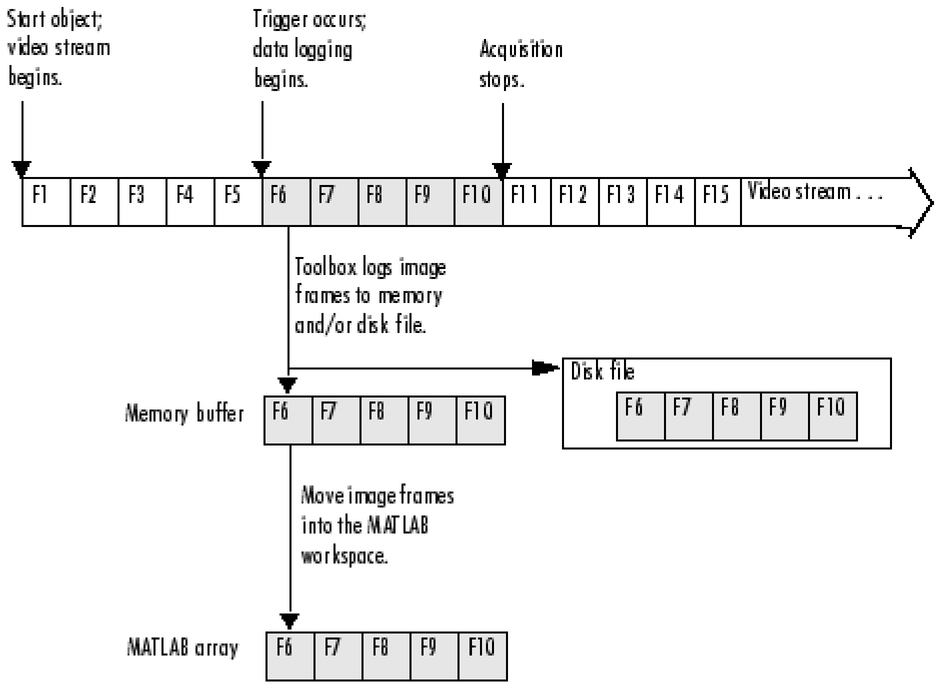
When you trigger an acquisition, the toolbox stores the image data in a memory buffer, a disk file, or both. To work with this data, you must bring it into the MATLAB workspace.

This chapter describes how you use video input object properties and toolbox functions to bring the logged data into the MATLAB workspace.

Image Acquisition Overview

When a trigger occurs, the toolbox acquires frames from the video stream and logs the frames to a buffer in memory, a disk file, or both, depending on the value of the `LoggingMode` property. To work with this logged image data, you must bring it into the MATLAB workspace.

The following figure illustrates a group of frames being acquired from the video stream, logged to memory and disk, and brought into the MATLAB workspace as a multidimensional numeric array. Note that when frames are brought into the MATLAB workspace, they are removed from the memory buffer.


Start object; video stream begins.
Trigger occurs; data logging begins.
Acquisition stops.
F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 F13 F14 F15
Video stream . . .
Toolbox logs image frames to memory and/or disk file.
Disk file
F6 F7 F8 F9 F10
Memory buffer
F6 F7 F8 F9 F10
Move image frames into the MATLAB workspace.
MATLAB array
F6 F7 F8 F9 F10

Overview of Image Acquisition

Bringing Image Data into the MATLAB Workspace

In this section...

Overview

The toolbox provides three ways to move frames from the memory buffer into the MATLAB workspace:

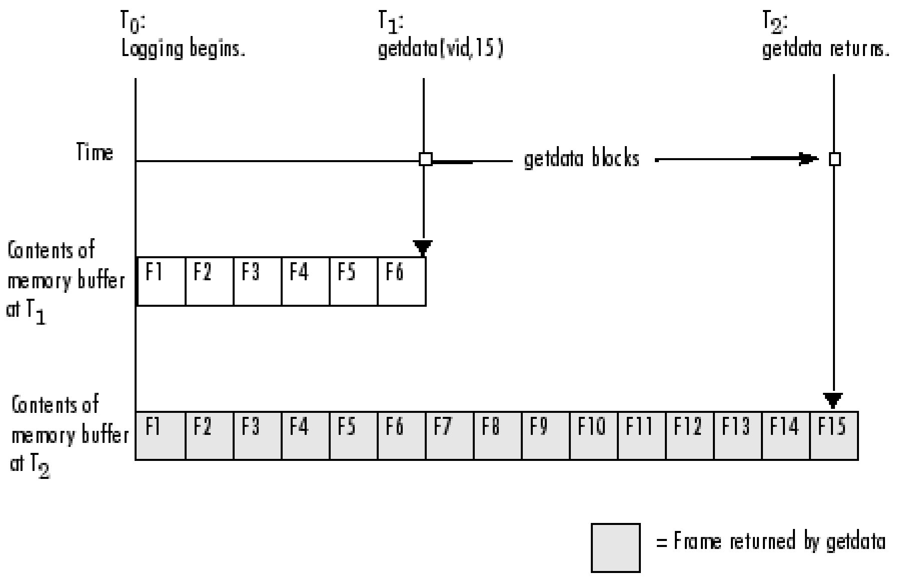
- **Removing multiple frames from the buffer** — To move a specified number of frames from the memory buffer into the workspace, use the `getdata` function. The `getdata` function removes the frames from the memory buffer as it moves them into the workspace. The function blocks the MATLAB command line until all the requested frames are available, or until a timeout value expires. For more information, see "Moving Multiple Frames into the Workspace" on page 7-3.
- **Viewing the most recently acquired frames in the buffer** — To bring the most recently acquired frames in the memory buffer into the workspace *without* removing them from the buffer, use the `peekdata` function. When returning frames, `peekdata` starts with the most recently acquired frame and works backward in the memory buffer. In contrast, `getdata` starts at the beginning of the buffer, returning the oldest acquired frame first. `peekdata` does not block the command line and is not guaranteed to return all the frames you request. For more information, see "Viewing Frames in the Memory Buffer" on page 7-5.
- **Bringing a single frame of data into the workspace** — As a convenience, the toolbox provides the `getsnapshot` function, which returns a single frame of data into the MATLAB workspace. Because the `getsnapshot` function does not require starting the object or triggering an acquisition, it is the easiest way to bring image data into the workspace. `getsnapshot` is independent of the memory buffer; it can return a frame even if the memory buffer is empty, and the frame returned does not affect the value of the `FramesAvailable` property. For more information, see "Bringing a Single Frame into the Workspace" on page 7-7. For an example of using `getsnapshot`, see the Image Acquisition Toolbox example **Acquiring a Single Image in a Loop** in the **Examples** list at the top of the Image Acquisition Toolbox main Documentation Center page, or open the file demoimaq_GetSnapshot.m in the MATLAB Editor.

Moving Multiple Frames into the Workspace

To move multiple frames of data from the memory buffer into the MATLAB workspace, use the `getdata` function. By default, `getdata` retrieves the number of frames specified in the `FramesPerTrigger` property but you can specify any number. See the `getdata` reference page for complete information about this function.

Note When the `getdata` function moves frames from the memory buffer into the workspace, it removes the frames from the memory buffer.

In this figure, `getdata` is called at T_1 with a request for 15 frames but only six frames are available in the memory buffer. `getdata` blocks until the specified number of frames becomes available, at T_2, at which point `getdata` moves the frames into the MATLAB workspace and returns control to the command prompt.


T0: Logging begins.
T1: getdata(vid,15)
T2: getdata returns.
Time
getdata blocks
Contents of memory buffer at T1
F1 F2 F3 F4 F5 F6
Contents of memory buffer at T2
F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 F13 F14 F15
= Frame returned by getdata

getdata Blocks Until Frames Become Available

Acquiring 10 Seconds of Image Data

This example shows how you can configure an approximate time-based acquisition using the `FramesPerTrigger` property:

1 **Create an image acquisition object** — This example creates a video input object for a Windows image acquisition device. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

```
vid = videoinput('winvideo',1);
```

2 **Configure properties** — To acquire 10 seconds of data, determine the frame rate of your image acquisition device and then multiply the frame rate by the number of seconds of data you want to acquire. The product of this multiplication is the value of the `FramesPerTrigger` property.

For this example, assume a frame rate of 30 frames per second (fps). Multiplying 30 by 10, you need to set the `FramesPerTrigger` property to the value 300.

```
vid.FramesPerTrigger = 300;
```

3 **Start the image acquisition object** — Call the `start` function to start the image acquisition object.

```
start(vid)
```

The object executes an immediate trigger and begins acquiring frames of data. The `start` function returns control to the command line immediately but the object continues logging the data to the memory buffer. After logging the specified number of frames, the object stops running.

4 **Bring the acquired data into the workspace** — To verify that you acquired the amount of data you wanted, use the optional `getdata` syntax that returns the timestamp of every frame

acquired. The difference between the first timestamp and the last timestamp should approximate the amount of data you expected.

```
[data time] = getdata(vid,300);

elapsed_time = time(300) - time(1)

     10.0467
```

5 **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

```
delete(vid)
clear vid
```

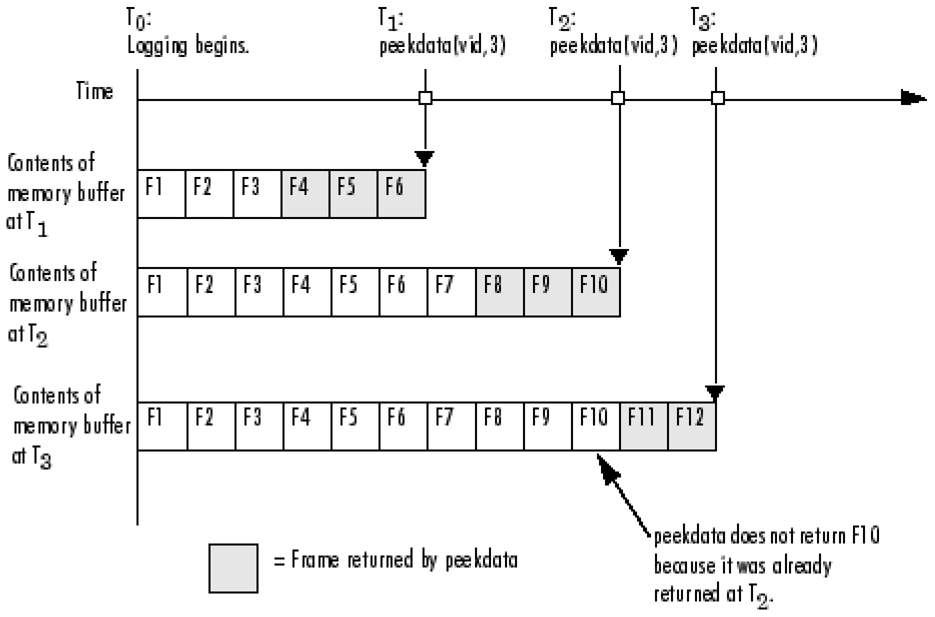
Viewing Frames in the Memory Buffer

To view sample frames from the memory buffer without removing them, use the `peekdata` function.

The `peekdata` function always returns the most recently acquired frames in the memory buffer. For example, if you request three frames, `peekdata` returns the most recently acquired frame in the buffer at the time of the request and the two frames that immediately precede it.

The following figure illustrates this process. The command `peekdata(vid,3)` is called at three different times (T_1, T_2, and T_3). The shaded frames indicate the frames returned by `peekdata` at each call. (`peekdata` returns frames without removing them from the memory buffer.)

Note in the figure that, at T_3, only two frames have become available since the last call to `peekdata`. In this case, `peekdata` returns only the two frames, with a warning that it returned less data than was requested.


T0: Logging begins.
T1: peekdata(vid,3)
T2: peekdata(vid,3)
T3: peekdata(vid,3)
Time
Contents of memory buffer at T1
F1 F2 F3 F4 F5 F6
Contents of memory buffer at T2
F1 F2 F3 F4 F5 F6 F7 F8 F9 F10
Contents of memory buffer at T3
F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12
= Frame returned by peekdata
peekdata does not return F10 because it was already returned at T2.

Frames Returned by peekdata

Note The `peekdata` function does not return any data while running if in disk logging mode.

The following example illustrates how to use `peekdata`:

1 **Create an image acquisition object** — This example creates a video input object for a Data Translation image acquisition device. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

```
vid = videoinput('dt',1);
```

2 **Configure properties** — For this example, configure a manual trigger. You must use the `triggerconfig` function to specify the trigger type.

```
triggerconfig(vid,'manual')
```

In addition, configure a large enough acquisition to allow several calls to `peekdata` before it finishes.

```
vid.FramesPerTrigger = 300;
```

3 **Start the image acquisition object** — Call the `start` function to start the image acquisition object.

```
start(vid)
```

The video object is now running but not logging.

```
isrunning(vid)

ans =

     1

islogging(vid)

ans =

     0
```

4 **Use peekdata to view frames before a trigger** — If you call `peekdata` before you trigger the acquisition, `peekdata` can only return a single frame of data because data logging has not been initiated and the memory buffer is empty. If more than one frame is requested, `peekdata` issues a warning that it is returning fewer than the requested number of frames.

```
pdata = peekdata(vid,50);
Warning: PEEKDATA could not return all the frames requested.
```

Verify that `peekdata` returned a single frame. A single frame of data should have the same width and height as specified by the `ROIPosition` property and the same number of bands, as specified by the `NumberOfBands` property. In this example, the video format of the data is RGB so the value of the `NumberOfBands` property is 3.

```
whos
  Name        Size                   Bytes  Class

  pdata       96x128x3               36864  uint8 array
  vid          1x1                    1060  videoinput object
```

Verify that the object has not acquired any frames.

```
vid.FramesAcquired
ans =
    0
```

5 **Trigger the acquisition** — Call the `trigger` function to trigger an acquisition.

```
trigger(vid)
```

The object begins logging frames to the memory buffer.

6 **View the most recently acquired frames** — While the acquisition is in progress, call `peekdata` several times to view the latest frames in the memory buffer. Depending on the number of frames you request, and the timing of these requests, `peekdata` might return fewer than the number of frames you specify.

```
pdata = peekdata(vid,50);
```

To verify that `peekdata` returned the frames you requested, check the dimensions of `pdata`. `peekdata` returns a four-dimensional array of frames, where the last dimension indicates the number of frames returned.

```
whos
  Name        Size                  Bytes   Class

  pdata       4-D                 1843200   uint8 array
  vid         1x1                    1060   videoinput object

size(pdata)

ans =

    96   128     3    50
```

7 **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

```
delete(vid)
clear vid
```

Bringing a Single Frame into the Workspace

To bring a single frame of image data into the MATLAB workspace, use the `getsnapshot` function. You can call the `getsnapshot` function at any time after object creation.

This example illustrates how simple it is to use the `getsnapshot` function.

1 **Create an image acquisition object** — This example creates a video input object for a Matrox device. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

```
vid = videoinput('matrox',1);
```

2 **Bring a frame into the workspace** — Call the `getsnapshot` function to bring a frame into the workspace. Note that you do not need to start the video input object before calling the `getsnapshot` function.

```
frame = getsnapshot(vid);
```

The `getsnapshot` function returns an image of the same width and height as specified by the `ROIPosition` property and the same number of bands as specified by the `NumberOfBands` property. In this example, the video format of the data is RGB so the value of the `NumberOfBands` property is 3.

```
whos
  Name        Size                  Bytes  Class

  frame      96x128x3               36864  uint8 array
  vid         1x1                    1060  videoinput object
```

Note that the frame returned by `getsnapshot` is not removed from the memory buffer, if frames are stored there, and does not affect the value of the `FramesAvailable` property.

3. **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

```
delete(vid)
clear vid
```

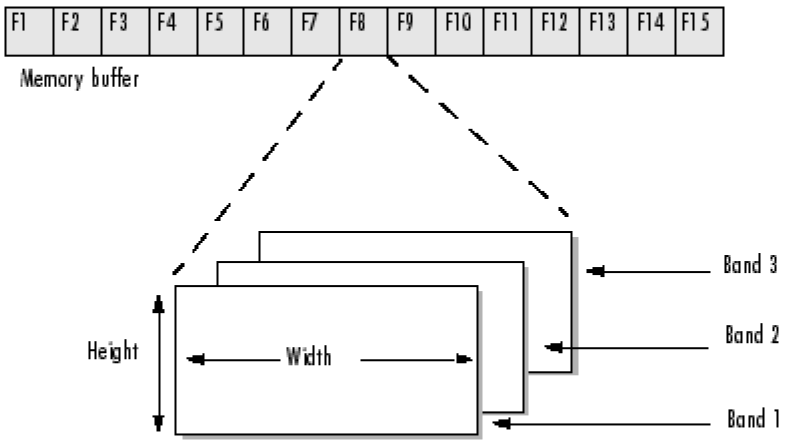
For an example of using `getsnapshot`, see the Image Acquisition Toolbox example **Acquiring a Single Image in a Loop** in the **Examples** list at the top of the Image Acquisition Toolbox main Documentation Center page, or open the file demoimaq_GetSnapshot.m in the MATLAB Editor.

Working with Image Data in MATLAB Workspace

In this section...

Understanding Image Data

The illustrations in this documentation show the video stream and the contents of the memory buffer as a sequence of individual frames. In reality, each frame is a multidimensional array. For more information about using multidimensional arrays, see "Multidimensional Arrays". The following figure illustrates the format of an individual frame.



Format of an Individual Frame

The following sections describes how the toolbox

- Determines the dimensions of the data returned on page 7-9
- Determines the data type used for the data on page 7-12
- Determines the color space of the data on page 7-14

This section also describes several ways to view acquired image data on page 7-12.

Determining the Dimensions of Image Data

The video format used by the image acquisition device is the primary determinant of the width, height, and the number of bands in each image frame. Image acquisition devices typically support multiple video formats. You select the video format when you create the video input object (described in "Specifying the Video Format" on page 5-8). The video input object stores the video format in the `VideoFormat` property.

The video input object stores the video resolution in the `VideoResolution` property.

Each image frame is three dimensional; however, the video format determines the number of bands in the third dimension. For color video formats, such as RGB, each image frame has three bands: one each for the red, green, and blue data. Other video formats, such as the grayscale RS170 standard, have only a single band. The video input object stores the size of the third dimension in the `NumberOfBands` property.

Note Because devices typically express video resolution as width-by-height, the toolbox uses this convention for the `VideoResolution` property. However, when data is brought into the MATLAB workspace, the image frame dimensions are listed in reverse order, height-by-width, because MATLAB expresses matrix dimensions as row-by-column.

ROIs and Image Dimensions

When you specify a region-of-interest (ROI) in the image being captured, the dimensions of the ROI determine the dimensions of the image frames returned. The `VideoResolution` property specifies the dimensions of the image data being provided by the device; the `ROIPosition` property specifies the dimensions of the image frames being logged. See the `ROIPosition` property reference page for more information.

Video Format and Image Dimensions

The following example illustrates how video format affects the size of the image frames returned.

1 **Select a video format** — Use the `imaqhwinfo` function to view the list of video formats supported by your image acquisition device. This example shows the video formats supported by a Matrox Orion frame grabber. The formats are industry standard, such as RS170, NTSC, and PAL. These standards define the image resolution.

```
info = imaqhwinfo('matrox');

info.DeviceInfo.SupportedFormats

ans =
  Columns 1 through 4

    'M_RS170'    'M_RS170_VIA_RGB'    'M_CCIR'    'M_CCIR_VIA_RGB'

  Columns 5 through 8

'M_NTSC'    'M_NTSC_RGB'      'M_NTSC_YC'    'M_PAL'

  Columns 9 through 10

'M_PAL_RGB'    'M_PAL_YC'
```

2 **Create an image acquisition object** — This example creates a video input object for a Matrox image acquisition device using the default video format, RS170. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

```
vid = videoinput('matrox',1);
```

3 **View the video format and video resolution properties** — The toolbox creates the object with the default video format. This format defines the video resolution.

```
vid.VideoFormat

ans =

   M_RS170

vid.VideoResolution

ans =

   [640 480]
```

4 **Bring a single frame into the workspace** — Call the `getsnapshot` function to bring a frame into the workspace.

```
frame = getsnapshot(vid);
```

The dimensions of the returned data reflect the image resolution and the value of the `NumberOfBands` property.

```
vid.NumberOfBands
ans =

   1

size(frame)

ans =

   480 640
```

5 **Start the image acquisition object** — Call the `start` function to start the image acquisition object.

```
start(vid)
```

The object executes an immediate trigger and begins acquiring frames of data.

6 **Bring multiple frames into the workspace** — Call the `getdata` function to bring multiple image frames into the MATLAB workspace.

```
data = getdata(vid,10);
```

The `getdata` function brings 10 frames of data into the workspace. Note that the returned data is a four-dimensional array: each frame is three-dimensional and the *n*th frame is indicated by the fourth dimension.

```
size(data)

ans =

   480 640 1 10
```

7 **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

```
delete(vid)
clear vid
```

Determining the Data Type of Image Frames

By default, the toolbox returns image frames in the data type used by the image acquisition device. If there is no MATLAB data type that matches the object's native data type, `getdata` chooses a MATLAB data type that preserves numerical accuracy. For example, in RGB 555 format, each color component is expressed in 5-bits. `getdata` returns each color as a `uint8` value.

You can specify the data type you want `getdata` to use for the returned data. For example, you can specify that `getdata` return image frames as an array of class `double`. To see a list of all the data types supported, see the `getdata` reference page.

The following example illustrates the data type of returned image data.

1. **Create an image acquisition object** — This example creates a video input object for a Matrox image acquisition device. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

   ```
   vid = videoinput('matrox',1);
   ```
2. **Bring a single frame into the workspace** — Call the `getsnapshot` function to bring a frame into the workspace.

   ```
   frame = getsnapshot(vid);
   ```
3. **View the class of the returned data** — Use the `class` function to determine the data type used for the returned image data.

   ```
   class(frame)

   ans =

     uint8
   ```
4. **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

   ```
   delete(vid)
   clear vid
   ```

Viewing Acquired Data

Once you bring the data into the MATLAB workspace, you can view it as you would any other image in MATLAB.

The Image Acquisition Toolbox software includes a function, `imaqmontage`, that you can use to view all the frames of a multiframe image array in a single MATLAB image object. `imaqmontage` arranges the frames so that they roughly form a square. `imaqmontage` can be useful for visually comparing multiple frames.

MATLAB includes two functions, `image` and `imagesc`, that display images in a figure window. Both functions create a MATLAB image object to display the frame. You can use image object properties to control aspects of the display. The `imagesc` function automatically scales the input data.

The Image Processing Toolbox software includes an additional display routine called `imshow`. Like `image` and `imagesc`, this function creates a MATLAB image object. However, `imshow` also automatically sets various image object properties to optimize the display.

Specifying the Color Space

In this section...

Specifying the Color Space

For most image acquisition devices, the video format of the video stream determines the color space of the acquired image data, that is, the way color information is represented numerically.

For example, many devices represent colors as RGB values. In this color space, colors are represented as a combination of various intensities of red, green, and blue. Another color space, widely used for digital video, is the YCbCr color space. In this color space, luminance (brightness or intensity) information is stored as a single component (Y). Chrominance (color) information is stored as two color-difference components (Cb and Cr). Cb represents the difference between the blue component and a reference value. Cr represents the difference between the red component and a reference value.

The toolbox can return image data in grayscale, RGB, and YCbCr. To specify the color representation of the image data, set the value of the `ReturnedColorSpace` property. To display image frames using the `image`, `imagesc`, or `imshow` functions, the data must use the RGB color space. Another MathWorks product, the Image Processing Toolbox software, includes functions that convert YCbCr data to RGB data, and vice versa.

Note Some devices that claim to support the YUV color space actually support the YCbCr color space. YUV is similar to YCbCr but not identical. The difference between YUV and YCbCr is the scaling factor applied to the result. YUV refers to a particular scaling factor used in composite NTSC and PAL formats. In most cases, you can specify the YCbCr color space for devices that support YUV.

You can determine your device's default color space using this code: `vid.ReturnedColorSpace`, where `vid` is the name of the video object. An example of this is shown in step 2 in the example below. There may be situations when you wish to change the color space. The example below shows a case where the default color space is `rgb`, and you change it to `grayscale` (step 3).

The following example illustrates how to specify the color space of the returned image data.

1 **Create an image acquisition object** — This example creates a video input object for a generic Windows image acquisition device. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

```
vid = videoinput('winvideo',1);
```

2 **View the default color space used for the data** — The value of the `ReturnedColorSpace` property indicates the color space of the image data.

```
vid.ReturnedColorSpace

ans =

rgb
```

3 **Modify the color space used for the data** — To change the color space of the returned image data, set the value of the `ReturnedColorSpace` property.

```
vid.ReturnedColorSpace = 'grayscale'

ans =

grayscale
```

4 **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

```
delete(vid)
clear vid
```

Converting Bayer Images

You can use the `ReturnedColorSpace` and `BayerSensorAlignment` properties to control Bayer demosaicing.

If your camera uses Bayer filtering, the toolbox supports the Bayer pattern and can return color if desired. By setting the `ReturnedColorSpace` property to `'bayer'`, the Image Acquisition Toolbox software will demosaic Bayer patterns returned by the hardware. This color space setting will interpolate Bayer pattern encoded images into standard RGB images.

In order to perform the demosaicing, the toolbox needs to know the pixel alignment of the sensor. This is the order of the red, green, and blue sensors and is normally specified by describing the four pixels in the upper-left corner of the sensor. It is the band sensitivity alignment of the pixels as interpreted by the camera's internal hardware. You must get this information from the camera's documentation and then specify the value for the alignment.

If your camera can return Bayer data, the toolbox can automatically convert it to RGB data for you, or you can specify it to do so. The following two examples illustrate both use cases.

Manual Conversion

The camera in this example has a Bayer sensor. The GigE Vision standard allows cameras to inform applications that the data is Bayer encoded and provides enough information for the application to convert the Bayer pattern into a color image. In this case the toolbox automatically converts the Bayer pattern into an RGB image.

1 Create a video object `vid` using the GigE Vision adaptor and the designated video format.

```
vid = videoinput('gige', 1, 'BayerGB8_640x480');
```

2 View the default color space used for the data.

```
vid.ReturnedColorSpace

ans =

rgb
```

3 Create a one-frame image `img` using the `getsnapshot` function.

```
img = getsnapshot(vid);
```

4 View the size of the acquired image.

```
size(img)

ans =

480  640  3
```

5 Sometimes you might not want the toolbox to automatically convert the Bayer pattern into a color image. For example, there are a number of different algorithms to convert from a Bayer pattern into an RGB image and you might wish to specify a different one than the toolbox uses or you might want to further process the raw data before converting it into a color image.

```
% Set the color space to grayscale.
vid.ReturnedColorSpace = 'grayscale';

% Acquire another image frame.
img = getsnapshot(vid);

% Now check the size of the new frame acquired using grayscale.
size(img)

ans =

480  640
```

Notice how the size changed from the `rgb` image to the `grayscale` image by comparing the `size` output in steps 4 and 5.

6 You can optionally use the `demosaic` function in the Image Processing Toolbox to convert Bayer patterns into color images.

```
% Create an image colorImage by using the demosaic function on the
% image img and convert it to color.
colorImage = demosaic(img, 'gbrg');

% Now check the size of the new color image.
size(colorImage)

ans =

480  640  3
```

7 Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

```
delete(vid)
clear vid
```

Automatic Conversion

The camera in this example returns data that is a Bayer mosaic, but the toolbox doesn't know it since the DCAM standard doesn't have any way for the camera to communicate that to software applications. You need to know that by reading the camera specifications or manual. The toolbox can automatically convert the Bayer encoded data to RGB data, but it must be programmed to do so.

1 Create a video object `vid` using the DCAM adaptor and the designated video format for raw data.

```
vid = videoinput('dcam', 1, 'F7_RAW8_640x480');
```

2 View the default color space used for the data.

```
vid.ReturnedColorSpace

ans =

grayscale
```

3 Create a one-frame image `img` using the `getsnapshot` function.

```
img = getsnapshot(vid);
```

4 View the size of the acquired image.

```
size(img)

ans =

480  640
```

5 The value of the `ReturnedColorSpace` property is `grayscale` because Bayer data is single-banded and the toolbox doesn't yet know that it needs to decode the data. Setting the `ReturnedColorSpace` property to `'bayer'` indicates that the toolbox should decode the data.

```
% Set the color space to Bayer.
vid.ReturnedColorSpace = 'bayer';
```

6 In order to properly decode the data, the toolbox also needs to know the alignment of the Bayer filter array. This should be in the camera documentation. You can then use the `BayerSensorAlignment` property to set the alignment.

```
% Set the alignment.
vid.BayerSensorAlignment = 'grbg';
```

The `getdata` and `getsnapshot` functions will now return color data.

```
% Acquire another image frame.
img = getsnapshot(vid);

% Now check the size of the new frame acquired returning color data.
size(img)

ans =

480  640  3
```

Remove the image acquisition object from memory.

```
delete(vid)
clear vid
```

Retrieving Timing Information

Introduction

The following sections describe how the toolbox provides acquisition timing information, particularly,

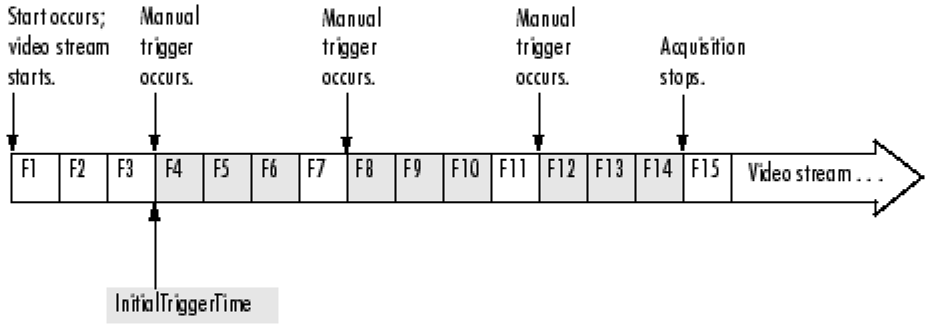
- Determining when a trigger executed on page 7-18
- Determining when a particular frame was acquired on page 7-19

To see an example of retrieving timing information, see "Determining the Frame Delay Duration" on page 7-19.

Determining When a Trigger Executed

To determine when a trigger executed, check the information returned by a trigger event in the object's event log. You can also get access to this information in a callback function associated with a trigger event. For more information, see "Retrieving Event Information" on page 8-7.

As a convenience, the toolbox returns the time of the *first* trigger execution in the video input object's `InitialTriggerTime` property. This figure indicates which trigger is returned in this property when multiple triggers are configured.



InitialTriggerTime Records First Trigger Execution

The trigger timing information is stored in MATLAB clock vector format. The following example displays the time of the first trigger for the video input object `vid`. The example uses the MATLAB `datestr` function to convert the information into a form that is more convenient to view.

```
datestr(datetime(vid.InitialTriggerTime))

ans =

02-Mar-2007 13:00:24
```

Determining When a Frame Was Acquired

The toolbox provides two ways to determine when a particular frame was acquired:

- By the absolute time of the acquisition
- By the elapsed time relative to the execution of the trigger

You can use the `getdata` function to retrieve both types of timing information.

Getting the Relative Acquisition Time

When you use the `getdata` function, you can optionally specify two return values. One return value contains the image data; the other return value contains a vector of timestamps that measure, in seconds, the time when the frame was acquired relative to the first trigger.

```
[data time] = getdata(vid);
```

To see an example, see "Determining the Frame Delay Duration" on page 7-19.

Getting the Absolute Acquisition Time

When you use the `getdata` function, you can optionally specify three return values. The first contains the image data, the second contains a vector of relative acquisition times, and the third is an array of structures where each structure contains metadata associated with a particular frame.

```
[data time meta ] = getdata(vid);
```

Each structure in the array contains the following four fields. The `AbsTime` field contains the absolute time the frame was acquired. You can also retrieve this metadata by using event callbacks. See "Retrieving Event Information" on page 8-7 for more information.

Frame Metadata

Field Name	Description
`AbsTime`	Absolute time the frame was acquired, returned in MATLAB clock format `[year month day hour minute seconds]`
`FrameNumber`	Frame number relative to when the object was started
`RelativeFrame`	Frame number relative to trigger execution
`TriggerIndex`	Trigger the event is associated with. For example, when the object starts, the associated trigger is 0. Upon stop, it is equivalent to the `TriggersExecuted` property.

Determining the Frame Delay Duration

To illustrate, this example calculates the duration of the delay specified by the `TriggerFrameDelay` property.

1 **Create an image acquisition object** — This example creates a video input object for a Data Translation image acquisition device using the default video format. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

```
vid = videoinput('dt',1);
```

2 **Configure properties** — For this example, configure a trigger frame delay large enough to produce a noticeable duration.

```
vid.TriggerFrameDelay = 50
```

3 **Start the image acquisition object** — Call the `start` function to start the image acquisition object.

```
start(vid)
```

The object executes an immediate trigger and begins acquiring frames of data. The `start` function returns control to the command line immediately but data logging does not begin until the trigger frame delay expires. After logging the specified number of frames, the object stops running.

4 **Bring the acquired data into the workspace** — Call the `getdata` function to bring frames into the workspace. Specify a return value to accept the timing information returned by `getdata`.

```
[data time ] = getdata(vid);
```

The variable `time` is a vector that contains the time each frame was logged, measured in seconds, relative to the execution of the first trigger. Check the first value in the time vector. It should reflect the duration of the delay before data logging started.

```
time

time =

    4.9987
    5.1587
    5.3188
    5.4465
    5.6065
    5.7665
    5.8945
    6.0544
    6.2143
    6.3424
```

5 **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

```
delete(vid)
clear vid
```

Using Events and Callbacks

Using Events and Callbacks

You can enhance the power and flexibility of your image acquisition application by using *event callbacks*. An event is a specific occurrence that can happen while an image acquisition object is running. The toolbox defines a set of events that include starting, stopping, or acquiring frames of data.

When a particular event occurs, the toolbox can execute a function that you specify. This is called a *callback*. Certain events can result in one or more callbacks. You can use callbacks to perform processing tasks while your image acquisition object continues running. For example, you can display a message, analyze data, or perform other tasks. The start and stop callbacks, however, execute synchronously; the object does not perform any further processing until the callback function finishes.

Callbacks are controlled through video input object properties. Each event type has an associated property. You specify the function that you want executed as the value of the property.

The following topics describe using events and callbacks.

- "Using the Default Callback Function" on page 8-3
- "Event Types" on page 8-4
- "Retrieving Event Information" on page 8-7
- "Creating and Executing Callback Functions" on page 8-10

Using the Default Callback Function

To illustrate how to use callbacks, this section presents a simple example that creates an image acquisition object and associates a callback function with the start event, trigger event, and stop event. For information about all the event callbacks supported by the toolbox, see “Event Types” on page 8-4.

The example uses the default callback function provided with the toolbox, `imaqcallback`. The default callback function displays the name of the object along with information about the type of event that occurred and when it occurred. To learn how to create your own callback functions, see “Creating and Executing Callback Functions” on page 8-10.

This example illustrates how to use the default callback function.

1. **Create an image acquisition object** — This example creates a video input object for a Matrox image acquisition device. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

   ```
   vid = videoinput('matrox',1);
   ```
2. **Configure properties** — Set the values of three callback properties. The example uses the default callback function `imaqcallback`.

   ```
   vid.StartFcn = @imaqcallback
   vid.TriggerFcn = @imaqcallback
   vid.StopFcn = @imaqcallback
   ```

   For this example, specify the amount of data to log.

   ```
   vid.FramesPerTrigger = 100;
   ```
3. **Start the image acquisition object** — Start the image acquisition object. The object executes an immediate trigger, acquires 100 frames of data, and then stops. With the three callback functions enabled, the object outputs information about each event as it occurs.

   ```
   start(vid)
   Start event occurred at 14:38:46 for video input object: M_RS170-matrox-1.
   Trigger event occurred at 14:38:46 for video input object: M_RS170-matrox-1.
   Stop event occurred at 14:38:49 for video input object: M_RS170-matrox-1.
   ```
4. **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

   ```
   delete(vid)
   clear vid
   ```

Event Types

The Image Acquisition Toolbox software supports several different types of events. Each event type has an associated video input object property that you can use to specify the function that executes when the event occurs.

This table lists the supported event types, the name of the video input object property associated with the event, and a brief description of the event. For detailed information about these callback properties, see the property reference list in “Image Acquisition Toolbox Properties” on page 5-31.

The toolbox generates a specific set of information for each event and stores it in an event structure. To learn more about the contents of these event structures and how to retrieve this information, see “Retrieving Event Information” on page 8-7.

Note Callbacks, including `ErrorFcn`, are executed only when the video object is in a running state. If you need to use the `ErrorFcn` callback for error handling during previewing, you must start the video object before previewing. To do that without logging data, use a manual trigger.

Events and Callback Function Properties

Event	Callback Property	Description
Error	`ErrorFcn`	The toolbox generates an error event when a run-time error occurs, such as a hardware error or timeout. Run-time errors do not include configuration errors such as setting an invalid property value. When an error event occurs, the toolbox executes the function specified by the `ErrorFcn` property. By default, the toolbox executes the default callback function for this event, `imaqcallback`, which displays the error message at the MATLAB command line.
Frames Acquired	`FramesAcquiredFcn`	The toolbox generates a frames acquired event when a specified number of frames have been acquired. You use the `FramesAcquiredFcnCount` property to specify this number. When a frames acquired event occurs, the toolbox executes the function specified by the `FramesAcquiredFcn` property.
Start	`StartFcn`	The toolbox generates a start event when an object is started. You use the `start` function to start an object. When a start event occurs, the toolbox executes the function specified by the `StartFcn` property. **Note** The `StartFcn` callback executes synchronously. If you specify a `StartFcn` callback function, the toolbox waits for the function to finish executing before performing any other processing. If an error occurs in the start callback function, the object never starts.
Stop	`StopFcn`	The toolbox generates a stop event when the object stops running. An object stops running when the `stop` function is called, the specified number of frames is acquired, or a run-time error occurs. When a stop event occurs, the toolbox executes the function specified by the `StopFcn` property. **Note** The `StopFcn` callback executes synchronously. If you specify a `StopFcn` callback function, the toolbox waits for the function to finish executing before performing any other processing.

Event	Callback Property	Description
Timer	`TimerFcn`	The toolbox generates a timer event when a specified amount of time expires. Time is measured relative to when the object starts running. You use the `TimerPeriod` property to specify the amount of time. **Note** Some timer events might not execute if your system is significantly slowed or if the `TimerPeriod` is set too small. When a timer event occurs, the toolbox executes the function specified by the `TimerFcn` property.
Trigger	`TriggerFcn`	The toolbox generates a trigger event when a trigger executes. The video input object executes immediate triggers. You execute manual triggers by calling the `trigger` function. The image acquisition device executes hardware triggers when a specified condition is met. When a trigger event occurs, the toolbox executes the function specified by the `TriggerFcn` property.

Retrieving Event Information

In this section...
“Introduction” on page 8-7
“Event Structures” on page 8-7
“Accessing Data in the Event Log” on page 8-8

Introduction

Each event has associated with it a set of information, generated by the toolbox and stored in an event structure. This information includes the event type, the time the event occurred, and other event-specific information. While a video input object is running, the toolbox records event information in the object's `EventLog` property. You can also access the event structure associated with an event in a callback function.

This section

- Defines the information in an event structure on page 8-7 for all event types
- Describes how to retrieve information on page 8-8 from the `EventLog` property

For information about accessing event information in a callback function, see “Creating and Executing Callback Functions” on page 8-10.

Event Structures

An event structure contains two fields: `Type` and `Data`. For example, this is an event structure for a trigger event:

```
Type: 'Trigger'
Data: [1x1 struct]
```

The `Type` field is a character vector that specifies the event type. For a trigger event, this field contains the character vector `'Trigger'`.

The `Data` field is a structure that contains information about the event. The composition of this structure varies depending on which type of event occurred. For details about the information associated with specific events, see the following sections:

- “Data Fields for Start, Stop, Frames Acquired, and Trigger Events” on page 8-7
- “Data Fields for Error Events” on page 8-8
- “Data Fields for Timer Events” on page 8-8

Data Fields for Start, Stop, Frames Acquired, and Trigger Events

For start, stop, frames acquired, and trigger events, the `Data` structure contains these fields.

Field Name	Description
`AbsTime`	Absolute time the event occurred, returned in MATLAB clock format `[year month day hour minute seconds]`

Field Name	Description
`FrameNumber`	Frame number relative to when the object was started
`RelativeFrame`	Frame number relative to the execution of a trigger
`TriggerIndex`	Trigger the event is associated with. For example, upon start, the associated trigger is 0. Upon stop, it is equivalent to the `TriggersExecuted` property.

Data Fields for Error Events

For error events, the `Data` structure contains these fields.

Field Name	Description
`AbsTime`	Absolute time the event occurred, returned in MATLAB clock format `[year month day hour minute seconds]`
`Message`	Text message associated with the error
`MessageID`	MATLAB message identifier associated with the error

Data Fields for Timer Events

For timer events, the `Data` structure contains these fields.

Field Name	Description
`AbsTime`	Absolute time the event occurred, returned in MATLAB clock format `[year month day hour minute seconds]`

Accessing Data in the Event Log

While a video input object is running, the toolbox stores event information in the object's `EventLog` property. The value of this property is an array of event structures. Each structure represents one event. For detailed information about the composition of an event structure for each type of event, see “Event Structures” on page 8-7.

The toolbox adds event structures to the `EventLog` array in the order in which the events occur. The first event structure reflects the first event recorded, the second event structure reflects the second event recorded, and so on.

Note Only start, stop, error, and trigger events are recorded in the `EventLog` property. Frames-acquired events and timer events are not included in the `EventLog`. Event structures for these events (and all the other events) are available to callback functions. For more information, see “Creating and Executing Callback Functions” on page 8-10.

To illustrate the event log, this example creates a video input object, runs it, and then examines the object's `EventLog` property:

1 **Create an image acquisition object** — This example creates a video input object for a Matrox image acquisition device. To run this example on your system, use the `imaqhwinfo` function to

get the object constructor for your image acquisition device and substitute that syntax for the following code.

```
vid = videoinput('matrox',1);
```

2 **Start the image acquisition object** — Start the image acquisition object. By default, the object executes an immediate trigger, acquires 10 frames of data, and then stops.

```
start(vid)
```

3 **View the event log** — Access the `EventLog` property of the video input object. The execution of the video input object generated three events: start, trigger, and stop. Thus the value of the `EventLog` property is a 1x3 array of event structures.

```
events = vid.EventLog
events =

1x3 struct array with fields:
    Type
    Data
```

To list the events that are recorded in the `EventLog` property, examine the contents of the `Type` field.

```
{events.Type}
ans =
    'Start'    'Trigger'    'Stop'
```

To get information about a particular event, access the `Data` field in that event structure. The example retrieves information about the trigger event.

```
trigdata = events(2).Data

trigdata = 

             AbsTime: [2004 12 29 16 40 52.5990]
         FrameNumber: 0
       RelativeFrame: 0
        TriggerIndex: 1
```

4 **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

```
delete(vid)
clear vid
```

Creating and Executing Callback Functions

In this section...

Introduction

The power of using event callbacks is the processing that you can perform in response to events. You decide which events you want to associate callbacks with and the functions these callbacks execute.

This section

- Describes how to create a callback function on page 8-10
- Describes how to specify the function on page 8-11 as the value of a callback property
- Provides two examples of using event callbacks:
  - Shows how to use callbacks to view a sample frame on page 8-13 from the frames being acquired

Note Callback function execution might be delayed if the callback involves a CPU-intensive task such as updating a figure.

Creating Callback Functions

This section explains how to create callback functions for the `TimerFcn`, `FramesAcquiredFcn`, `StartFcn`, `StopFcn`, `TriggerFcn`, and `ErrorFcn` callbacks.

Callback functions require at least two input arguments:

- The image acquisition object
- The event structure associated with the event

The function header for this callback function illustrates this basic syntax.

```
function mycallback(obj,event)
```

The first argument, `obj`, is the image acquisition object itself. Because the object is available, you can use in your callback function any of the toolbox functions, such as `getdata`, that require the object as an argument. You can also access all object properties.

The second argument, `event`, is the event structure associated with the event. This event information pertains only to the event that caused the callback function to execute. For a complete list of supported event types and their associated event structures, see "Event Structures" on page 8-7.

In addition to these two required input arguments, you can also specify additional, application-specific arguments for your callback function.

Note To receive the object and event arguments, and any additional arguments, you must use a cell array when specifying the name of the function as the value of a callback property. For more information, see "Specifying Callback Functions" on page 8-11.

Writing a Callback Function

To illustrate, this example implements a callback function for a frames-acquired event. This callback function enables you to monitor the frames being acquired by viewing a sample frame periodically.

To implement this function, the callback function acquires a single frame of data and displays the acquired frame in a MATLAB figure window. The function also accesses the event structure passed as an argument to display the timestamp of the frame being displayed. The `drawnow` command in the callback function forces MATLAB to update the display.

```
function display_frame(obj,event)

sample_frame = peekdata(obj,1);

imagesc(sample_frame);

drawnow; % force an update of the figure window

abstime = event.Data.AbsTime;

t = fix(abstime);

sprintf('%s %d:%d:%d','timestamp', t(4),t(5),t(6))
```

To see how this function can be used as a callback, see "Viewing a Sample Frame" on page 8-13.

Specifying Callback Functions

You associate a callback function with a specific event by setting the value of the event's callback property. The video input object supports callback properties for all types of events.

You can specify the callback function as the value of the property in any of three ways:

- Character vector on page 8-11
- Cell array on page 8-12
- Function handle on page 8-12

The following sections provide more information about each of these options.

Note To access the object or event structure passed to the callback function, you must specify the function as a cell array or as a function handle.

Using a Character Vector to Specify Callback Functions

You can specify the callback function as a character vector . For example, this code specifies the callback function `mycallback` as the value of the start event callback property `StartFcn` for the video input object `vid`.

```
vid.StartFcn = 'mycallback';
```

In this case, the callback is evaluated in the MATLAB workspace.

Using a Cell Array to Specify Callback Functions

You can specify the callback function as a character vector inside a cell array.

For example, this code specifies the callback function `mycallback` as the value of the start event callback property `StartFcn` for the video input object `vid`.

```
vid.StartFcn = {'mycallback'};
```

To specify additional parameters, include them as additional elements in the cell array.

```
time = datestr(datetime('now'),0);
vid.StartFcn = {'mycallback',time};
```

The first two arguments passed to the callback function are still the video input object (`obj`) and the event structure (`event`). Additional arguments follow these two arguments.

Using Function Handles to Specify Callback Functions

You can specify the callback function as a function handle.

For example, this code specifies the callback function `mycallback` as the value of the start event callback property `StartFcn` for the video input object `vid`.

```
vid.StartFcn = @mycallback;
```

To specify additional parameters, include the function handle and the parameters as elements in the cell array.

```
time = datestr(datetime('now'),0);
vid.StartFcn = {@mycallback,time};
```

If you are executing a local callback function from within a MATLAB file, you must specify the callback as a function handle.

Specifying a Toolbox Function as a Callback

In addition to specifying callback functions of your own creation, you can also specify the `start`, `stop`, or `trigger` toolbox functions as callbacks. For example, this code sets the value of the stop event callback to Image Acquisition Toolbox `start` function.

```
vid.StopFcn = @start;
```

Disabling Callbacks

If an error occurs in the execution of the callback function, the toolbox disables the callback and displays a message similar to the following.

```
start(vid)
??? Error using ==> frames_cb
Too many input arguments.

Warning: The FramesAcquiredFcn callback is being disabled.
```

To enable a callback that has been disabled, set the value of the property associated with the callback or restart the object.

Viewing a Sample Frame

This example creates a video input object and sets the frames acquired event callback function property to the `display_frame` function, created in “Writing a Callback Function” on page 8-11.

The example sets the `TriggerRepeat` property of the object to 4 so that 50 frames are acquired. When run, the example displays a sample frame from the acquired data every time five frames have been acquired.

1. **Create an image acquisition object** — This example creates a video input object for a Matrox image acquisition device. To run this example on your system, use the `imaqhwinfo` function to get the object constructor for your image acquisition device and substitute that syntax for the following code.

   ```
   vid = videoinput('matrox', 1);
   ```
2. **Configure property values** — This example sets the `FramesPerTrigger` value to 30 and the `TriggerRepeat` property to 4. The example also specifies as the value of the `FramesAcquiredFcn` callback the event callback function `display_frame`, created in “Writing a Callback Function” on page 8-11. The object will execute the `FramesAcquiredFcn` every five frames, as specified by the value of the `FramesAcquiredFcnCount` property.

   ```
   vid.FramesPerTrigger = 30;
   vid.TriggerRepeat = 4;
   vid.FramesAcquiredFcnCount = 5;
   vid.FramesAcquiredFcn = {'display_frame'};
   ```
3. **Acquire data** — Start the video input object. Every time five frames are acquired, the object executes the `display_frame` callback function. This callback function displays the most recently acquired frame logged to the memory buffer.

   ```
   start(vid)
   ```
4. **Clean up** — Always remove image acquisition objects from memory, and the variables that reference them, when you no longer need them.

   ```
   delete(vid)
   clear vid
   ```

Using the From Video Device Block in Simulink

The Image Acquisition Toolbox software includes a block that can be used in Simulink to bring live video data into models.

Open Image Acquisition Toolbox Block Library

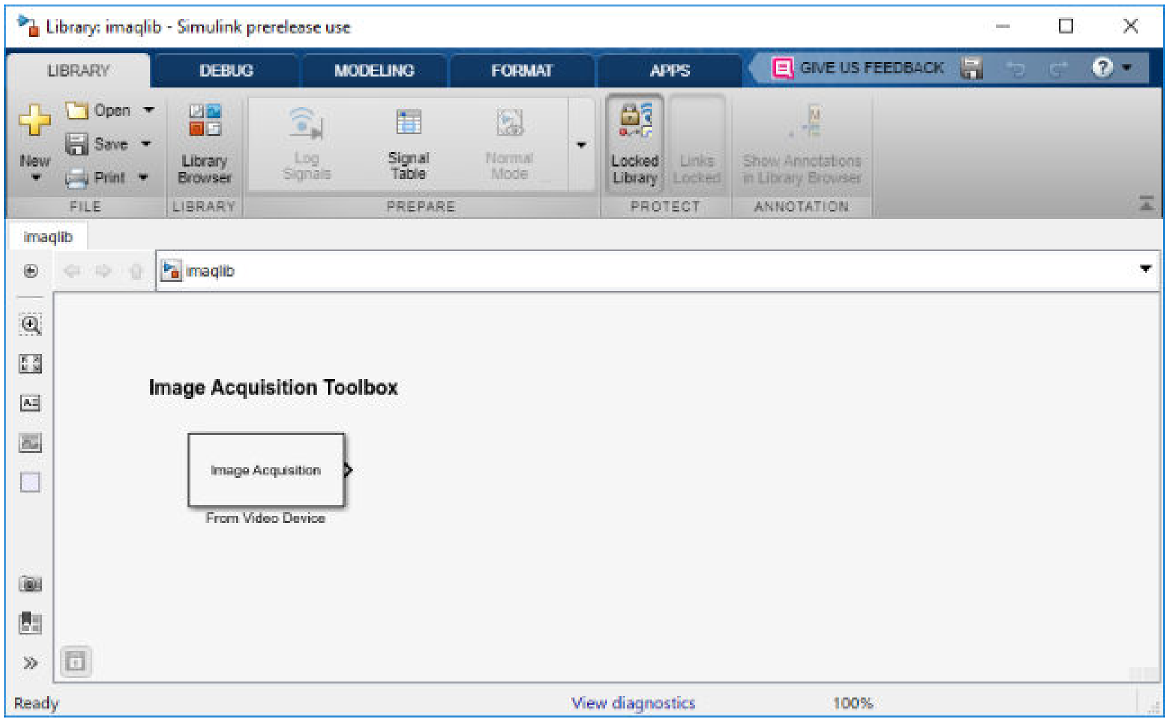
In this section...
"From the Command Line" on page 9-2
"From the Simulink Library Browser" on page 9-2

From the Command Line

To open the Image Acquisition Toolbox block library, enter the following in the MATLAB Command Window.

```
imaqlib
```

MATLAB displays the contents of the library in a separate window.



From the Simulink Library Browser

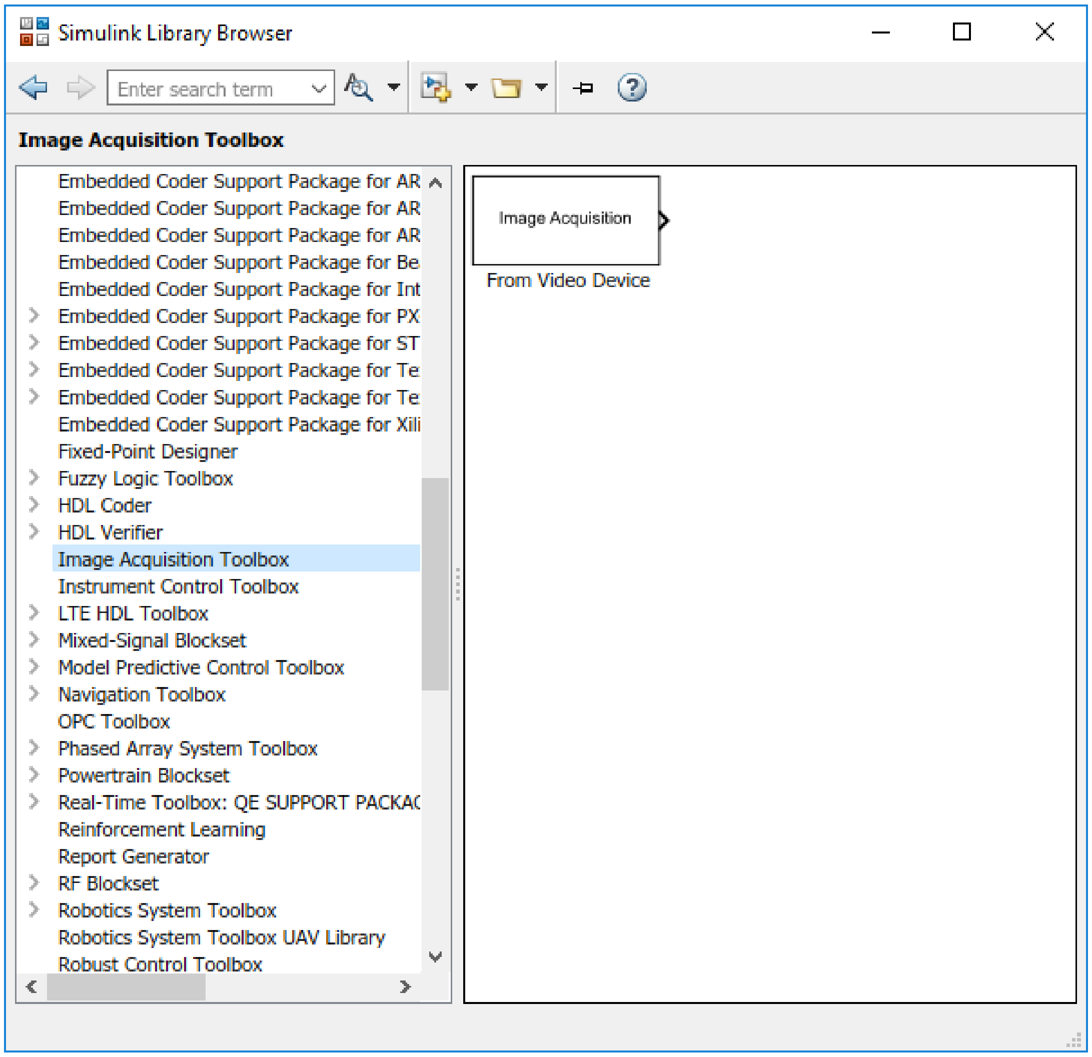
To open the Image Acquisition Toolbox block library, start the Simulink Library Browser by entering the following in the MATLAB Command Window.

```
simulink
```

On the Simulink start page, click **Blank Model** and then **Create Model**. An empty Editor window opens.

On the toolstrip, click **Library Browser** on the **Simulation** tab.

The Simulink Library Browser opens. Its left pane contains a tree of available block libraries in alphabetical order. Click the `Image Acquisition Toolbox` node.



Embedded Coder Support Package for AR
Embedded Coder Support Package for AR
Embedded Coder Support Package for AR
Embedded Coder Support Package for Be
Embedded Coder Support Package for Int
Embedded Coder Support Package for PX
Embedded Coder Support Package for ST
Embedded Coder Support Package for Te
Embedded Coder Support Package for Te
Embedded Coder Support Package for Xili
Fixed-Point Designer
Fuzzy Logic Toolbox
HDL Coder
HDL Verifier
Image Acquisition Toolbox
Instrument Control Toolbox
LTE HDL Toolbox
Mixed-Signal Blockset
Model Predictive Control Toolbox
Navigation Toolbox
OPC Toolbox
Phased Array System Toolbox
Powertrain Blockset
Real-Time Toolbox: QE SUPPORT PACKAC
Reinforcement Learning
Report Generator
RF Blockset
Robotics System Toolbox
Robotics System Toolbox UAV Library
Robust Control Toolbox

To use a block, add it to an existing model or create a new model.

See Also

More About

- “Save Video Data to a File” on page 9-6

Code Generation with From Video Device Block

Code Generation Workflow

The From Video Device block supports code generation with Simulink Coder. Generating code from the From Video Device block enables you to run models containing the block in Accelerator, Rapid Accelerator, and Deployed modes.

A typical workflow for code generation follows.

1. Develop a model using the From Video Device block and sink blocks from other toolboxes, such as the Computer Vision Toolbox™.
2. Run the simulation to verify that your device is working.
3. Build the model to generate code and create the executable.

The deployed application can then be used on a machine that does not have MATLAB and Simulink.

Code Generation with Simulink Coder

You can use Image Acquisition Toolbox, Simulink Coder, and Embedded Coder® together to generate code (on the host end) that you can use to implement your model for a practical application. For more information on code generation, see the Simulink Coder documentation.

Note If you use a GigE Vision camera with the From Video Device block, you must install GenICam to use the generated application outside of MATLAB. After you install the GenICam driver, load the DLL files by manually adding the path to the DLL files to the system path (in **Control Panel > System > Advanced system settings > Environment Variables...**) .

Shared Library Dependencies

The From Video Device block generates code with limited portability. The block uses precompiled shared libraries, such as DLLs, to support I/O for specific types of devices. Simulink Coder provides functions to help you set up and manage the build information for your models. One of the build information functions that Simulink Coder provides is `packNGo`. This function allows you to package model code and dependent shared libraries into a zip file for deployment. The target system does not need to have MATLAB installed but it does need to be supported by MATLAB.

The block supports use of the `packNGo` function. Source-specific properties for your device are honored when code is generated. The generated code compiles with both C and C++ compilers.

To set up `packNGo`, run the following code in the MATLAB Command Window.

```
set_param(gcs,'PostCodeGenCommand','packNGo(buildInfo)');
```

In this example, `gcs` is the current model that you want to build. Building the model creates a zip file with the same name as the model name. You can move this zip file to another machine and build the source code in the zip file to create an executable that can run independent of MATLAB and Simulink. For more information, see `packNGo`.

Note The From Video Device block supports the use of Simulink Rapid Accelerator mode and code generation on Windows platforms. Code generation is also supported on Linux®, but Rapid Accelerator mode is not.

Note If you get a “Device in use” error message when using the block with certain hardware, such as Matrox, close any programs that are using the hardware, and try using the block again.

Note On Linux platforms, you need to add the directory where you unzip the libraries to the environment variable `LD_LIBRARY_PATH`.

See Also

From Video Device

Save Video Data to a File

In this section...

This example shows how to build a simple model using the From Video Device block in conjunction with blocks from other blockset libraries.

Note Block names are not shown by default in the model. To display the hidden block names while working in the model, select **Display** and clear the **Hide Automatic Names** check box.

Step 1: Create a New Model

To start Simulink and create a new model, enter the following in the MATLAB Command Window.

```
simulink
```

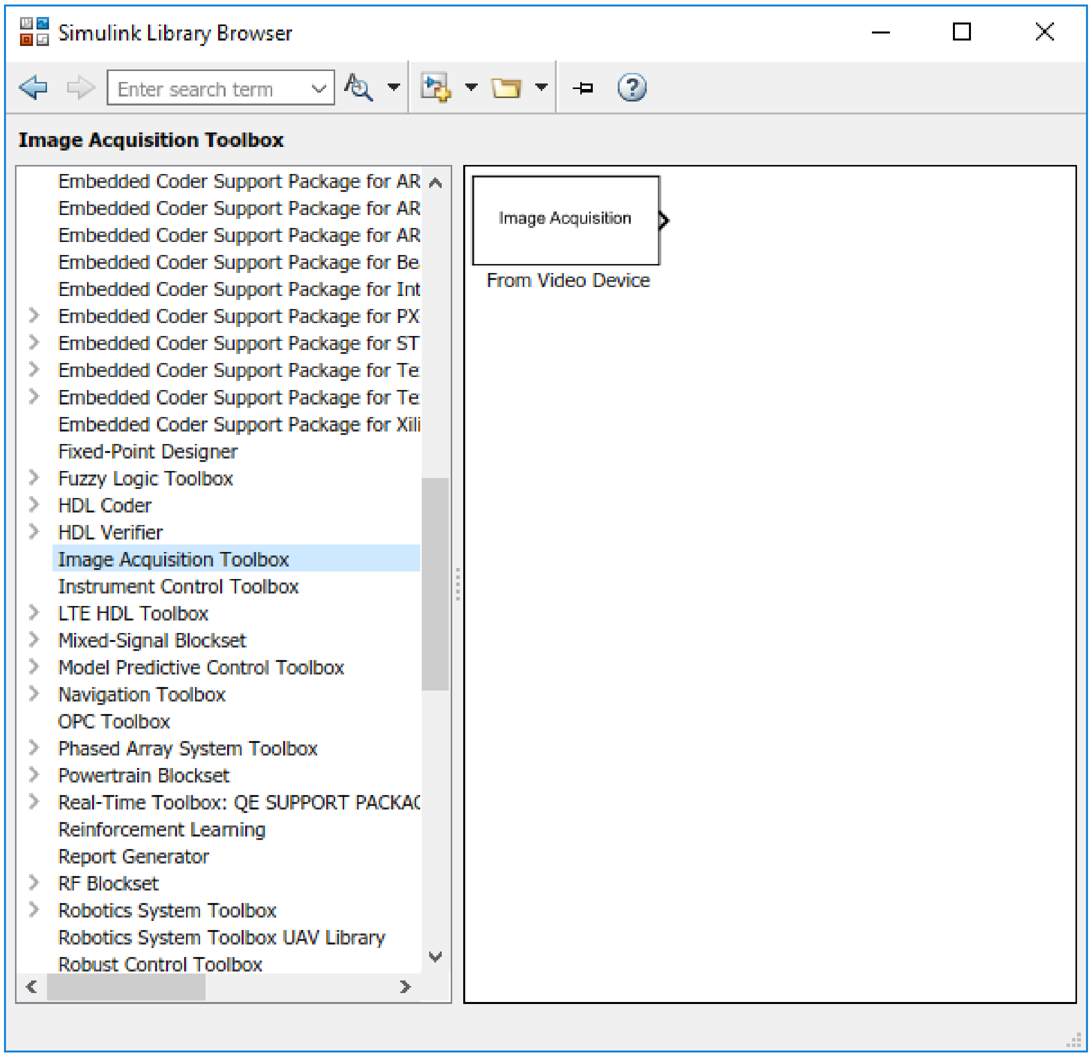
On the Simulink start page, click **Blank Model** and then **Create Model**. An empty Editor window opens.

In the Editor, click **Save** on the **Simulation** tab to assign a name to your new model.

Step 2: Open the Image Acquisition Toolbox Library

In the model Editor window, click the **Library Browser** button on the **Simulation** tab.

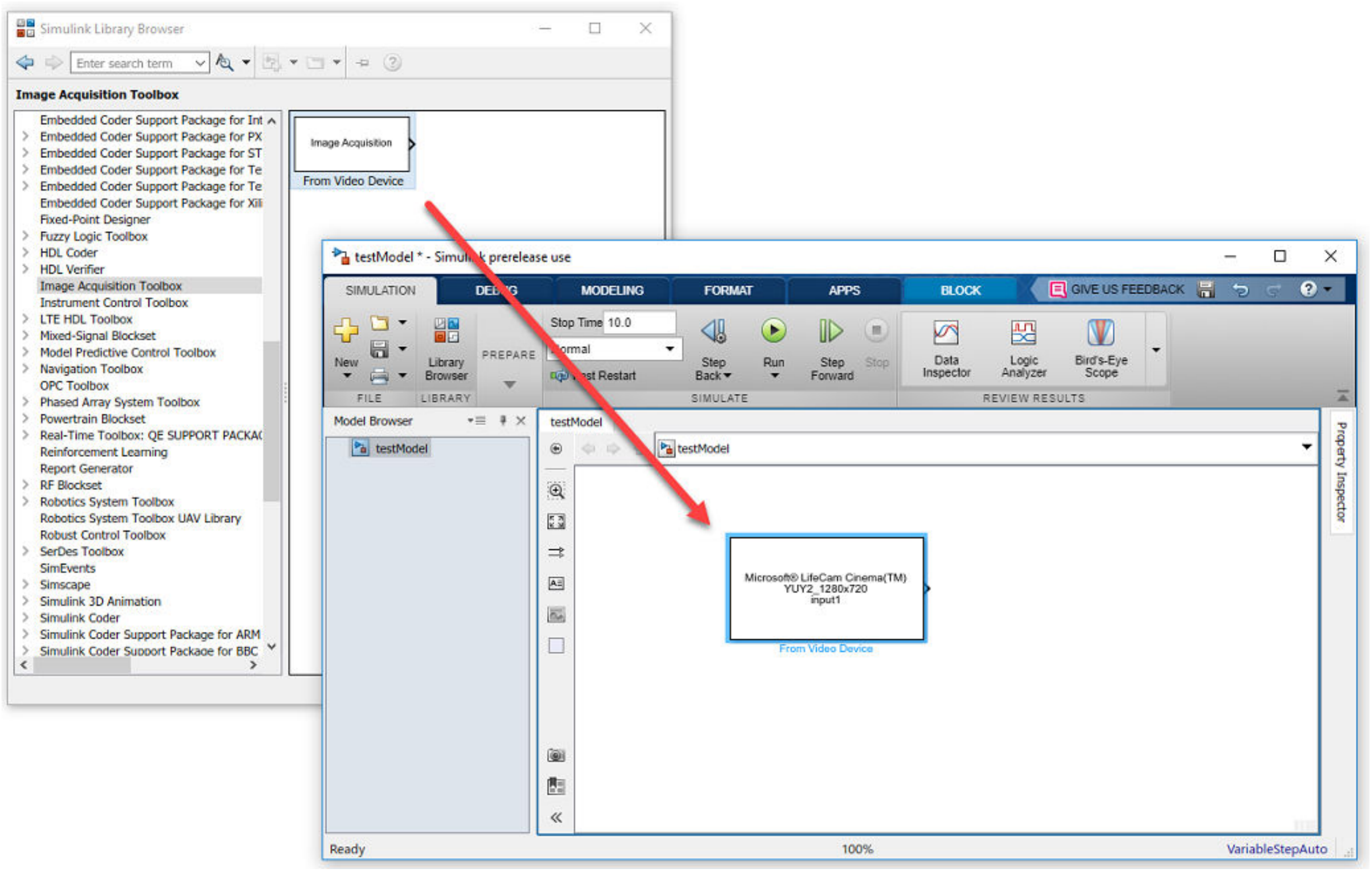
The Simulink Library Browser opens. Its left pane contains a tree of available block libraries in alphabetical order. Click the **Image Acquisition Toolbox** node.


Simulink Library Browser
Enter search term
Image Acquisition Toolbox
Embedded Coder Support Package for AR
Embedded Coder Support Package for AR
Embedded Coder Support Package for AR
Embedded Coder Support Package for Be
Embedded Coder Support Package for Int
Embedded Coder Support Package for PX
Embedded Coder Support Package for ST
Embedded Coder Support Package for Te
Embedded Coder Support Package for Te
Embedded Coder Support Package for Xili
Fixed-Point Designer
Fuzzy Logic Toolbox
HDL Coder
HDL Verifier
Image Acquisition Toolbox
Instrument Control Toolbox
LTE HDL Toolbox
Mixed-Signal Blockset
Model Predictive Control Toolbox
Navigation Toolbox
OPC Toolbox
Phased Array System Toolbox
Powertrain Blockset
Real-Time Toolbox: QE SUPPORT PACKAC
Reinforcement Learning
Report Generator
RF Blockset
Robotics System Toolbox
Robotics System Toolbox UAV Library
Robust Control Toolbox
Image Acquisition
From Video Device

To use a block, add it to an existing model or create a new model.

Step 3: Drag the From Video Device Block into the Model

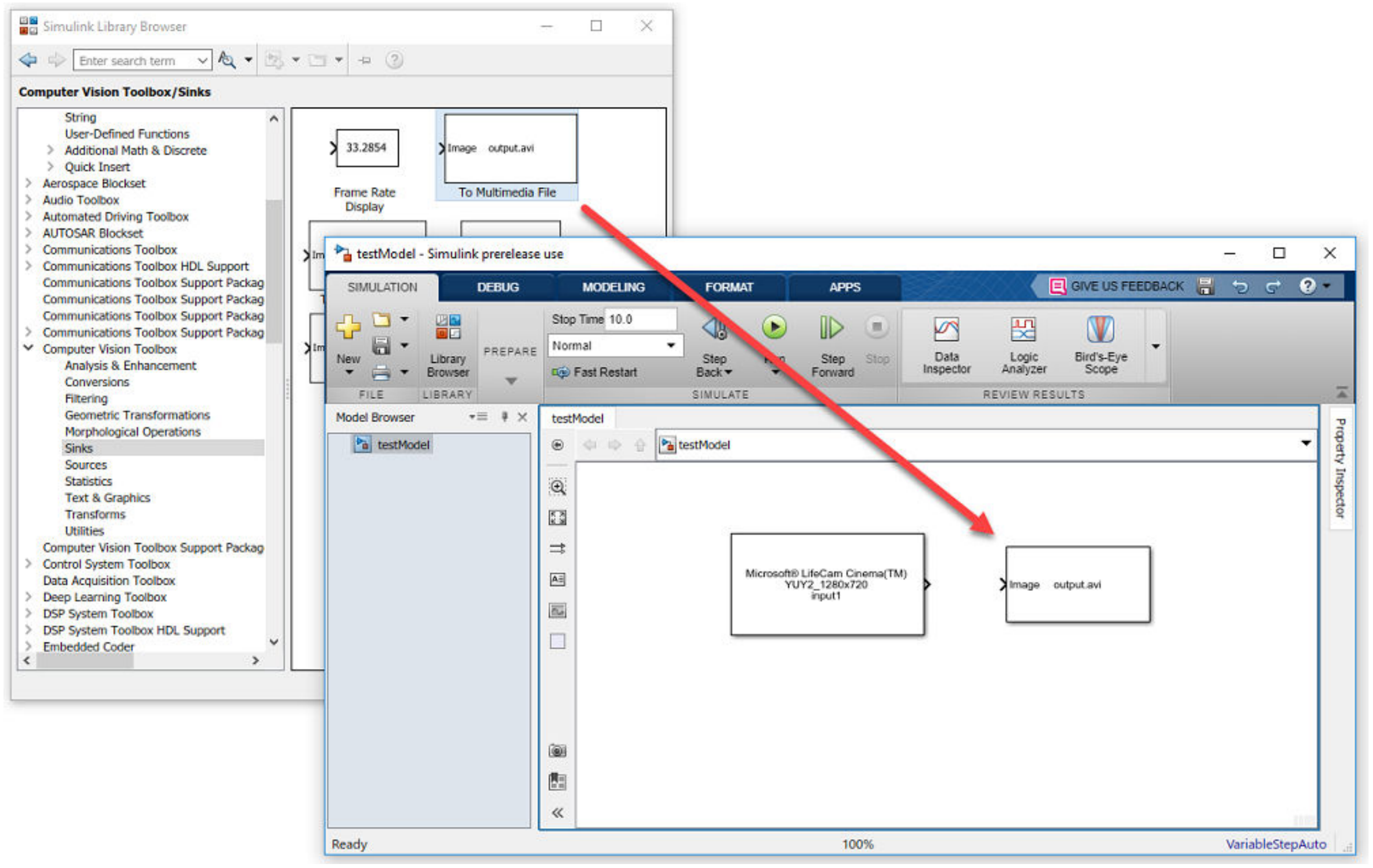
To use the From Video Device block in a model, drag the block into the Simulink Editor. Note how the name on the block changes to reflect the device connected to your system that is associated with the block. If you have multiple devices connected, you can choose the device to use in the parameters dialog box by double-clicking the block.



Step 4: Drag Other Blocks to Complete the Model

To illustrate how to use the block, this example creates a simple model that acquires data and then returns the data to a file in AVI format. To create this model, this example uses a block from Computer Vision Toolbox.

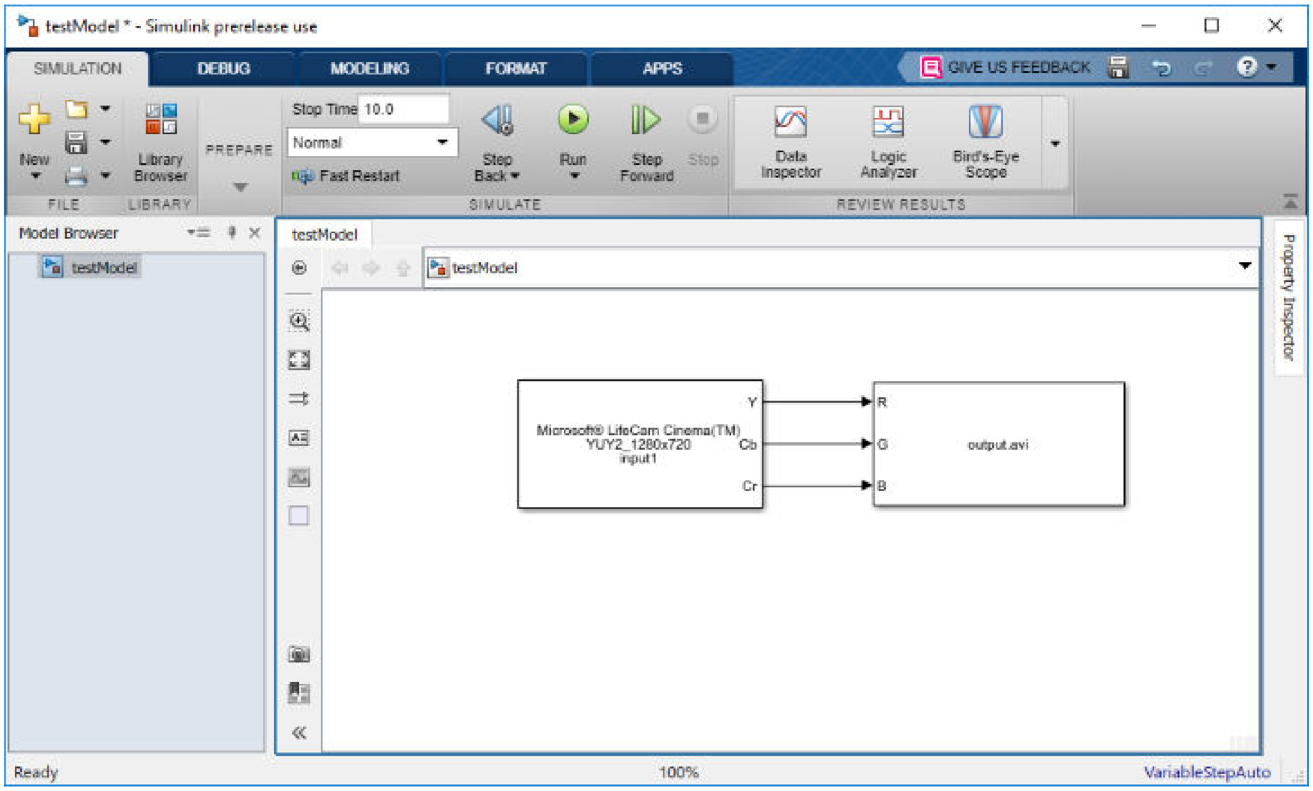
Open the Computer Vision Toolbox library. In the library window, open the **Sinks** subsystem. From this subsystem, click the To Multimedia File block in the library and drag the block into the Simulink Editor.



Step 5: Connect the Blocks

Connect the three outputs from the From Video Device block to the three corresponding inputs on the To Multimedia File block. If the ports are not displayed, you can display them in the parameters dialog box by double-clicking the block. One quick way to make all three connections at once is to select the From Video Device block, press and hold the **Ctrl** key, and then click the To Multimedia File block.

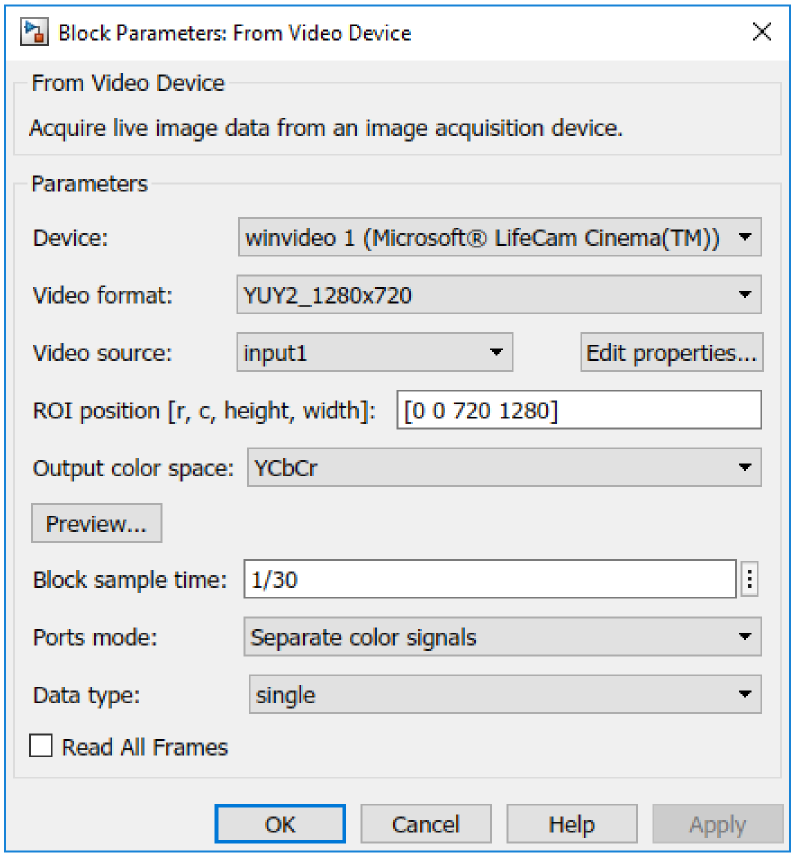
The input ports of the To Multimedia File block are RGB, but you can use a camera that has either YCbCr or RGB output ports.



Step 6: Specify From Video Device Block Parameter Values

To check From Video Device block parameter settings, double-click the block icon in the Simulink Editor. The parameters dialog box for the From Video Device block opens. Use the various fields in the dialog box to set or change the values of the From Video Device block parameters.

For example, using this dialog box, you can specify the device you want to use, select the video format you want to use with the device, or specify the block sample time. For details, see From Video Device.



Acquire live image data from an image acquisition device.

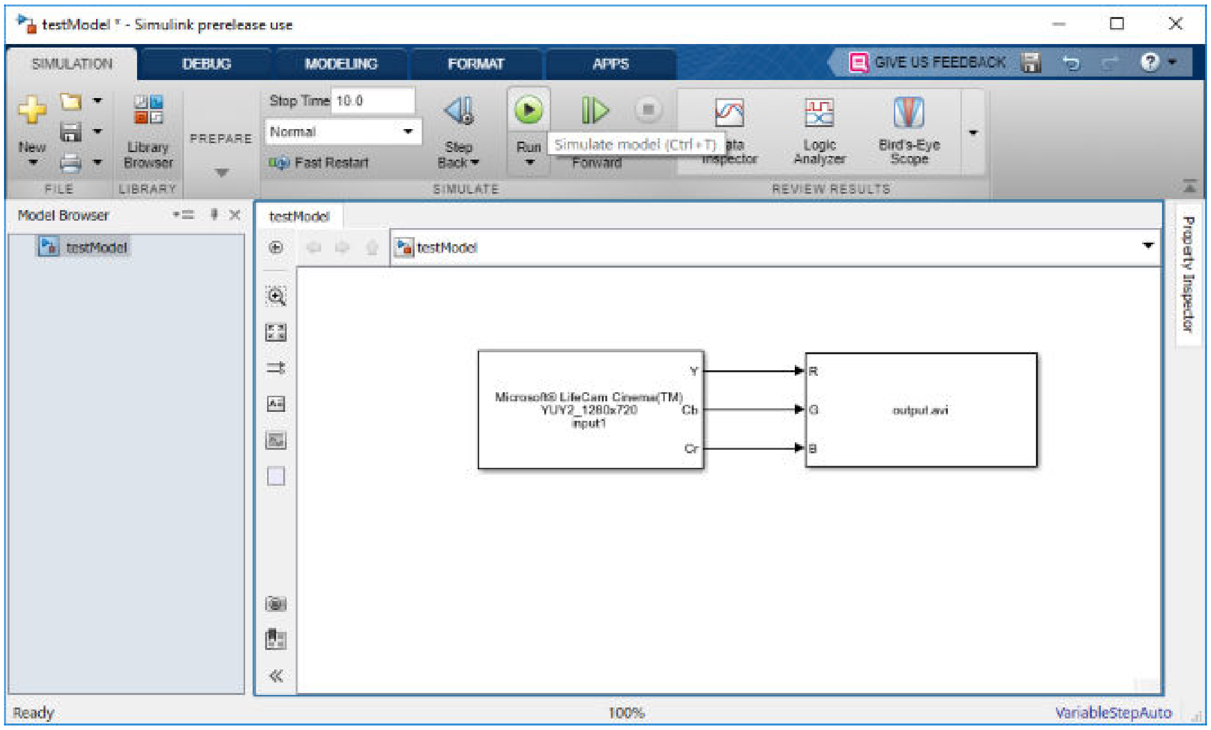
Device: winvideo 1 (Microsoft® LifeCam Cinema(TM))
Video format: YUY2_1280x720
Video source: input1
ROI position [r, c, height, width]: [0 0 720 1280]
Output color space: YCbCr
Block sample time: 1/30
Ports mode: Separate color signals
Data type: single
☐ Read All Frames

You can set parameters for any of the blocks you include in your model. For example, to specify the name of the AVI file, double-click the To Multimedia File block. Make sure that you have write permission to the directory into which the block writes the AVI file.

Step 7: Run the Simulation

To run the simulation, click the green **Run** button on the Simulink Editor toolstrip. You can use toolstrip options to specify how long to run the simulation and to stop it.

While the simulation is running, the status bar at the bottom of the Simulink Editor indicates the progress of the simulation. When the simulation finishes, the software saves an AVI file to the current working directory.



See Also

From Video Device

Configuring GigE Vision Devices

Types of Setups

The Image Acquisition Toolbox software supports GigE Vision devices. The following sections describe information on installing and configuring the devices to work with the Image Acquisition Toolbox software. Separate troubleshooting information is found in “Troubleshooting GigE Vision Devices on Windows” on page 16-19.

Note Not all cameras that use Ethernet are GigE Vision. A camera must have the GigE Vision logo appearing on it or its data sheet to be a GigE Vision device.

There are five different setups you can use for GigE Vision cameras.

- Direct to a PC not on a network — PC is connected to camera with a Cat 5e or 6 Ethernet cable. PC is not on a network. This is one of the setups that offers the best acquisition speed.
- Direct to a PC on a network, using two Ethernet cards — PC is connected to camera with a Cat 5e or 6 Ethernet cable. PC is connected to a network. This is one of the setups that offers the best acquisition speed.
- Indirect to a PC on a network, with PC and camera on same subnet — PC is connected to a network with a Cat 5e or 6 Ethernet cable. Camera is connected to the same network with a Cat 5e or 6 Ethernet cable. You may connect multiple cameras to the network using separate cables.
- Multiple cameras to a PC directly, using multiple Ethernet cards — PC is connected to camera 1 with a Cat 5e or 6 Ethernet cable. PC is connected to camera 2 with a separate Cat 5e or 6 Ethernet cable. PC is optionally connected to a network. This is one of the setups that offers the best acquisition speed.
- Multiple cameras to a PC directly, using switch or hub — PC is connected to a switch or hub directly with a Cat 5e or 6 Ethernet cable. Camera 1 is connected to switch/hub with a Cat 5e or 6 Ethernet cable. Camera 2 is connected to the switch/hub with a separate Cat 5e or 6 Ethernet cable. PC is optionally connected to a network. Alternatively, switch/hub is optionally connected to a network.

Network Hardware Configuration Notes

The following notes apply to network connections and hardware.

Using the same network as the PC on a shared network connection — Plug the camera into the network that the PC is plugged into. They must be on the same subnet. A system administrator can configure a VLAN if necessary.

Using a private network connection — You can connect the camera through the main/only Ethernet card, or through a second Ethernet card. In either scenario, a switch can be used to connect multiple cameras.

Ethernet cards — Ethernet cards must be 1000 Mbps. If direct connection or PC network allows, use a card that supports jumbo frames for larger packet sizes. Also, on Windows, increase the number of receive buffers if reception is poor.

Switches for connecting multiple cameras — Use a switch that has full duplex 1000 Gbps per port capacity. It can be a managed switch, but does not have to be.

Network Adaptor Configuration Notes

In this section...
“Windows Configuration” on page 10-4
“Linux Configuration” on page 10-5
“Mac Configuration” on page 10-5

Windows Configuration

Important Note: When you install your vendor software that came with your device, do not install your vendor's filtering or performance networking driver.

Let Windows automatically determine the IP if you are using a single direct connection to the PC, instead of attempting to use static IP. Otherwise, leave organizational IP configuration settings in place.

Use your vendor software to configure the camera for DHCP/LLA.

If you have multiple cameras connected to multiple Ethernet cards, you cannot have them all set to automatic IP configuration. You must specify the IP address for each card and each card must be on a different subnet.

Enable large frame support if your Ethernet card, and switch if present, supports it and you are using a direct connection. If you are not using a direct connection, you can enable large frame support if all switches and routers in your organization's network support it.

Set the Receive Buffers high, 2048 for example.

Installation of GigE Vision Cameras and Drivers on Windows

Follow these steps to install a GigE Vision camera on a Windows machine.

1. It is not necessary to install your vendor software that came with your device, but you may want to in order to verify that the device is running outside of MATLAB.

   Important Note: Do not install your vendor's filtering or performance networking driver.
2. In the Windows Network Connections dialog box (part of Control Panel), if using a second network adaptor, you can optionally rename your second network adaptor to “GigE Vision” to help distinguish it from your primary adaptor.

   If the **Status** column says “Limited or no connectivity,” it will not impact your camera, as that status applies to the Internet.
3. Open the Properties dialog box of the Ethernet card by double-clicking it in Network Connections. If you are using a separate Ethernet card for the GigE camera, make sure that in the **This connection uses the following items** section on the **General** tab you have nothing selected except for **Internet Protocol (TCP/IP)**. Be sure to use TCP/IP version 4, and not version 6.

   Make sure that any vendor drivers are unchecked and that anti-virus program drivers are unchecked. If you cannot uncheck the anti-virus software from the adaptor due to organization restrictions, you may need to purchase a second gigabit Ethernet card. In this case, leave all of the options as is for the network card for your PC, and configure the second card as described here, which will only connect to your camera.

4 In Windows Device Manager, make sure your network cards show up as using the correct network card vendor driver.

For example, in the Device Manager window, under **Network adapters**, you should see **Intel PRO/1000 PT Desktop Adapter** if you use that particular Ethernet card.

Check your adaptor properties. If your situation allows, as described in the next section, make sure that **Jumbo Frames** is enabled in the **Settings** on the **Advanced** tab. Make sure that **Receive Descriptors** is enabled in the **Settings > Performance Options** on the **Advanced** tab. Make sure that the correct adaptor is listed in the **Driver** tab and that it has not been replaced with a vendor-specific driver instead of the driver of the Ethernet card.

Note You do not need to install GenICam to use the GigE adaptor, because it is now included in the installation of the toolbox. However, if you are using the From Video Device block and doing code generation, you would need to install GenICam to run the generated application outside of MATLAB.

Linux Configuration

You will not need any drivers from your vendor and we recommend that you do not install any that may have come with your device.

We recommend that you have your system administrator help with the following setup tasks:

- Getting the Ethernet card recognized by the kernel.
- Getting the IP and MTU configuration set up for direct connection.

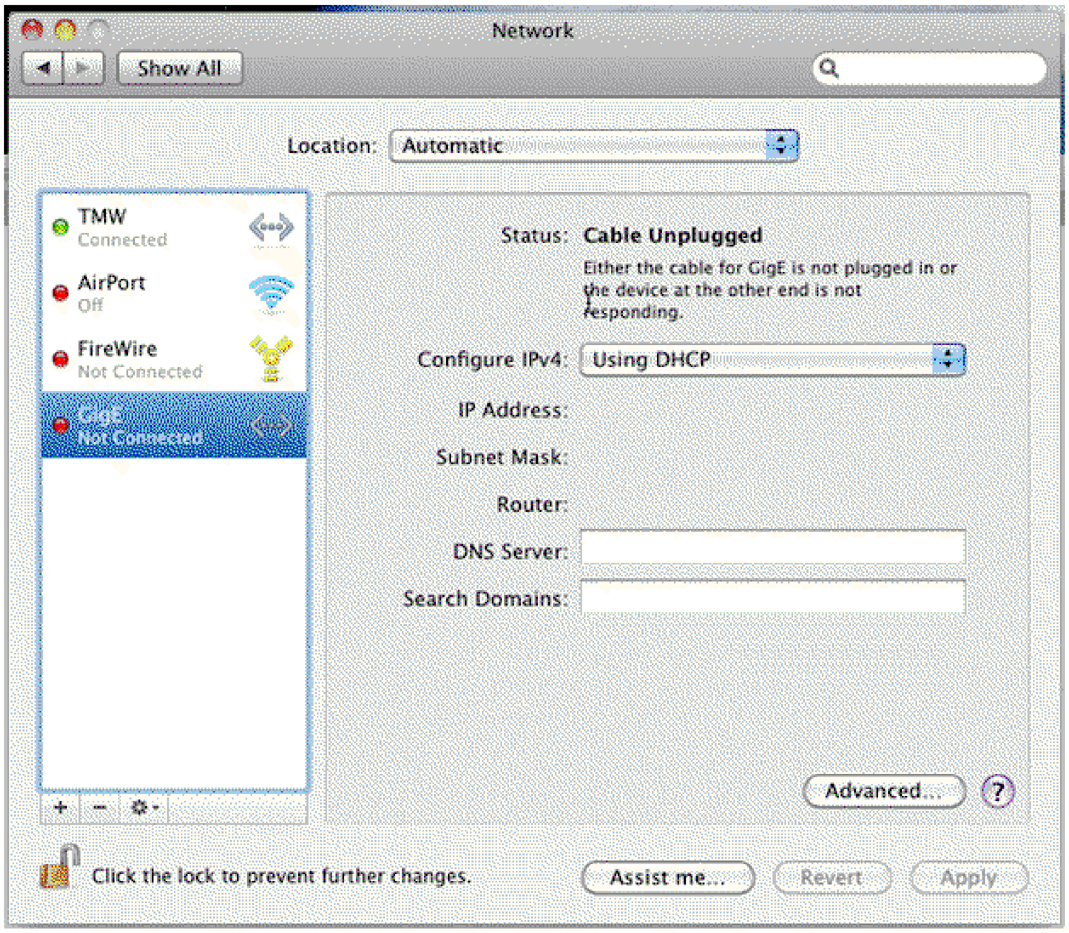
  For dynamic IP configuration of a camera and Ethernet card not connected to an organizational network, avahi-autoipd can be used. However, we recommend that each direct connection to a camera have an interface with a static IP such as `10.10.x.y` or `192.168.x.y`.

  If you want to use jumbo frames and your Ethernet card and switches (if present) allow, configure the MTU accordingly.

Mac Configuration

You will not need any drivers from your vendor and we recommend that you do not install any that may have come with your device.

You should configure your Ethernet connection as shown:



Location: Automatic

Status: **Cable Unplugged**

Either the cable for GigE is not plugged in or the device at the other end is not responding.

Configure IPv4: Using DHCP

IP Address:

Subnet Mask:

Router:

DNS Server:

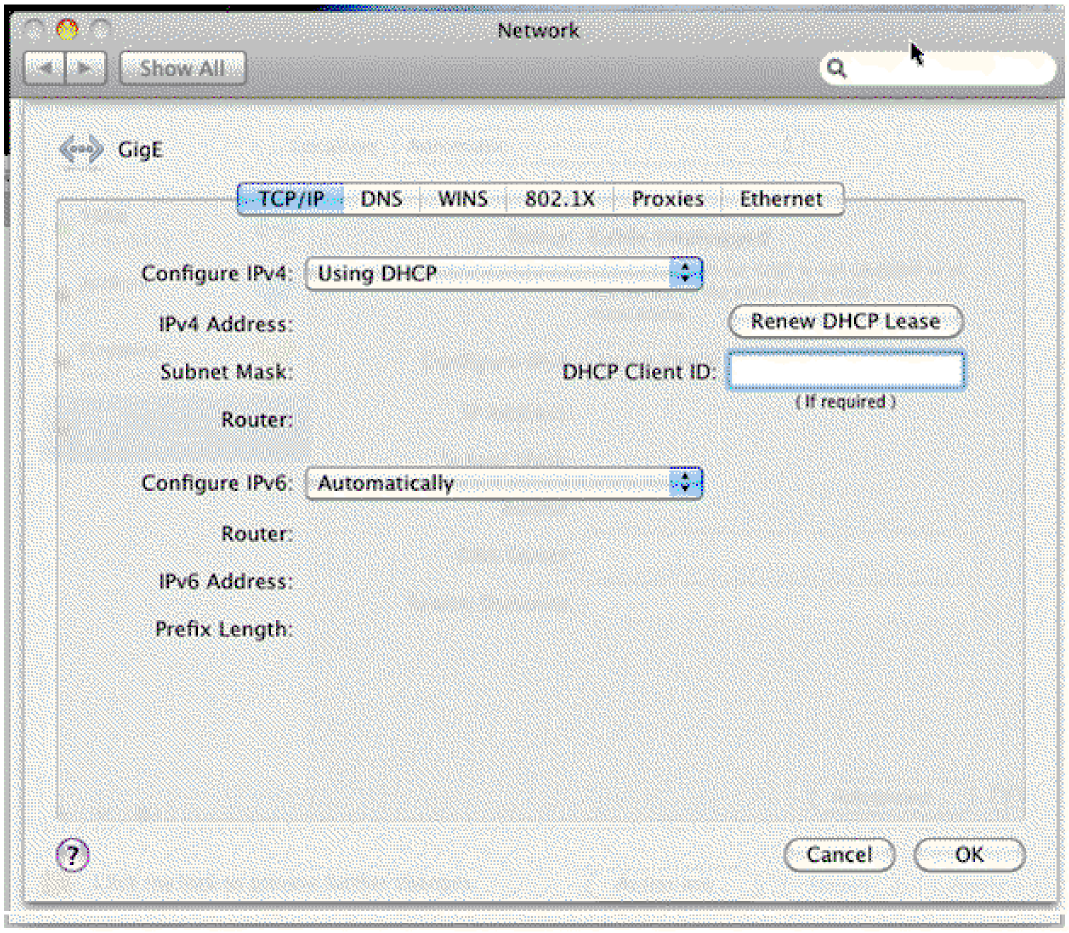
Search Domains:

Click the lock to prevent further changes.

In the configuration shown here, the Mac Pro has two Ethernet connections, one to an internal network, and one for GigE Vision. The GigE Vision connection is set to use DHCP.

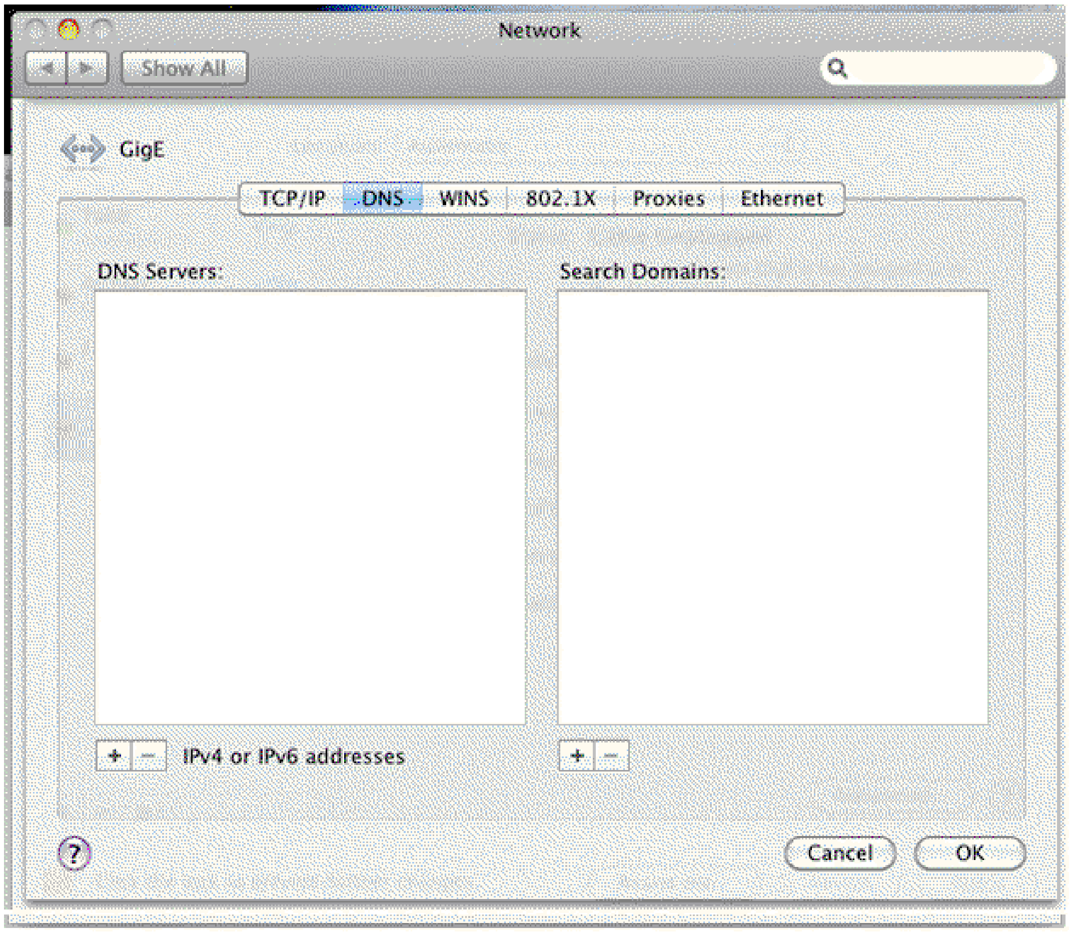
Advanced settings are set as shown in the following diagrams.

The **TCP/IP** tab.

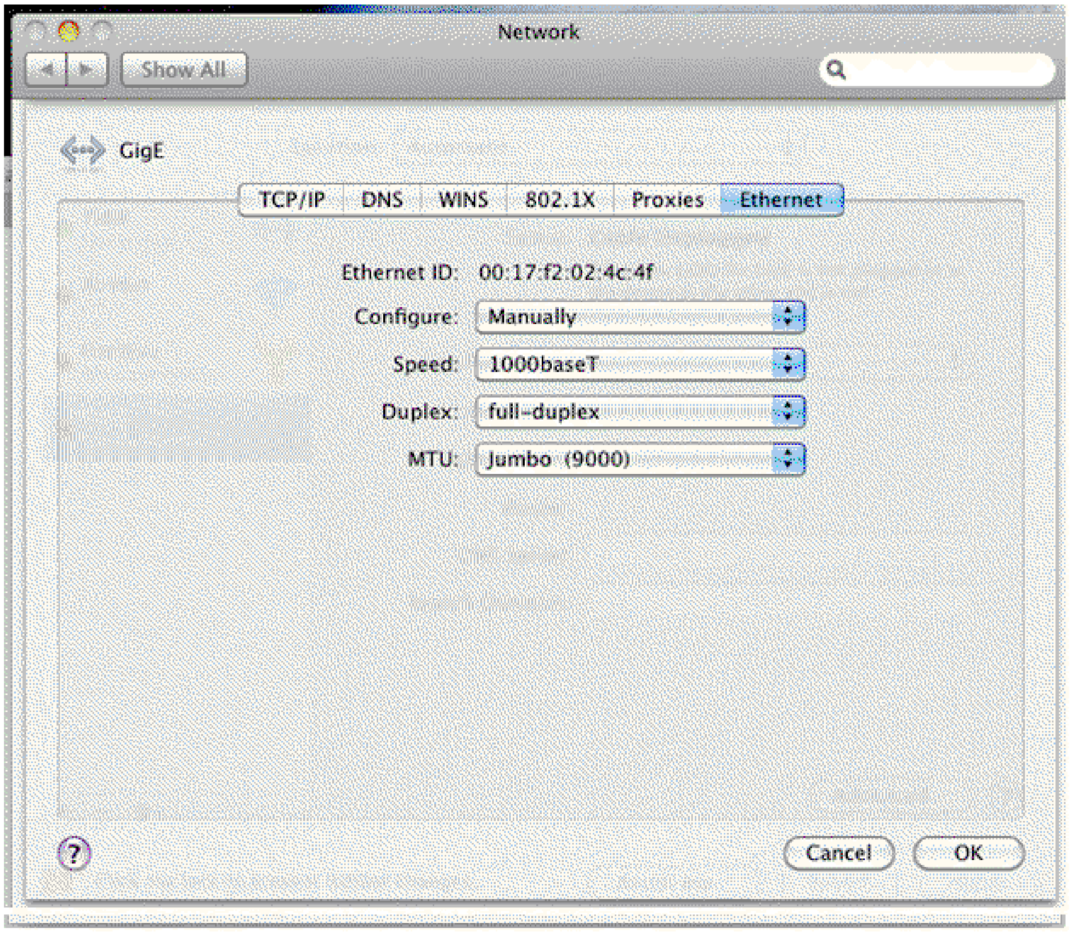


Configure IPv4: Using DHCP
IPv4 Address:
Subnet Mask:
DHCP Client ID:
Router:
Configure IPv6: Automatically
Router:
IPv6 Address:
Prefix Length:

The **DNS** tab.



The **Ethernet** tab.



Ethernet ID: 00:17:f2:02:4c:4f
Configure: Manually
Speed: 1000baseT
Duplex: full-duplex
MTU: Jumbo (9000)

If you are using a MacBook, you may not have the option of Jumbo frames in the **MTU**.

Software Configuration

You need to have GenICam installed, but that is done for you by the Image Acquisition Toolbox. The necessary environment variables should automatically be set as part of the installation. You can optionally check to verify the following environment variables. See the examples below.

Note If you have a camera that requires a GenICam XML file on a local drive (most cameras do not), you should set `MWIMAQ_GENICAM_XML_FILES` environment variable to the directory of your choice, and then install the camera's XML file in that directory. However, most cameras do not require or use local XML files.

Windows Example

```
MWIMAQ_GENICAM_XML_FILES=C:\cameraXML
```

You can test the installation by using the following command:

```
imaqhwinfo('gige')
```

and by looking at the relevant sections of the output when you run the `imaqsupport` function.

Linux Example

```
MWIMAQ_GENICAM_XML_FILES=/local/cameraXML
```

You can test the installation by using the following command:

```
imaqhwinfo('gige')
```

Mac Example

```
MWIMAQ_GENICAM_XML_FILES=/local/cameraXML
```

You can test the installation by using the following command:

```
imaqhwinfo('gige')
```

and by looking at the relevant sections of the output when you run the `imaqsupport` function.

Note You do not need to install GenICam to use the GigE adaptor, because it is now included in the installation of the toolbox. However, if you are using the From Video Device block and doing code generation, you would need to install GenICam to run the generated application outside of MATLAB.

Setting Preferences

There are three GigE Vision related preferences in the Image Acquisition Preferences. In MATLAB, on the **Home** tab, in the **Environment** section, click **Preferences > Image Acquisition**.

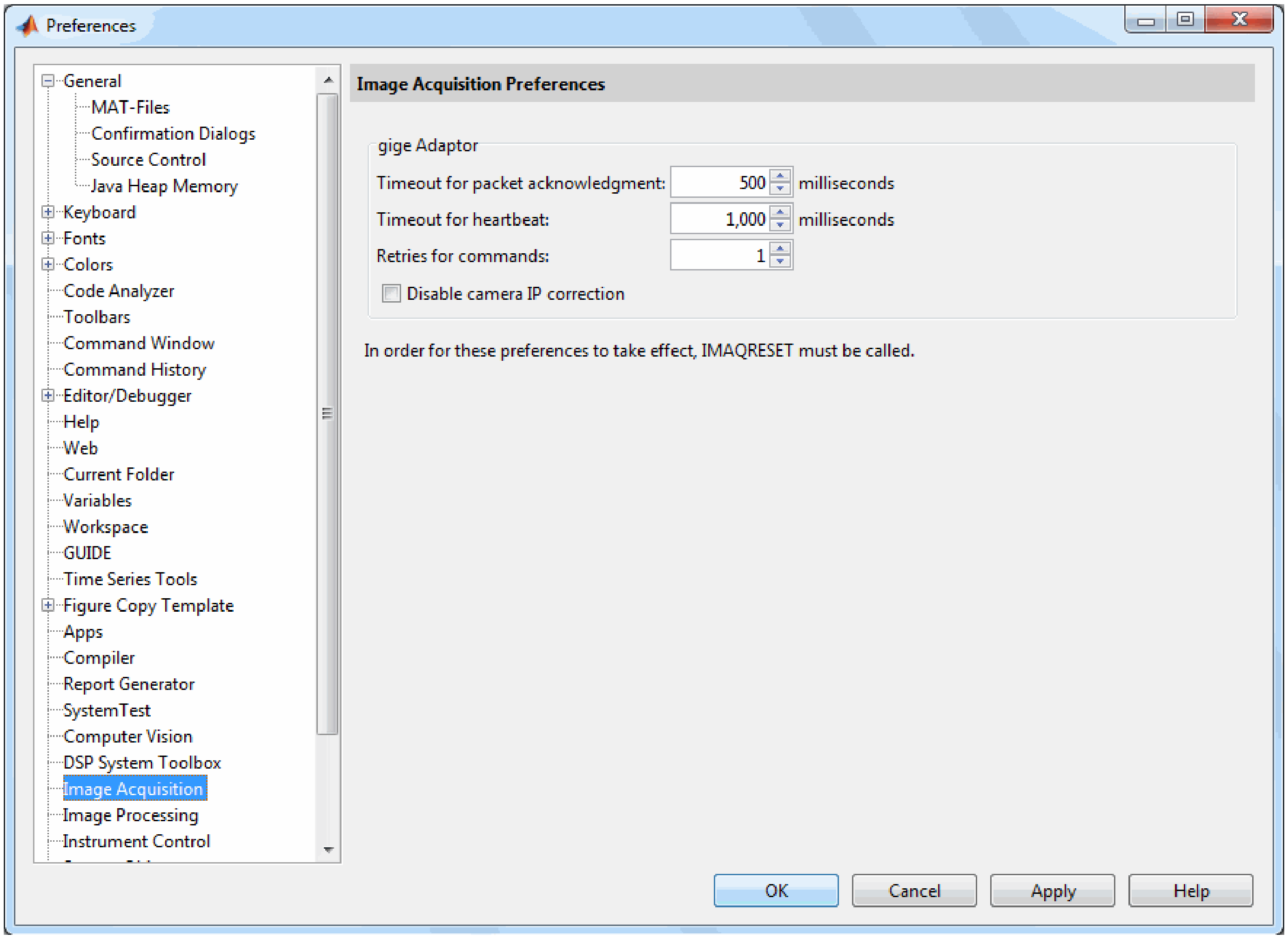


Image Acquisition Preferences

gige Adaptor

Timeout for packet acknowledgment: 500 milliseconds

Timeout for heartbeat: 1,000 milliseconds

Retries for commands: 1

☐ Disable camera IP correction

In order for these preferences to take effect, IMAQRESET must be called.

Timeout for packet acknowledgement – this is a timeout value for the time between the sending of a command (for camera discovery or control) and the time that the acknowledgement is received from the camera.

Timeout for heartbeat – the camera requires that the application send a packet every so often (like a heartbeat) to keep the control connection alive. This is the setting for that packet period. Setting it too low can add unnecessary load to the computer and to the camera. Setting it too high can cause the camera to remain in use too long beyond when the toolbox attempts to relinquish control, leading to a failure to obtain control to start another acquisition.

Retries for commands – this is the number of attempts that the toolbox will make to send a command to the camera before deciding that the send has failed. The time between retries is set by the **Timeout for packet acknowledgement** setting.

Disable camera IP correction – check if you want to disable automatic IP correction for your camera. Clear the check mark to re-enable IP correction.

Troubleshooting

For troubleshooting information for GigE Vision devices on Windows, see “Troubleshooting GigE Vision Devices on Windows” on page 16-19.

For troubleshooting information for GigE Vision devices on Linux, see “Troubleshooting GigE Vision Devices on Linux” on page 16-20.

For troubleshooting information for GigE Vision devices on macOS, see “Troubleshooting GigE Vision Devices on macOS” on page 16-21.

Using the GigE Vision Interface

GigE Vision Acquisition: gigecam Object vs. videoinput Object

The Image Acquisition Toolbox includes a separate interface for use with GigE Vision Compliant cameras. This interface is designed for GigE Vision cameras and supports more GigE-specific functionality.

You can continue to use the GigE Vision adaptor (`gige`) with the `videoinput` object, or you can use the `gigecam` object, which takes advantage of GigE properties and features and is more consistent with GigE Vision conventions for displaying properties and managing selector properties.

Note The GigE Vision support, using either object, requires that you download and install the necessary files via MATLAB Add-Ons. The GigE Vision Hardware support package installs the files for both the `gige` adaptor for the `videoinput` object and the `gigecam` object. For more information, see "Installing the Support Packages for Image Acquisition Toolbox Adaptors" on page 4-5.

Advantages of `gigecam` Object

- Designed for GigE Vision cameras
- Allows use of GigE camera commands
- Better handling of GigE Vision camera properties
- Uses GigE Vision advanced property features

Advantages of `videoinput` Object

- Uses advanced toolbox features such as buffering and callbacks
- Supports code generation
- Supported in **Image Acquisition Explorer**, the VideoDevice System Object, and Simulink

If you do not need to use any advanced toolbox features and you do not need to use **Image Acquisition Explorer**, the VideoDevice System Object, or Simulink, use the `gigecam` object to take advantage of the advanced GigE Vision Standard feature support it offers.

Connect to GigE Vision Cameras

Use the `gigecamlist` function to return the list of available GigE Vision Compliant cameras connected to your system. The function returns a table with the following information for each camera detected: model, manufacturer, IP address, and serial number. If you plug in different cameras during the MATLAB session, the `gigecamlist` function returns an updated list of cameras.

In this example, two cameras have been detected.

```
gigecamlist
```

```
ans =

    Model                Manufacturer           IPAddress        SerialNumber
  ____________________   ___________________   ________________   ______________

  'MV1-D1312-80-G2-12'   'Photonofocus AG'     '169.254.192.165'  '022600017445'
  'mvBlueCOUGER-X120aG'  'MATRIX VISION GmbH'  '169.254.242.122'  'GX000818'
```

Note The GigE Vision support requires that you download and install the necessary files via MATLAB Add-Ons. The GigE Vision Hardware support package installs the files for both the `gige` adaptor for the `videoinput` object and the `gigecam` object. For more information, see “Installing the Support Packages for Image Acquisition Toolbox Adaptors” on page 4-5.

If you have the support package installed and `gigecamlist` does not recognize your camera, see the troubleshooting information in “GigE Vision Hardware” on page 16-19.

Set Properties for GigE Acquisition

In this section...
"Property Display" on page 11-4
"Set GigE Properties" on page 11-6
"Use GigE Commands" on page 11-7

Property Display

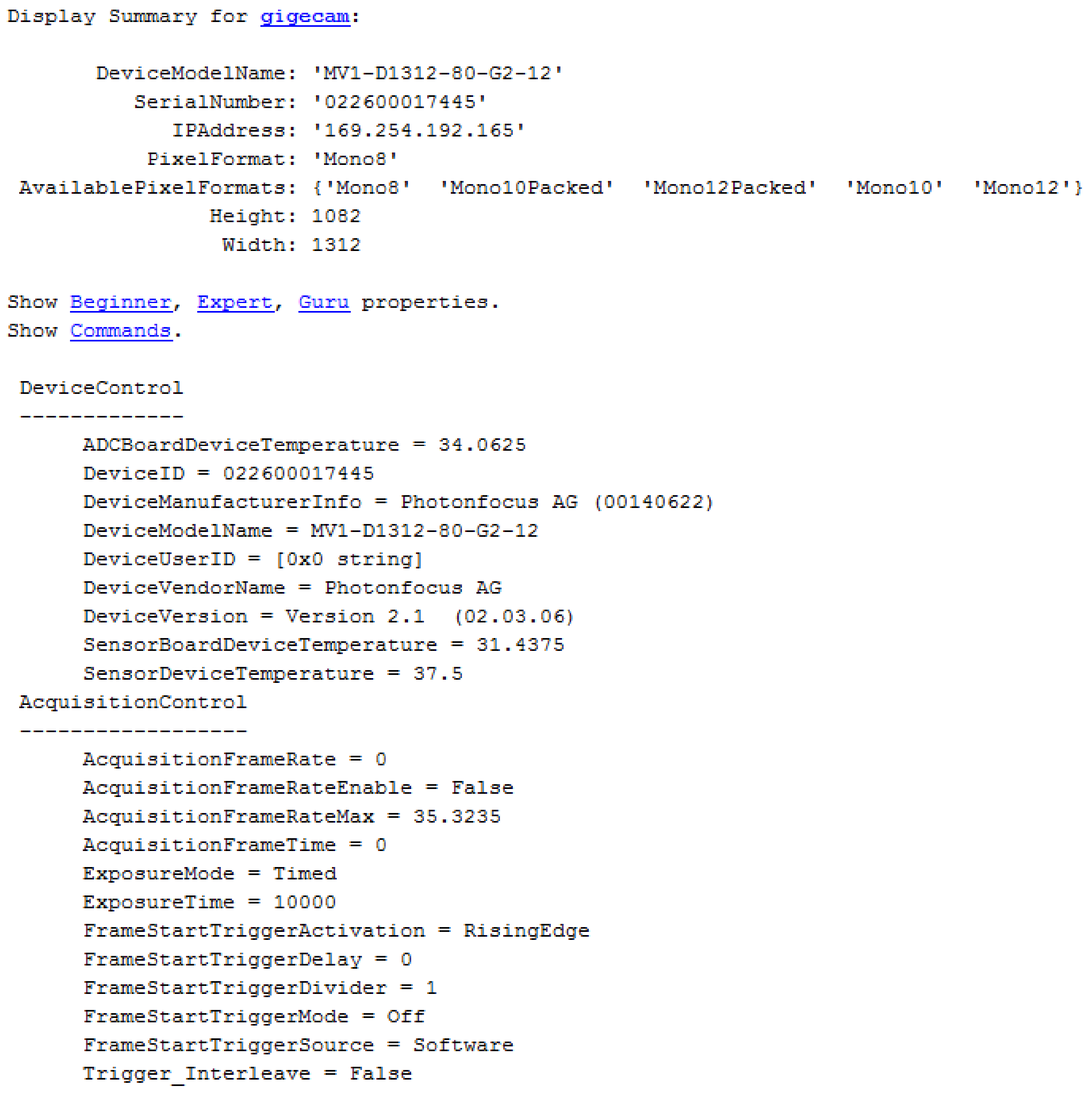
One of the main advantages of using the `gigecam` object for image acquisition, instead of the `gige` adaptor with the `videoinput` object, is the advanced property features of GigE Vision Compliant hardware.

When you create the `gigecam` object, the basic properties are displayed, as shown here.

```
g = gigecam
```

```
g =

Display Summary for gigecam:

         DeviceModelName: 'MV1-D1312-80-G2-12'
            SerialNumber: '022600017445'
               IPAddress: '169.254.192.165'
             PixelFormat: 'Mono8'
   AvailablePixelFormats: {'Mono8' 'Mono10Packed' 'Mono12Packed' 'Mono10' 'Mono12'}
                  Height: 1082
                   Width: 1312
                 Timeout: 10

Show Beginner, Expert, Guru properties.
Show Commands.
```

When you click **Beginner**, the Beginner level camera properties are displayed.



```
Display Summary for gigecam:

       DeviceModelName: 'MV1-D1312-80-G2-12'
          SerialNumber: '022600017445'
             IPAddress: '169.254.192.165'
           PixelFormat: 'Mono8'
 AvailablePixelFormats: {'Mono8'  'Mono10Packed'  'Mono12Packed'  'Mono10'  'Mono12'}
                Height: 1082
                 Width: 1312

Show Beginner, Expert, Guru properties.
Show Commands.

 DeviceControl
 -------------
      ADCBoardDeviceTemperature = 34.0625
      DeviceID = 022600017445
      DeviceManufacturerInfo = Photonfocus AG (00140622)
      DeviceModelName = MV1-D1312-80-G2-12
      DeviceUserID = [0x0 string]
      DeviceVendorName = Photonfocus AG
      DeviceVersion = Version 2.1  (02.03.06)
      SensorBoardDeviceTemperature = 31.4375
      SensorDeviceTemperature = 37.5
 AcquisitionControl
 ------------------
      AcquisitionFrameRate = 0
      AcquisitionFrameRateEnable = False
      AcquisitionFrameRateMax = 35.3235
      AcquisitionFrameTime = 0
      ExposureMode = Timed
      ExposureTime = 10000
      FrameStartTriggerActivation = RisingEdge
      FrameStartTriggerDelay = 0
      FrameStartTriggerDivider = 1
      FrameStartTriggerMode = Off
      FrameStartTriggerSource = Software
      Trigger_Interleave = False
```

The list of available properties is specific to your camera. The display of properties is broken into categories based on GenICam categories as specified by camera manufacturers. For example, in the display shown here, you can see a set of device control properties, and a set of acquisition control properties. There are other categories not shown in this graphic, such as analog control, convolver, and image format control.

The GigE Vision category standard also provides levels of expertise for the available categories. When you create the `gigecam` object, you see a small set of commonly used properties with links to the expanded property list based on expertise. To see the additional properties, click **Beginner**, **Expert**, or **Guru**.

Set GigE Properties

You can set properties two different ways — as additional arguments when you create the object using the `gigecam` function, or anytime after you create the object using the syntax shown in this section.

Set a Property When Creating the Object

When you use the `gigecam` function with no arguments, it creates the object and connects to the single GigE Vision Compliant camera on your system, or to the first camera it finds listed in the output of the `gigecamlist` function if you have multiple cameras. If you use an argument to create the object — either an IP address, index number, or serial number — as described in “Create the gigecam Object” on page 11-8, that argument must be the first argument.

```
g = gigecam('169.254.242.122')
```

To set a property when creating the object, it must be specified as a name-value pair after the IP address, index number, or serial number. The following command creates the object using the camera on the IP address used as the first argument, then sets the `PixelFormat` property to `Mono10`.

```
g = gigecam('169.254.242.122', 'PixelFormat', 'Mono10')
```

If you are creating the object with just one camera connected, you can use the index number `1` as the first input argument, then a property-value pair.

```
g = gigecam(1, 'PixelFormat', 'Mono10')
```

You can set multiple properties in this way, and you can use pairs of either character vectors or numerics.

```
g = gigecam(1, 'ExposureTime', 20000, 'PixelFormat', 'Mono10')
```

Set a Property After Creating the Object

You can set or change properties any time after you create the object, using this syntax, where `g` is the object name.

```
g.ExposureTime = 20000
```

If you want to change the `Timeout` from its default value of 10 seconds, to increase it to 20 seconds for example, use this syntax.

```
g.Timeout = 20
```

This way of setting properties also supports both character vectors and numerics.

```
g.LinLog_Mode = 'On';
```

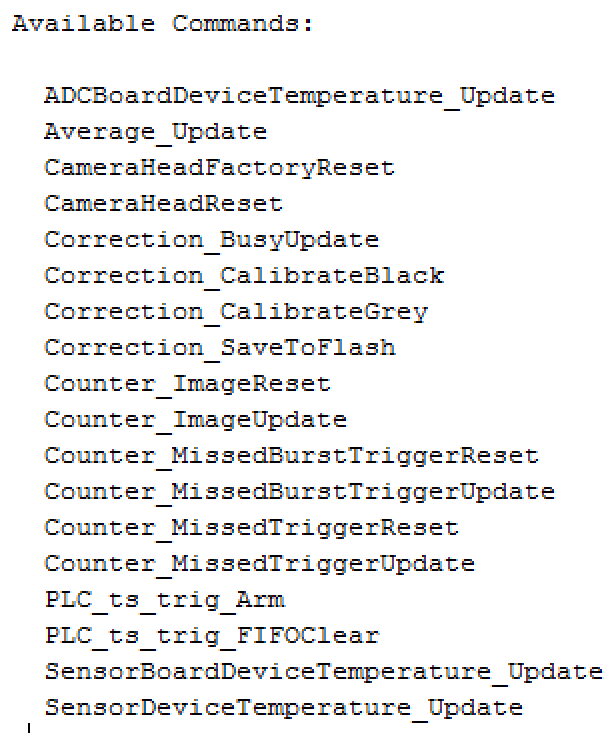
Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Use GigE Commands

You can use any of the GigE camera commands that your camera supports.

The `commands` function tells you what commands are available for your camera to use. The output depends on the commands that are supported by your specific hardware. To get the list, use the `commands` function with the object name, which is `g` in this example.

```
commands(g)
```



```
Available Commands:

  ADCBoardDeviceTemperature_Update
  Average_Update
  CameraHeadFactoryReset
  CameraHeadReset
  Correction_BusyUpdate
  Correction_CalibrateBlack
  Correction_CalibrateGrey
  Correction_SaveToFlash
  Counter_ImageReset
  Counter_ImageUpdate
  Counter_MissedBurstTriggerReset
  Counter_MissedBurstTriggerUpdate
  Counter_MissedTriggerReset
  Counter_MissedTriggerUpdate
  PLC_ts_trig_Arm
  PLC_ts_trig_FIFOClear
  SensorBoardDeviceTemperature_Update
  SensorDeviceTemperature_Update
```

Then use `executeCommand` to execute any of the commands found by the `commands` function. The command name is passed as a character vector. For example, set a calibration correction.

```
executeCommand(g, 'Correction_CalibrateGrey');
```

The camera is set to correct the grey calibration when you acquire images.

You may have a camera that has a command to perform auto focus. With a `gigecam` object named `gcam` and a GigE command named `AutoFocus`.

```
executeCommand(gcam, 'AutoFocus');
```

You can also see the list of commands for your camera by clicking the **Show Commands** link at the bottom of the properties list when you create the `gigecam` object.

Acquire Images from GigE Vision Cameras

In this section...

Create the gigecam Object

To acquire images from a GigE Vision Compliant camera, you first use the `gigecam` function to create a GigE object. You can use it in one of three ways:

- Connect to the first or only camera, using no input arguments
- Specify a camera by IP address, using the address (specified as a character vector) as an input argument
- Specify a camera by the list order, using an index number as the input argument
- Specify a camera by serial number, using the number (as a character vector) as an input argument

You can also optionally set a property when you create the object. For more information, see "Set Properties for GigE Acquisition" on page 11-4.

Note that you cannot create more than one object connected to the same device, and trying to do that generates an error.

After you create the object, you can preview and acquire images.

Note The GigE Vision support requires that you download and install the necessary files via MATLAB Add-Ons. The GigE Vision Hardware support package installs the files for both the `gige` adaptor for the `videoinput` object and the `gigecam` object. For more information, see "Installing the Support Packages for Image Acquisition Toolbox Adaptors" on page 4-5.

Create a gigecam Object Using No Arguments

Use the `gigecamlist` function to ensure that MATLAB is discovering your camera.

```
gigecamlist
```

```
ans = 

    Model                Manufacturer           IPAddress        SerialNumber  
  ____________________   ___________________   _______________   ______________

  'MV1-D1312-80-G2-12'   'Photonofocus AG'     '169.254.192.165'  '022600017445'
```

Using the `gigecam` function with no arguments creates the object, and connects to the single GigE Vision camera on your system. If you have multiple cameras and you use the `gigecam` function with no input argument, it creates the object and connects it to the first camera it finds listed in the output of the `gigecamlist` function.

Create an object, `g`.

```
g = gigecam
```

```
g =

Display Summary for gigecam:

         DeviceModelName: 'MV1-D1312-80-G2-12'
            SerialNumber: '022600017445'
               IPAddress: '169.254.192.165'
             PixelFormat: 'Mono8'
   AvailablePixelFormats: {'Mono8' 'Mono10Packed' 'Mono12Packed' 'Mono10' 'Mono12'}
                  Height: 1082
                   Width: 1312
                 Timeout: 10

Show Beginner, Expert, Guru properties.
Show Commands.
```

Create a gigecam Object Using IP Address

Use the `gigecam` function with the IP address of the camera (specified as a character vector) as the input argument to create the object and connect it to the camera with that address. You can see the IP address for your camera in the list returned by the `gigecamlist` function.

Use the `gigecamlist` function to ensure that MATLAB is discovering your cameras.

```
gigecamlist
```

```
ans =

    Model                 Manufacturer           IPAddress        SerialNumber
  ____________________    __________________    ______________    ______________

  'MV1-D1312-80-G2-12'   'Photonofocus AG'     '169.254.192.165'  '022600017445'
  'mvBlueCOUGER-X120aG'  'MATRIX VISION GmbH'  '169.254.242.122'  'GX000818'
```

Create an object, `g`, using the IP address of the camera.

```
g = gigecam('169.254.242.122')
```

```
g =

Display Summary for gigecam:

         DeviceModelName: 'mvBlueCOUGER-X120aG'
            SerialNumber: 'GX000818'
               IPAddress: '169.254.242.122'
             PixelFormat: 'Mono8'
   AvailablePixelFormats: {'Mono8' 'Mono12' 'Mono14' 'Mono16' 'Mono12Packed'
                           'BayerGR8' 'BayerGR10' 'BayerGR12' 'BayerGR16' 'BayerGR12Packed'
                           'YUV422Packed' 'YUV422_YUYVPacked' 'YUV444Packed'}
                  Height: 1082
                   Width: 1312
                 Timeout: 10

Show Beginner, Expert, Guru properties.
Show Commands.
```

Create a gigecam Object Using Serial Number

You can also create the object in this same way using the serial number. You use the same syntax, but use a serial number instead of the IP address, also as a character vector.

```
g = gigecam('022600017445')
```

Create a gigecam Object Using Device Number as an Index

Use the `gigecam` function with an index as the input argument to create the object corresponding to that index and connect it to that camera. The index corresponds to the order of cameras in the table returned by `gigecamlist` when you have multiple cameras connected.

Use the `gigecamlist` function to ensure that MATLAB is discovering your cameras.

```
gigecamlist

ans = 

    Model                  Manufacturer              IPAddress         SerialNumber
  ____________________    ___________________    _______________    ______________

  'MV1-D1312-80-G2-12'    'Photonofocus AG'      '169.254.192.165'    '022600017445'
  'mvBlueCOUGER-X120aG'   'MATRIX VISION GmbH'   '169.254.242.122'    'GX000818'
```

Create an object, `g`, using the index number.

```
g = gigecam(2)

g = 

Display Summary for gigecam:

         DeviceModelName: 'mvBlueCOUGER-X120aG'
            SerialNumber: 'GX000818'
               IPAddress: '169.254.242.122'
             PixelFormat: 'Mono8'
   AvailablePixelFormats: {'Mono8' 'Mono12' 'Mono14' 'Mono16' 'Mono12Packed'
                           'BayerGR8' 'BayerGR10' 'BayerGR12' 'BayerGR16' 'BayerGR12Packed'
                           'YUV422Packed' 'YUV422_YUYVPacked' 'YUV444Packed'}
                  Height: 1082
                   Width: 1312
                 Timeout: 10

Show Beginner, Expert, Guru properties.
Show Commands.
```

It creates the object and connects it to the Matrix Vision camera with that index number, in this case, the second one displayed by `gigecamlist`. If you only have one camera, you do not need to use the index.

Acquire One Image Frame from a GigE Camera

Use the `snapshot` function to acquire one image frame from a GigE Vision Compliant camera.

1 Use the `gigecamlist` function to ensure that MATLAB is discovering your camera.

   ```
   gigecamlist
   ```

```
ans = 

    Model                 Manufacturer            IPAddress        SerialNumber  
  ____________________   ___________________   _______________    ______________

  'MV1-D1312-80-G2-12'   'Photonofocus AG'     '169.254.192.165'  '022600017445'
```

2 Use the `gigecam` function to create the object and connect it to the camera.

```
g = gigecam
```

```
g = 

Display Summary for gigecam:

         DeviceModelName: 'MV1-D1312-80-G2-12'
            SerialNumber: '022600017445'
               IPAddress: '169.254.192.165'
             PixelFormat: 'Mono8'
   AvailablePixelFormats: {'Mono8' 'Mono10Packed' 'Mono12Packed' 'Mono10' 'Mono12'}
                  Height: 1082
                   Width: 1312
                 Timeout: 10

Show Beginner, Expert, Guru properties.
Show Commands.
```

It creates the object and connects it to the Photonofocus AG camera.

3 Preview the image from the camera.

```
preview(g)
```

The preview window displays live video stream from your camera. The preview dynamically updates, so if you change a property while previewing, the image changes to reflect the property change.

4 Optionally, set any properties. Properties are displayed when you create the object, as shown in step 2. For example, you could change the `ExposureTime` setting.

```
g.ExposureTime = 20000
```

For more information, see “Set Properties for GigE Acquisition” on page 11-4.

5 Optionally, use any of the GigE camera commands that your camera supports.

For more information, see “Set Properties for GigE Acquisition” on page 11-4.

6 Close the preview.

```
closePreview(g)
```

7 Acquire a single image from the camera using the `snapshot` function, and assign it to the variable `img`

```
img = snapshot(g);
```

8 Display the acquired image.

```
imshow(img)
```

9 Clean up by clearing the object.

```
clear g
```

Using the Kinect for Windows Adaptor

Important Information About the Kinect Adaptor

The Kinect Adaptor lets you acquire images using a Kinect for Windows device. Kinects are often used in automotive IVS, robotics, human-computer interaction (HCI), security systems, entertainment systems, game design, and civil engineering. They can be used for analyzing skeletons, 3D mapping, gesture recognition, human travel patterns, sports and games, etc.

The Kinect adaptor is supported on 64-bit Windows.

Doing image acquisition with a Kinect for Windows camera is similar to using other cameras and adaptors, but with several key differences:

- The Kinect for Windows device has two separate physical sensors, and each one uses a different `DeviceID` in the `videoinput` object. The Kinect color sensor returns color image data. The Kinect depth sensor returns depth and skeletal data. For information about Kinect device discovery and the use of two device IDs, see "Detecting the Kinect Devices" on page 12-5.
- The Kinect for Windows device returns four data streams. The image stream is returned by the color sensor and contains color data in various color formats. The depth stream is returned by the depth sensor and returns depth information in pixels. The skeletal stream is returned by the depth sensor and returns metadata about the skeletons. There is also an audio stream, but this is unused by Image Acquisition Toolbox. For details on the streams, see "Data Streams Returned by the Kinect" on page 12-3.
- The Kinect for Windows can track up to six people. It can provide full tracking on two people, and position tracking on up to four more.
- In Image Acquisition Toolbox, skeletal metadata is accessed through the depth sensor object. For an example showing how to access the skeletal metadata, see "Acquiring Image and Skeletal Data Using Kinect" on page 12-7.

Note The Kinect adaptor is intended for use only with the Kinect for Windows sensor.

Note With previous versions of the Image Acquisition Toolbox, the files for all of the adaptors were included in your installation. Starting with version R2014a, each adaptor is available separately through support packages. In order to use the Image Acquisition Toolbox, you must install the adaptor that your camera uses. See "Image Acquisition Support Packages for Hardware Adaptors" on page 4-2 for information about installing the adaptors using MATLAB Add-Ons. See "Installing the Kinect for Windows Sensor Support Package" on page 12-20 for information specific to installing the Kinect support package. Also, in order to use the Kinect for Windows support, you must have version 1.6 of the Kinect for Windows Runtime installed on your system. If you do not already have it installed, it will be installed when you install the Kinect support package.

Data Streams Returned by the Kinect

The Kinect for Windows device returns these data streams.

- Image stream (returned by the color sensor)
- Depth stream (returned by the depth sensor)
- Skeletal stream (returned by the depth sensor)
- Audio stream (not used by the Image Acquisition Toolbox, but could be used with MATLAB audiorecorder)

Image Stream

The image stream returns color image data and other formats using the Kinect color sensor. It supports the following formats.

Format	Description
`RawYUV_640x480`	Raw YUV format. Resolution of 640 x 480, frame rate of 15 frames per second, which is the maximum allowed.
`RGB_1280x960`	RGB format. Resolution of 1280 x 960, frame rate of 12 frames per second, which is the maximum allowed.
`RGB_640x480`	RGB format. Resolution of 640 x 480, frame rate of 30 frames per second, which is the maximum allowed.
`YUV_640x480`	YUV format. Resolution of 640 x 480, frame rate of 15 frames per second, which is the maximum allowed.
`Infrared_640x480`	Infrared format. MONO16 frame type with resolution of 640 x 480, frame rate of 30 frames per second, which is the maximum allowed. The infrared stream allows you to capture frames in low light situations.
`RawBayer_1280x960`	Raw Bayer format. MONO8 frame type with resolution of 1280 x 960, frame rate of 12 frames per second, which is the maximum allowed. This format returns the raw Bayer pattern, so you can use your own algorithm to reconstruct the color image.

Format	Description
`RawBayer_640x480`	Raw Bayer format. MONO8 frame type with resolution of 640 x 480, frame rate of 30 frames per second, which is the maximum allowed. This format returns the raw Bayer pattern, so you can use your own algorithm to reconstruct the color image.

Depth Stream

The depth stream returns person segmentation data using the Kinect depth sensor. The depth map is distance in millimeters from the camera plane. For Skeletal Tracking only two people can be tracked at a given time, although six people can be segmented at a time. This means it can provide full tracking on two skeletons, and partial position tracking on up to four more. The tracking ranges are a default range of 50 cm to 400 cm and a near range of 40 cm to 300 cm.

The depth stream supports the following formats.

Format	Description
`Depth_640x480`	Resolution of 640 x 480, frame rate of 30 frames per second
`Depth_320x240`	Resolution of 320 x 240, frame rate of 30 frames per second
`Depth_80x60`	Resolution of 80 x 60, frame rate of 30 frames per second

Skeletal Stream

The skeletal stream returns skeletal data using the Kinect depth device. The skeleton frame returned contains data on the ground plane position and a time stamp. It contains the overall position of the skeleton and the 3-D position of all 20 joints (position in meters). Two skeletons are actively tracked, and another four are tracked passively.

Note To understand the differences in using the Kinect adaptor compared to other toolbox adaptors, see “Important Information About the Kinect Adaptor” on page 12-2. For information about Kinect device discovery and the use of two device IDs, see “Detecting the Kinect Devices” on page 12-5. For an example that shows how to access the skeletal metadata, see “Acquiring Image and Skeletal Data Using Kinect” on page 12-7.

Detecting the Kinect Devices

Typically in the Image Acquisition Toolbox, each camera or image device has one `DeviceID`. Because the Kinect for Windows camera has two separate sensors, the color sensor and the depth sensor, the toolbox lists two `DeviceIDs`. If you use `imaqhwinfo` on the adaptor, you can see this.

```
info = imaqhwinfo('kinect');
info

info =

       AdaptorDllName: '<matlabroot>\toolbox\imaq\imaqadaptors\win64\mwkinectimaq.dll'
    AdaptorDllVersion: '4.6 (R2013b)'
          AdaptorName: 'kinect'
            DeviceIDs: {[1]  [2]}
           DeviceInfo: [1x2 struct]
```

You can see the two device IDs in the output.

If you look at each device, you can see that they represent the color sensor and the depth sensor. The following shows the color sensor.

```
info.DeviceInfo(1)

ans =

             DefaultFormat: 'RGB_640x480'
       DeviceFileSupported: 0
                DeviceName: 'Kinect Color Sensor'
                  DeviceID: 1
     VideoInputConstructor: 'videoinput('kinect', 1)'
    VideoDeviceConstructor: 'imaq.VideoDevice('kinect', 1)'
          SupportedFormats: {'RGB_1280x960'  'RGB_640x480'  'RawYUV_640x480'  'YUV_640x480'
                               'Infrared_640x480'  'RawBayer_1280x960'  'RawBayer_640x480'}
```

In the output, you can see that Device 1 is the color sensor.

The following shows the depth sensor, which is Device 2.

```
info.DeviceInfo(2)

ans =

             DefaultFormat: 'Depth_640x480'
       DeviceFileSupported: 0
                DeviceName: 'Kinect Depth Sensor'
                  DeviceID: 2
     VideoInputConstructor: 'videoinput('kinect', 2)'
    VideoDeviceConstructor: 'imaq.VideoDevice('kinect', 2)'
          SupportedFormats: {'Depth_640x480'  'Depth_320x240'  'Depth_80x60'}
```

You can use multiple Kinect cameras together. Multiple Kinect sensors are enumerated as `DeviceIDs [1] [2] [3] [4]` and so on. For example, if you had two Kinect cameras, the first one would have `Kinect Color Sensor` with `DeviceID 1` and `Kinect Depth Sensor` with `DeviceID 2` and the second Kinect camera would have `Kinect Color Sensor` with `DeviceID 3` and `Kinect Depth Sensor` with `DeviceID 4`.

Note To understand the differences in using the Kinect adaptor compared to other toolbox adaptors, see “Important Information About the Kinect Adaptor” on page 12-2. For more information on the Kinect streams, see “Data Streams Returned by the Kinect” on page 12-3. For an example that shows how to access the skeletal metadata, see “Acquiring Image and Skeletal Data Using Kinect” on page 12-7.

Acquiring Image and Skeletal Data Using Kinect

In “Detecting the Kinect Devices” on page 12-5, you could see that the two sensors on the Kinect for Windows are represented by two device IDs, one for the color sensor and one of the depth sensor. In that example, Device 1 is the color sensor and Device 2 is the depth sensor. This example shows how to create a `videoinput` object for the color sensor to acquire RGB images and then for the depth sensor to acquire skeletal data.

1. Create the `videoinput` object for the color sensor. `DeviceID` 1 is used for the color sensor.

   ```
   vid = videoinput('kinect',1,'RGB_640x480');
   ```
2. Look at the device-specific properties on the source device, which is the color sensor on the Kinect camera.

   ```
   src = getselectedsource(vid);

   src

   Display Summary for Video Source Object:

         General Settings:
           Parent = [1x1 videoinput]
           Selected = on
           SourceName = ColorSource
           Tag =
           Type = videosource

         Device Specific Properties:
           Accelerometer = [0.0 -1.0 0.0]
           AutoExposure = on
           AutoWhiteBalance = on
           BacklightCompensation = AverageBrightness
           Brightness = 0.2156
           CameraElevationAngle = 3
           Contrast = 1
           ExposureTime = 1.0
           FrameInterval = 0
           FrameRate = 30
           Gain = 0
           Gamma = 2.2
           Hue = 0
           PowerLineFrequency = Disabled
           Saturation = 1
           Sharpness = 0.5
           WhiteBalance = 2700
   ```

   As you can see in the output, the color sensor has a set of device-specific properties.

Device-Specific Property – Color Sensor	Description
`Accelerometer`	Returns 3D vector of acceleration data for both the color and depth sensors. The data is updated while the device is running or previewing. This 1 x 3 double represents the `x`, `y`, and `z` values of acceleration in gravity units `g` (`9.81m/s^2`). For example, `[0.06 -1.00 -0.09]` represents values of `x` as `0.06` g, `y` as `-1.00` g, and `z` as `-0.09` g.
`AutoExposure`	Use to set the exposure automatically. This control whether other related properties are activated. Values are `on` (default) and `off`. `on` means that exposure is set automatically, and these properties are not able to be set and will throw a warning: `FrameInterval`, `ExposureTime`, and `Gain`. `off` means that these properties are not able to be set and will throw a warning: `PowerLineFrequency`, `BacklightCompensation`, and `Brightness`.
`AutoWhiteBalance`	Use to enable or disable automatic white balance setting. `on` (default) means that it will automatically configure white balance and the `WhiteBalance` property cannot be set. `off` means that the `WhiteBalance` property is settable.
`BacklightCompensation`	Configures backlight compensation modes to adjust the camera to capture images dependent on environmental conditions. Note that this property is only valid if `AutoExposure` is set to `Enabled`. The default is `AverageBrightness`. Values are: `AverageBrightness` favors an average brightness level `CenterPriority` favors the center of the scene `LowLightsPriority` favors a low light level `CenterOnly` favors the center only

Device-Specific Property – Color Sensor	Description
`Brightness`	Indicates the brightness level. The value range is `0.0` to `1.0`, and the default value is `0.2156`. Note that this property is only valid if `AutoExposure` is set to `Enabled`.
`CameraElevationAngle`	Controls the angle of the sensor lens. This is the camera angle relative to the ground. The value must be an integer property with range of -27 to 27 degrees. The default value is the last set value, since this is a sticky setting. Only set it if you want to change the angle of the camera. This property is shared with the depth sensor also.
`Contrast`	Indicates contrast level. Values must be in the range `0.5` to `2`, with a default value of `1`.
`ExposureTime`	Indicates the exposure time in increments of 1/10,000 of a second. The value range is `0` to `4000`, and the default is `0`. Note that this property is only valid if `AutoExposure` is set to `Disabled`.
`FrameInterval`	Indicates the frame interval in units of 1/10,000 of a second. The value range is `0` to `4000`, and the default is `0`. Note that this property is only valid if `AutoExposure` is set to `Disabled`.
`FrameRate`	Frames per second for the acquisition. This property is read only and the possible values for the color sensor are `12`, `15`, and `30` (default). It reflects the actual frame rate when running.
`Gain`	Indicates a multiplier for the RGB color values. The value range is `1.0` to `16.0`, and the default is `1.0`. Note that this property is only valid if `AutoExposure` is set to `Disabled`.
`Gamma`	Indicates gamma measurement. Values must be in the range `1` to `2.8`, with a default value of `2.2`.
`Hue`	Indicates hue setting. Values must be in the range `-22` to `22`, with a default value of `0`.

Device-Specific Property – Color Sensor	Description
PowerLineFrequency	Option for reducing flicker caused by the frequency of a power line. Values are Disabled, FiftyHertz, and SixtyHertz. The default is Disabled. Note that this property is only valid if AutoExposure is set to Enabled.
Saturation	Indicates saturation level. Values must be in the range 0 to 2, with a default value of 1.
Sharpness	Indicates sharpness level. Values must be in the range 0 to 1, with a default value of 0.5.
WhiteBalance	Indicates color temperature in degrees Kelvin. The value range is 2700 to 6500 and the default is 2700. Note that this property is only valid if AutoWhiteBalance is set to Disabled.

3 You can optionally set some of these properties shown in the previous step. For example, you might be acquiring images in a low light situation. You could adjust the acquisition for this by setting the BacklightCompensation property to LowLightsPriority, which favors a low light level.

```
src.BacklightCompensation = 'LowLightsPriority';
```

4 Preview the color stream by calling preview on the color sensor object created in step 1.

```
preview(vid);
```

When you are done previewing, close the preview window.

```
closepreview(vid);
```

5 Create the videoinput object for the depth sensor. Note that a second object is created (vid2), and DeviceID 2 is used for the depth sensor.

```
vid2 = videoinput('kinect',2,'Depth_640x480');
```

6 Look at the device-specific properties on the source device, which is the depth sensor on the Kinect.

```
src = getselectedsource(vid2);

src

Display Summary for Video Source Object:

      General Settings:
        Parent = [1x1 videoinput]
        Selected = on
        SourceName = DepthSource
        Tag = 
        Type = videosource

      Device Specific Properties:
```

```
Accelerometer = [0.0 -1.0 0.0]
BodyPosture = Standing
CameraElevationAngle = 4
DepthMode = Default
FrameRate = 30
IREmitter = on
SkeletonsToTrack = [1x0 double]
TrackingMode = off
```

As you can see in the output, the depth sensor has a set of device-specific properties associated with skeletal tracking. These properties are specific to the depth sensor.

Device-Specific Property – Depth Sensor	**Description**
`Accelerometer`	Returns 3D vector of acceleration data for both the color and depth sensors. The data is updated while the device is running or previewing. This 1 x 3 double represents the `x`, `y`, and `z` values of acceleration in gravity units `g` (`9.81m/s^2`). For example, `[0.06 -1.00 -0.09]` represents values of `x` as `0.06` g, `y` as `-1.00` g, and `z` as `-0.09` g.
`BodyPosture`	Indicates whether the tracked skeletons are standing or sitting. Values are `Standing` (gives 20 point skeleton data) and `Seated` (gives 10 point skeleton data, using joint indices 2 - 11). `Standing` is the default. Note that if `BodyPosture` is set to `Seated` mode, and `TrackingMode` is set to `Position`, no position is returned, since `Position` is the location of the hip joint and the hip joint is not tracked in `Seated` mode. See the subsection “BodyPosture Joint Indices” at the end of this example for the list of indices of the 20 skeletal joints.
`CameraElevationAngle`	Controls the angle of the sensor lens. This is the camera angle relative to the ground. The value must be an integer property with range of -27 to 27 degrees. The default value is the last set value, since this is a sticky setting. Only set it if you want to change the angle of the camera. This property is shared with the color sensor also.
`DepthMode`	Indicates the range of depth in the depth map. Values are `Default` (range of 50 to 400 cm) and `Near` (range of 40 to 300 cm).

Device-Specific Property – Depth Sensor	Description
`FrameRate`	Frames per second for the acquisition. This property is read only and is fixed at `30` for the depth sensor for all formats. It reflects the actual frame rate when running.
`IREmitter`	Controls whether the IR emitter is on or off. Values are `on` and `off`. Initially, the default value is `on`. However, this is a sticky property, so the default value is the last set value. If you set it to `off`, it will remain off in future uses until you change the setting. An advantage of this property is that it is useful when using multiple Kinect devices to avoid interference.
`SkeletonsToTrack`	Indicates the Skeleton Tracking ID returned as part of the metadata. Values are: `[]` Default tracking `[TrackingID1]` Track 1 skeleton with Tracking ID = TrackingID1 `[TrackingID1 TrackingID2]` Track 2 skeletons with Tracking IDs = TrackingID1 and TrackingID2
`TrackingMode`	Indicates tracking state. Values are: `Skeleton` tracks full skeleton with joints `Position` tracks hip joint position only `Off` disables skeleton position tracking (default) Note that if `BodyPosture` is set to `Seated` mode, and `TrackingMode` is set to `Position`, no position is returned, since `Position` is the location of the hip joint and the hip joint is not tracked in `Seated` mode.

7 Start the second `videoinput` object (the depth stream).

```
start(vid2);
```

8 Skeletal data is accessed as metadata on the depth stream. You can use `getdata` to access it.

```
% Get the data on the object.
[frame, ts, metaData] = getdata(vid2);

% Look at the metadata to see the parameters in the skeletal data.
metaData

metaData =

10x1 struct array with fields:
    AbsTime: [1x1 double]
    FrameNumber: [1x1 double]
    IsPositionTracked: [1x6 logical]
    IsSkeletonTracked: [1x6 logical]
    JointDepthIndices: [20x2x6 double]
    JointImageIndices: [20x2x6 double]
    JointTrackingState: [20x6 double]
    JointWorldCoordinates: [20x3x6 double]
    PositionDepthIndices: [2x6 double]
    PositionImageIndices: [2x6 double]
    PositionWorldCoordinates: [3x6 double]
    RelativeFrame: [1x1 double]
    SegmentationData: [640x480 double]
    SkeletonTrackingID: [1x6 double]
    TriggerIndex: [1x1 double]
```

These metadata fields are related to tracking the skeletons.

MetaData	Description
`AbsTime`	This is a 1 x 1 double and represents the full timestamp, including date and time, in MATLAB clock format.
`FrameNumber`	This is a 1 x 1 double and represents the frame number.
`IsPositionTracked`	This is a 1 x 6 Boolean matrix of true/false values for the tracking of the position of each of the six skeletons. A `1` indicates the position is tracked and a `0` indicates it is not.
`IsSkeletonTracked`	This is a 1 x 6 Boolean matrix of true/false values for the tracked state of each of the six skeletons. A `1` indicates it is tracked and a `0` indicates it is not.
`JointDepthIndices`	If the `BodyPosture` property is set to `Standing`, this is a 20 x 2 x 6 double matrix of x-and y-coordinates for 20 joints in pixels relative to the depth image, for the six possible skeletons. If `BodyPosture` is set to `Seated`, this would be a 10 x 2 x 6 double for 10 joints.
`JointImageIndices`	If the `BodyPosture` property is set to `Standing`, this is a 20 x 2 x 6 double matrix of x-and y-coordinates for 20 joints in pixels relative to the color image, for the six possible skeletons. If `BodyPosture` is set to `Seated`, this would be a 10 x 2 x 6 double for 10 joints.

MetaData	Description
`JointTrackingState`	This 20 x 6 integer matrix contains enumerated values for the tracking accuracy of each joint for all six skeletons. Values include: `0` not tracked `1` position inferred `2` position tracked
`JointWorldCoordinates`	This is a 20 x 3 x 6 double matrix of x-, y- and z-coordinates for 20 joints, in meters from the sensor, for the six possible skeletons, if the `BodyPosture` is set to `Standing`. If it is set to `Seated`, this would be a 10 x 3 x 6 double for 10 joints. See step 9 for the syntax on how to see this data.
`PositionDepthIndices`	A 2 x 6 double matrix of X and Y coordinates of each skeleton in pixels relative to the depth image.
`PositionImageIndices`	A 2 x 6 double matrix of X and Y coordinates of each skeleton in pixels relative to the color image.
`PositionWorldCoordinates`	A 3 x 6 double matrix of the X, Y and Z coordinates of each skeleton in meters relative to the sensor.
`RelativeFrame`	This 1 x 1 double represents the frame number relative to the execution of a trigger if triggering is used.
`SegmentationData`	Image size double array with each pixel mapped to a tracked/detected skeleton, represented by numbers 1 to 6. This segmentation map is a bitmap with pixel values corresponding to the index of the person in the field-of-view who is closest to the camera at that pixel position. A value of 0 means there is no tracked skeleton.
`SkeletonTrackingID`	This 1 x 6 integer matrix contains the tracking IDs of all six skeletons. These IDs track specific skeletons using the `SkeletonsToTrack` property in step 5. Tracking IDs are generated by the Kinect and change from acquisition to acquisition.

MetaData	Description
TriggerIndex	This is a 1 x 1 double and represents the trigger the event is associated with if triggering is used.

9 You can look at any individual property by drilling into the metadata. For example, look at the IsSkeletonTracked property.

```
metaData.IsSkeletonTracked

ans =

     1     0     0     0     0     0
```

In this case it means that of the six possible skeletons, there is one skeleton being tracked and it is in the first position. If you have multiple skeletons, this property is useful to confirm which ones are being tracked.

10 Get the joint locations for the first person in world coordinates using the JointWorldCoordinates property. Since this is the person in position 1, the index uses 1.

```
metaData.JointWorldCoordinates(:,:,1)

ans =

   -0.1408   -0.3257    2.1674
   -0.1408   -0.2257    2.1674
   -0.1368   -0.0098    2.2594
   -0.1324    0.1963    2.3447
   -0.3024   -0.0058    2.2574
   -0.3622   -0.3361    2.1641
   -0.3843   -0.6279    1.9877
   -0.4043   -0.6779    1.9877
    0.0301   -0.0125    2.2603
    0.2364    0.2775    2.2117
    0.3775    0.5872    2.2022
    0.4075    0.6372    2.2022
   -0.2532   -0.4392    2.0742
   -0.1869   -0.8425    1.8432
   -0.1869   -1.2941    1.8432
   -0.1969   -1.3541    1.8432
   -0.0360   -0.4436    2.0771
    0.0382   -0.8350    1.8286
    0.1096   -1.2114    1.5896
    0.1196   -1.2514    1.5896
```

The columns represent the X, Y, and Z coordinates in meters of the 20 points on skeleton 1.

11 You can optionally view the segmentation data as an image.

```
% View the segmentation data as an image.
imagesc(metaDataDepth.SegmentationData);
% Set the color map to jet to color code the people detected.
colormap(jet);
```

BodyPosture Joint Indices

The `BodyPosture` property, in step 5, indicates whether the tracked skeletons are standing or sitting. Values are `Standing` (gives 20 point skeleton data) and `Seated` (gives 10 point skeleton data, using joint indices 2 - 11).

This is the order of the joints returned by the Kinect adaptor:

```
Hip_Center = 1;
Spine = 2;
Shoulder_Center = 3;
Head = 4;
Shoulder_Left = 5;
Elbow_Left = 6;
Wrist_Left = 7;
Hand_Left = 8;
Shoulder_Right = 9;
Elbow_Right = 10;
Wrist_Right = 11;
Hand_Right = 12;
Hip_Left = 13;
Knee_Left = 14;
Ankle_Left = 15;
Foot_Left = 16;
Hip_Right = 17;
Knee_Right = 18;
Ankle_Right = 19;
Foot_Right = 20;
```

When `BodyPosture` is set to `Standing`, all 20 indices are returned, as shown above. When `BodyPosture` is set to `Seated`, numbers 2 through 11 are returned, since this represents the upper body of the skeleton.

Note To understand the differences in using the Kinect adaptor compared to previous toolbox adaptors, see "Important Information About the Kinect Adaptor" on page 12-2. For information about Kinect device discovery and the use of two device IDs, see "Detecting the Kinect Devices" on page 12-5. For an example of simultaneous acquisition, see "Acquiring from Color and Depth Devices Simultaneously" on page 12-17.

Acquiring from Color and Depth Devices Simultaneously

You can synchronize the data from the Kinect for Windows color stream and the depth stream using software manual triggering.

This synchronization method example triggers both objects manually.

1. Create the objects for the color and depth sensors. Device 1 is the color sensor and Device 2 is the depth sensor.

   ```
   vid = videoinput('kinect',1);
   vid2 = videoinput('kinect',2);
   ```
2. Get the source properties for the depth device.

   ```
   srcDepth = getselectedsource(vid2);
   ```
3. Set the frames per trigger for both devices to 1.

   ```
   vid.FramesPerTrigger = 1;
   vid2.FramesPerTrigger = 1;
   ```
4. Set the trigger repeat for both devices to 200, in order to acquire 201 frames from both the color sensor and the depth sensor.

   ```
   vid.TriggerRepeat = 200;
   vid2.TriggerRepeat = 200;
   ```
5. Configure the camera for manual triggering for both sensors.

   ```
   triggerconfig([vid vid2],'manual');
   ```
6. Start both video objects.

   ```
   start([vid vid2]);
   ```
7. Trigger the devices, then get the acquired data.

   ```
   % Trigger 200 times to get the frames.
   for i = 1:201
       % Trigger both objects.
       trigger([vid vid2])
       % Get the acquired frames and metadata.
       [imgColor, ts_color, metaData_Color] = getdata(vid);
       [imgDepth, ts_depth, metaData_Depth] = getdata(vid2);
   end
   ```

Using Skeleton Viewer for Kinect Skeletal Data

If you do an acquisition with a Kinect for Windows and get skeletal data, you can view the skeleton joints in this viewer. This example function displays one RGB image with skeleton joint locations overlaid on the image.

1. Create the Kinect objects and acquire image and skeletal data, as shown in “Acquiring Image and Skeletal Data Using Kinect” on page 12-7.
2. Use the `skeletonViewer` function to view the skeletal data.

   In this code, `skeleton` is the joint image locations returned by the Kinect depth sensor, and `image` is the RGB image corresponding to the skeleton frame. `nSkeleton` is the number of skeletons.

   ```
   function [] = skeletonViewer(skeleton, image, nSkeleton)
   ```

   This is the order of the joints returned by the Kinect adaptor:

   ```
      Hip_Center = 1;
      Spine = 2;
      Shoulder_Center = 3;
      Head = 4;
      Shoulder_Left = 5;
      Elbow_Left = 6;
      Wrist_Left = 7;
      Hand_Left = 8;
      Shoulder_Right = 9;
      Elbow_Right = 10;
      Wrist_Right = 11;
      Hand_Right = 12;
      Hip_Left = 13;
      Knee_Left = 14;
      Ankle_Left = 15;
      Foot_Left = 16;
      Hip_Right = 17;
      Knee_Right = 18;
      Ankle_Right = 19;
      Foot_Right = 20;
   ```
3. Show the RGB image.

   ```
   imshow(image);
   ```
4. Create a skeleton connection map to link the joints.

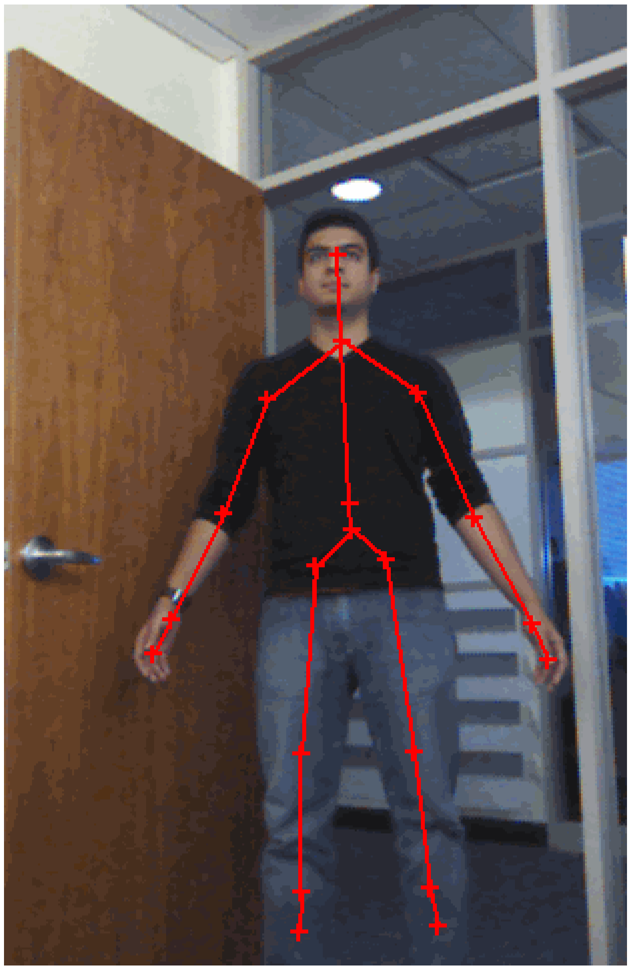
   ```
   SkeletonConnectionMap = [[1 2]; % Spine
                            [2 3];
                            [3 4];
                            [3 5]; %Left Hand
                            [5 6];
                            [6 7];
                            [7 8];
                            [3 9]; %Right Hand
                            [9 10];
                            [10 11];
                            [11 12];
                            [1 17]; % Right Leg
                            [17 18];
   ```

```
                               [18 19];
                               [19 20];
                               [1 13]; % Left Leg
                               [13 14];
                               [14 15];
                               [15 16]];
```

5 Draw the skeletons on the RGB image.

```
for i = 1:19

     if nSkeleton > 0
         X1 = [skeleton(SkeletonConnectionMap(i,1),1,1) skeleton(SkeletonConnectionMap(i,2),1,1)];
         Y1 = [skeleton(SkeletonConnectionMap(i,1),2,1) skeleton(SkeletonConnectionMap(i,2),2,1)];
         line(X1,Y1, 'LineWidth', 1.5, 'LineStyle', '-', 'Marker', '+', 'Color', 'r');
     end
     if nSkeleton > 1
         X2 = [skeleton(SkeletonConnectionMap(i,1),1,2) skeleton(SkeletonConnectionMap(i,2),1,2)];
         Y2 = [skeleton(SkeletonConnectionMap(i,1),2,2) skeleton(SkeletonConnectionMap(i,2),2,2)];
         line(X2,Y2, 'LineWidth', 1.5, 'LineStyle', '-', 'Marker', '+', 'Color', 'g');
     end
    hold on;
 end
 hold off;
```

The viewer will show the following for this example, which contains the color image of one person, with the skeletal data overlaid on the image.



Installing the Kinect for Windows Sensor Support Package

With previous versions of the Image Acquisition Toolbox, the files for all of the adaptors were included in your installation. Starting with version R2014a, each adaptor is available separately through MATLAB Add-Ons. In order to use the Image Acquisition Toolbox, you must install the adaptor that your camera uses, in this case, the Kinect for Windows Sensor support package.

In order to use the Kinect for Windows support in the Image Acquisition Toolbox, you must have the correct version of the Kinect for Windows Runtime installed on your system. If you do not already have it installed, it will be installed when you install the Kinect support package. After you complete the support package installation, you can acquire images using the Kinect for Windows V1 or V2 with the Image Acquisition Toolbox.

Using this installation process, you download and install the following file(s) on your host computer:

- MATLAB files to use Kinect for Windows V1 cameras with the toolbox
- MATLAB files to use Kinect for Windows V2 cameras with the toolbox
- Kinect for Windows Runtime, if you do not already have a current version installed

Note You can use this support package only on a host computer running a version of 64-bit Windows that Image Acquisition Toolbox supports.

If the installation fails, see the **Troubleshooting** section at the end of this topic.

1. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.
2. In the Add-On Explorer, scroll to the **Hardware Support Packages** section, and click **show all** to find your support package.
3. You can refine the list by selecting **Imaging/Cameras** in the **Refine by Hardware Type** section on the left side of the Explorer.
4. Select Image Acquisition Toolbox Support Package for Kinect For Windows Sensor.

Troubleshooting

If the setup fails, it could be caused by an internet security setting. If you get an error message such as “KINECT Setup Failed – An error occurred while installing,” try the following and then run the installer again.

1. In Internet Explorer, go to **Tools > Internet Options**.
2. In Internet Options, select the **Advanced** tab.
3. Under the **Security** subsection, uncheck **Check for publisher’s certificate revocation** to temporarily disable it, and click **OK**.
4. Run the installer again.
5. After you have installed the support package, re-enable the security option in Internet Explorer.

Using the Matrox Interface

Matrox Acquisition – matroxcam Object vs videoinput Object

The Image Acquisition Toolbox includes a separate interface for use with Matrox frame grabbers. This interface is designed for Matrox hardware and supports more Matrox-specific functionality.

You can continue to use the Matrox adaptor (`matrox`) with the `videoinput` object, or you can use the `matroxcam` object, which takes advantage of Matrox features.

Note The Matrox support, using either object, requires that you download and install the necessary files via MATLAB Add-Ons. The Matrox Hardware support package installs the files for both the `matrox` adaptor for the `videoinput` object and the `matroxcam` object. For more information, see "Installing the Support Packages for Image Acquisition Toolbox Adaptors" on page 4-5.

Advantages of `matroxcam` Object

- More robust support for a wider configuration of supported devices. Supports more device configurations than the `videoinput` `matrox` adaptor.
- Added support for the MIL 10.
- Supports newer devices.

Advantages of `videoinput` Object

- Uses advanced toolbox features such as buffering and callbacks
- Supported in **Image Acquisition Explorer**, the VideoDevice System Object, and Simulink

If you do not need to use any advanced toolbox features and you do not need to use the **Image Acquisition Explorer**, the VideoDevice System Object, or the Simulink block, use the `matroxcam` object to take advantage of the advanced Matrox feature support it offers.

Note that the `matroxcam` and `matroxlist` functions that are available with the `matroxcam` object starting in release R2015a work with MIL or MIL-Lite 10.x only. Use of the `matrox` adaptor with the `videoinput` object can use either MIL 9 or 10.

Connect to Matrox Frame Grabbers

Use the `matroxlist` function to return the list of available Matrox frame grabbers connected to your system. The function returns a cell array with model name and digitizer number for each frame grabber detected.

In this example, three frame grabbers have been detected.

```
matroxlist
```

```
ans =

   Solios XCL (digitizer 0)
   Solios XCL (digitizer 1)
   VIO (digitizer 0)
```

If no boards are detected, it returns an empty cell array.

Note The Matrox support requires that you download and install the necessary files via MATLAB Add-Ons. The Matrox Hardware support package installs the files for both the `matrox` adaptor for the `videoinput` object and the `matroxcam` object. For more information, see "Installing the Support Packages for Image Acquisition Toolbox Adaptors" on page 4-5.

If you have the support package installed and `matroxlist` does not recognize your hardware, see the troubleshooting information in "Matrox Hardware" on page 16-10.

Set Properties for Matrox Acquisition

You cannot directly set properties for the `matroxcam` object in the Image Acquisition Toolbox. To set acquisition properties, use your Digitizer Configuration File (DCF) file. You can set properties in the DCF file using the Matrox Intellicam software. The DCF file contains properties relating to exposure signal, grab mode, sync signal, camera, video signal, video timing, and pixel clock. Once you have configured these properties in your DCF file, you create the `matroxcam` object using that file name and path as an input argument.

1. Set any properties you want to configure in your DCF file, using the Matrox Intellicam software.
2. Use the `matroxlist` function to ensure that MATLAB is discovering your frame grabber.

   ```
   matroxlist

   ans =

      Solios XCL (digitizer 0)
      Solios XCL (digitizer 1)
      VIO (digitizer 0)
   ```
3. Use the `matroxcam` function to create the object and connect it to the frame grabber. If you want to use the second frame grabber in the list, the Solios XCL at digitizer 1, use a `2` as the index number, since it is the second device on the list. The second argument must be the name of your DCF file, entered as a character vector. It must contain the fully qualified path to the file as well. In this example, the DCF file is named `mycam.dcf`.

   ```
   m = matroxcam(2, 'C:\Drivers\Solios\dcf\XCL\Basler\A404K\mycam.dcf')

   m =

   Display Summary for matroxcam:

            DeviceName: 'Solios XCL (digitizer 1)'
               DCFName: 'C:\Drivers\Solios\dcf\XCL\Basler\A404K\mycam.dcf'
       FrameResolution: '1300 x 1080'
               Timeout: 10
   ```

   The four properties shown when you create the object are read-only properties that identify the frame grabber.
4. You can then preview and acquire images, as described in “Acquire Images from Matrox Frame Grabbers” on page 13-6.

Note If you need to change any properties after you preview your image, you must change them in the DCF file, and then create a new object to pick up the changes.

Configuring Hardware Triggering

If your DCF file is configured for hardware triggering, then you must provide the trigger to acquire images. To do that, call the `snapshot` function as you normally would, as described in “Acquire One Image Frame from a Matrox Frame Grabber” on page 13-7, and then perform the hardware trigger to acquire the frame.

Note that when you call the `snapshot` function with hardware triggering set, it will not timeout as it normally would. Therefore, the MATLAB command-line will be blocked until you perform the hardware trigger.

Acquire Images from Matrox Frame Grabbers

In this section...

Create the matroxcam Object

To acquire images from a Matrox frame grabber, use the `matroxcam` function to create a Matrox object. Specify a frame grabber by the list order, using an index number, as the first input argument. The second input argument must be the name and fully qualified path of your DCF file.

Note that you cannot create more than one object connected to the same device, and trying to do that generates an error.

After you create the object, you can preview and acquire images.

Note The Matrox support requires that you download and install the necessary files via MATLAB Add-Ons. The Matrox Hardware support package installs the files for both the `matrox` adaptor for the `videoinput` object and the `matroxcam` object. For more information, see "Installing the Support Packages for Image Acquisition Toolbox Adaptors" on page 4-5.

Create a matroxcam Object Using Device Number as an Index

Use the `matroxcam` function with an index as the first input argument to create the object corresponding to that index and connect it to that frame grabber. The index corresponds to the order of boards in the cell array returned by `matroxlist` when you have multiple frame grabbers connected. If you only have one frame grabber, you must use a `1` as the input argument.

Use the `matroxlist` function to ensure that MATLAB is discovering your frame grabbers.

```
matroxlist
```

```
ans =

   Solios XCL (digitizer 0)
   Solios XCL (digitizer 1)
   VIO (digitizer 0)
```

Create an object, `m`, using the index number and DCF file. If you want to use the second frame grabber in the list, the Solios XCL at digitizer 1, use a `2` as the index number, since it is the second camera on the list. The second argument must be the name of your DCF file, entered as a character vector. It must contain the fully qualified path to the file as well. In this example, the DCF file is named `mycam.dcf`.

```
m = matroxcam(2, 'C:\Drivers\Solios\dcf\XCL\Basler\A404K\mycam.dcf')
```

```
m =

Display Summary for matroxcam:

         DeviceName: 'Solios XCL (digitizer 1)'
```

```
        DCFName: 'C:\Drivers\Solios\dcf\XCL\Basler\A404K\mycam.dcf'
FrameResolution: '1300 x 1080'
        Timeout: 10
```

It creates the object and connects it to the Solios XCL with that index number, in this case, the second one displayed by `matroxlist`. The DCF file is specified so that the acquisition can use the properties you have set in your DCF file.

The four properties shown when you create the object are read-only properties that identify the frame grabber.

Acquire One Image Frame from a Matrox Frame Grabber

Use the `snapshot` function to acquire one image frame from a Matrox frame grabber.

1 Use the `matroxlist` function to ensure that MATLAB is discovering your frame grabber.

```
matroxlist
```

```
ans =

   Solios XCL (digitizer 0)
   Solios XCL (digitizer 1)
   VIO (digitizer 0)
```

2 Use the `matroxcam` function to create the object and connect it to the frame grabber. If you want to use the second frame grabber in the list, the Solios XCL at digitizer 1, use a `2` as the index number, since it is the second board on the list. The second argument must be the name and path of your DCF file, entered as a character vector.

```
m = matroxcam(2, 'C:\Drivers\Solios\dcf\XCL\Basler\A404K\mycam.dcf')
```

```
m =

Display Summary for matroxcam:

         DeviceName: 'Solios XCL (digitizer 1)'
            DCFName: 'C:\Drivers\Solios\dcf\XCL\Basler\A404K\mycam.dcf'
    FrameResolution: '1300 x 1080'
            Timeout: 10
```

It creates the object and connects it to the Solios XCL with that index number, in this case, the second one displayed by `matroxlist`. The DCF file is specified so that the acquisition can use the properties you have set in your DCF file.

3 Preview the image from the camera.

```
preview(m)
```

4 You can leave the **Preview** window open, or close it any time. To close the preview:

```
closePreview(m)
```

5 Acquire a single image using the `snapshot` function, and assign it to the variable `img`

```
img = snapshot(m);
```

6 Display the acquired image.

```
imshow(img)
```

7 Clean up by clearing the object.

```
clear m
```

Configuring Hardware Triggering

If your DCF file is configured for hardware triggering, then you must provide the trigger to acquire images. To do that, call the `snapshot` function as you normally would, as shown in step 5, and then perform the hardware trigger to acquire the frame.

Note that when you call the `snapshot` function with hardware triggering set, it will not timeout as it normally would. Therefore, the MATLAB command-line will be blocked until you perform the hardware trigger.

Using the VideoDevice System Object

VideoDevice System Object Overview

The Image Acquisition Toolbox introduces the VideoDevice System object, which allows single-frame image acquisition and code generation from MATLAB.

You use the `imaq.VideoDevice` function to create the System object. It supports the same adaptors and hardware that the `videoinput` object supports; however, it has different functions and properties associated with it. For example, the System object uses the `step` function to acquire single frames.

Creating the VideoDevice System Object

You use the `imaq.VideoDevice` function to create the System object. You can specify the `adaptorname`, `deviceid`, and `format` at the time of object creation, or it will use defaults, as follows.

Constructor	Purpose
`obj = imaq.VideoDevice`	Creates a VideoDevice System object, `obj`, that acquires images from a specified image acquisition device. When you specify no parameters, by default, it selects the first available device for the first adaptor returned by `imaqhwinfo`.
`obj = imaq.VideoDevice(adaptorname)`	Creates a VideoDevice System object, `obj`, using the first device of the specified `adaptorname`. `adaptorname` is a character vector that specifies the name of the adaptor used to communicate with the device. Use the `imaqhwinfo` function to determine the adaptors available on your system.
`obj = imaq.VideoDevice(adaptorname, deviceid)`	Creates a VideoDevice System object, `obj`, with the default format for specified `adaptorname` and `deviceid`. `deviceid` is a numeric scalar value that identifies a particular device available through the specified `adaptorname`. Use the `imaqhwinfo(adaptorname)` syntax to determine the devices available and corresponding values for `deviceid`.
`obj = imaq.VideoDevice(adaptorname, deviceid, format)`	Creates a VideoDevice System object, `obj`, where `format` is a character vector that specifies a particular video format supported by the device or a device configuration file (also known as a camera file).
`obj = imaq.VideoDevice(adaptorname, deviceid, format, P1, V1, ...)`	Creates a VideoDevice System object, `obj`, with the specified property values. If an invalid property name or property value is specified, the object is not created.

Specifying properties at the time of object creation is optional. They can also be specified after the object is created. See "Using Properties on a VideoDevice System Object" on page 14-8 for a list of applicable properties.

Using VideoDevice System Object to Acquire Frames

You can use these functions with the VideoDevice System object.

Function	Purpose
`step`	Acquire a single frame from the image acquisition device. `frame = step(obj);` acquires a single frame from the VideoDevice System object, `obj`. Note that the first time you call step, it acquires exclusive use of the hardware and will start streaming data.
`release`	Release VideoDevice resources and allow property value changes. `release(obj)` releases system resources (such as memory, file handles, or hardware connections) of System object, `obj`, and allows all its properties and input characteristics to be changed.
`isLocked`	Returns a value that indicates if the VideoDevice resource is locked. (Use `release` to unlock.) `L = isLocked(obj)` returns a logical value, `L`, which indicates whether properties are locked for the System object, obj. The object performs an internal initialization the first time the `step` function is executed. This initialization locks properties and input specifications. Once this occurs, the `isLocked` function returns a value of `true`.
`preview`	Activate a live image preview window. `preview(obj)` creates a Video Preview window that displays live video data for the VideoDevice System object, `obj`. The Video Preview window displays the video data at 100% magnification. The size of the preview image is determined by the value of the VideoDevice System object `ROI` property. If not specified, it uses the default resolution for the device.
`closepreview`	Close live image preview window. `closepreview(obj)` closes the live preview window for VideoDevice System object, `obj`.
`imaqhwinfo`	Returns information about the object. `imaqhwinfo(obj)` displays information about the VideoDevice System object, `obj`.

The basic workflow for using the VideoDevice System object is to create the object, preview the image, set any properties, acquire a frame, and clear the object, as shown here.

1. Construct a VideoDevice System object associated with the Winvideo adaptor with device ID of 1.

   ```
   vidobj = imaq.VideoDevice('winvideo', 1);
   ```
2. Set an object-level property, such as `ReturnedColorSpace`.

   ```
   vidobj.ReturnedColorSpace = 'grayscale';
   ```

   Note that the syntax for setting an object-level property is `<object_name>.<property_name> = <property_value>`, where the value can be a character vector or a numeric.
3. Set a device-specific property, such as `Brightness`.

   ```
   vidobj.DeviceProperties.Brightness = 150;
   ```

   Note that the syntax for setting a device-specific property is to list the object name, the `DeviceProperties` object, and the property name using dot notation, and then make it equal to the property value.
4. Preview the image.

   ```
   preview(vidobj)
   ```
5. Acquire a single frame using the `step` function.

   ```
   frame = step(vidobj);
   ```
6. Display the acquired frame.

   ```
   imshow(frame)
   ```
7. Release the hardware resource.

   ```
   release(vidobj);
   ```
8. Clear the VideoDevice System object.

   ```
   clear vidobj;
   ```

Kinect for Windows Metadata

You can return Kinect for Windows skeleton data using the VideoDevice System object on the Kinect Depth sensor.

Typically in the Image Acquisition Toolbox, each camera or image device has one device ID. Because the Kinect for Windows camera has two separate sensors, the Color sensor and the Depth sensor, the toolbox lists two device IDs. The Kinect Color sensor is device 1 and the Kinect depth sensor is device 2.

This example uses a Kinect V1 device. The toolbox also supports Kinect V2. For information on the properties and metadata of Kinect V2 devices, install the Image Acquisition Toolbox Support Package for Kinect For Windows Sensor and see the “Acquire Images with Kinect V2” section in the documentation.

To create a System object using the Color sensor:

```
vidobjcolor = imaq.VideoDevice('kinect', 1);
```

To create a System object using the Depth sensor:

```
vidobjdepth = imaq.VideoDevice('kinect', 2);
```

The Depth sensor returns skeleton metadata. To access this, you need to add a second output argument for the `step` function. The Color sensor works the same way as other devices. So acquiring a frame using the Kinect Color sensor is done as shown here:

```
imageData = step(vidobjcolor);
```

where `imageData` is the frame acquired if `vidobjcolor` is a System object created with Device 1, the Kinect Color sensor.

The Kinect Depth sensor requires a second output argument, as shown here:

```
[depthData metadata] = step(vidobjdepth);
```

where `depthData` is the frame acquired if `vidobjdepth` is a System object created with Device 2, the Kinect Depth sensor, and `metadata` is the skeleton metadata returned with the frame.

These metadata fields are related to tracking the skeletons. The metadata is returned as a structure that contains these parameters:

```
    IsPositionTracked
    IsSkeletonTracked
    JointDepthIndices
    JointImageIndices
    JointTrackingState
    JointWorldCoordinates
    PositionDepthIndices
    PositionImageIndices
    PositionWorldCoordinates
    SegmentationData
    SkeletonTrackingID
```

You can then look at both outputs. To see the image frame:

```
imshow(imageData)
```

To see the metadata output:

```
metadata
```

Note The Kinect for Windows Depth sensor may take some seconds to be ready to begin acquiring skeletal metadata. In order to see values in the metadata output, you need to acquire multiple frames using the step function repeatedly. You can do this by using a for loop.

Note By default the System object returns data as single precision values with the range 0.0 to 1.0. The value represents the fraction through the sensor's dynamic range. The Kinect depth sensor has a range of 0 to 8192 mm.

"Acquiring Image and Skeletal Data Using Kinect" on page 12-7 is an example that shows how to access the skeletal metadata using the `videoinput` object (not the VideoDevice System object), and it contains information about the properties you can set on both the Color and Depth sensors, and descriptions of all the metadata fields. The property names and values are the same as they would be for the System object, but you would then need to set the properties as shown in step 3 of the above example (in the current topic) for use with the VideoDevice System object.

Using Properties on a VideoDevice System Object

You can specify properties at the time of object creation, or they can be specified and changed after the object is created.

Properties that can be used with the VideoDevice System object include:

Property	Description
`Device`	Device from which to acquire images. Specify the image acquisition device to use to acquire a frame. It consists of the device name, adaptor, and device ID. The default device is the first device returned by `imaqhwinfo`. `obj.Device` shows the list of available devices for VideoDevice System object, `obj`.
`VideoFormat`	Video format to be used by the image acquisition device. Specify the video format to use while acquiring the frame. The default value of `VideoFormat` is the default format returned by `imaqhwinfo` for the selected device. To specify a Video Format using a device file, set the `VideoFormat` property to `'From device file'` This option exists only if your device supports device configuration files. `obj.VideoFormat` shows the list of available video formats.
`DeviceFile`	Name of file specifying video format. This property is only visible when `VideoFormat` is set to `'From device file'`.
`DeviceProperties`	Object containing properties specific to the image acquisition device. `obj.DeviceProperties.<property_name> = <property_value>` shows a device-specific property for VideoDevice System object, `obj`.
`ROI`	Region-of-interest for acquisition. This is set to the default ROI value for the specified device, which is the maximum resolution possible for the specified format. You can change the value to change the size of the captured image. The format is 1-based, that is, it is specified in pixels in a 1-by-4 element vector `[x y width height]`, where `x` is x offset and `y` is y offset. Note that this differs from the `videoinput` object and the From Video Device block, which are 0-based.

Property	Description
`HardwareTriggering`	Turn hardware triggering on/off. Set this property to `'on'` to enable hardware triggering to acquire images. The property is visible only when the device supports hardware triggering.
`TriggerConfiguration`	Specifies the trigger source and trigger condition before acquisition. The triggering condition must be met via the trigger source before a frame is acquired. This property is visible only when `HardwareTriggering` is set to `'on'`. `obj.TriggerConfiguration` shows the list of available hardware trigger configurations.
`ReturnedColorSpace`	Specify the color space of the returned image. The default value of the property depends on the device and the video format selected. Possible values are {`rgb`|`grayscale`|`YCbCr`} when the default returned color space for the device is not `grayscale`. Possible values are {`rgb`|`grayscale`|`YCbCr`|`bayer`} when the default returned color space for the device is `grayscale` `obj.ReturnedColorSpace` shows the list of available color space settings.
`BayerSensorAlignment`	Character vector indicating the 2x2 sensor alignment. Specifies Bayer patterns returned by hardware. Specify the sensor alignment for Bayer demosaicing. The default value of this property is `'grbg'`. Possible values are {`grbg`|`gbrg`|`rggb`|`bggr`}. Visible only if `ReturnedColorSpace` is set to `'bayer'`. `obj.BayerSensorAlignment` shows the list of available sensor alignments.
`ReturnedDataType`	The returned data type of the acquired frame. The default `ReturnedDataType` is `single`. `obj.ReturnedDataType` shows the list of available data types.
`ReadAllFrames`	Specify whether to read one image frame or all available frames. Set to `'on'` to capture all available image frames. When set to the default of `'off'`, the System object takes a snapshot of one frame, which is the equivalent of the `getsnapshot` function in the toolbox. When the option is on, all available image frames are captured, which is the equivalent of the `getdata` function in the toolbox.

Note The setting of properties for the System object supports tab completion for enumerated properties while coding in MATLAB. Using the tab completion is an easy way to see available property values. After you type the property name, type a comma, then a space, then the first quote mark for the value, then hit tab to see the possible values.

Once you have created a VideoDevice System object, you can set either object-level properties or device-specific properties on it.

To set an object-level property, use this syntax:

```
vidobj.ReturnedColorSpace = 'grayscale';
```

You can see that the syntax for setting an object-level property is to use `<object_name>.<property_name> = <property_value>`, where the value may be a character vector or a numeric.

Another example of an object-level property is setting the region-of-interest, or `ROI`, to change the dimensions of the acquired image. The ROI format is specified in pixels in a 1-by-4 element vector `[x y width height]`.

```
vidobj.ROI = [1 1 200 200];
```

Note This ROI value is 1-based. This differs from the `videoinput` object and the From Video Device block, which are 0-based.

To set a device-specific property, use this syntax:

```
vidobj.DeviceProperties.Brightness = 150;
```

You can see that the syntax for setting a device-specific property is to use dot notation with the object name, the `DeviceProperties` object, and the property name and then make it equal to the property value.

Another example of a device-specific property is setting the frame rate for a device that supports it.

```
vidobj.DeviceProperties.FrameRate = '30';
```

Note Once you have done a step, in order to change a property or set a new one, you need to release the object using the `release` function, before setting the new property.

Code Generation with VideoDevice System Object

In this section...

Using the codegen Function

The VideoDevice System object supports code generation in MATLAB via the `codegen` function. To use the `codegen` function, you must have a MATLAB Coder license. System objects also support code generation using the MATLAB Function block in Simulink. You can also use the System object with MATLAB Compiler™.

Note The MATLAB Compiler software supports System objects for use inside MATLAB functions. The MATLAB Compiler does not support System objects for use in MATLAB scripts.

Note If you use the `codegen` command to generate a MEX function on a Windows platform, you need to perform `imaqreset` before running the generated MEX file. However, MEX does not work correctly if the Kinect for Windows Sensor support package is installed.

After running the generated MEX file, if you run some MATLAB code that includes a VideoDevice System object with a camera adaptor that is also used in the generated MEX file, you need to perform `clear mex` first.

Note The `codegen` command can be used to generate executable files on non-Windows platforms. However, generation of the MEX function is not supported on Linux and macOS platforms.

For more information see the documentation for the MATLAB `codegen` function.

Shared Library Dependencies

The VideoDevice System object generates code with limited portability. The System object uses precompiled shared libraries, such as DLLs, to support I/O for specific types of devices. The shared library locations that the generated executable requires are as follows:

- Specific MathWorks shared libraries under [MATLABROOT]\bin\<ARCH>\
- MathWorks adaptor libraries under [MATLABROOT]\SupportPackages\<RELEASE>\toolbox\imaq \supportpackages\genericvideo\adaptor\<ARCH>\ specific to the device selected.

For example, your path may look like this on a Windows system and using release R2018a:

C:\ProgramData\MATLAB\SupportPackages\R2018a\toolbox\imaq\supportpackages\genericvideo \adaptor\win64

You will need to add the above folders to your system path before running the generated executable outside of MATLAB.

Usage Rules for System Objects in Generated MATLAB Code

- Assign System objects to persistent variables.
- Global variables are not supported.
- Initialize System objects once by embedding the object handles in an `if` statement with a call to `isempty( )`.
- Call the constructor exactly once for each System object.
- Set arguments to System object constructors as compile-time constants.
- Use the object constructor to set System object properties because you cannot use dot notation for code generation. You can use the `get` function to display properties.
- Test your code in simulation before generating code.

The following shows an example of some of these rules.

```
% Note: System Objects created for Codegen have to be persistent variables.
persistent vid;

% Construct the IMAQ VideoDevice System Object.
if isempty(vid)
    % Note: All required parameters must be passed to the System Object at
    % the point of construction.
    vid = imaq.VideoDevice('winvideo', 1, 'MJPG_320x240', ...
                           'ROI', [1 1 320 240], ...
                           'ReturnedColorSpace', 'rgb', ...
                           'DeviceProperties.Brightness', 130, ...
                           'DeviceProperties.Sharpness', 220);
end
```

Limitations on Using System Objects in Generated MATLAB Code

Ensure that the value assigned to a nontunable or public property is a constant and that there is at most one assignment to that property (including the assignment in the constructor). Do not set any properties during code generation.

The only System object functions supported in code generation are:

- `get`
- `getNumInputs`
- `getNumOutputs`
- `reset`
- `step`

Do not set System objects to become outputs from the MATLAB Function block.

Do not pass a System object as an example input argument to a function being compiled with `codegen`.

Do not pass a System object to functions declared as extrinsic (i.e., functions called in interpreted mode) using the `coder.extrinsic` function. Do not return System objects from any extrinsic functions.

15

Adding Support for Additional Hardware

Support for Additional Hardware

The Image Acquisition Toolbox software supports connections with hardware from many common vendors, but it might not support the hardware you use. To add support for your hardware, you can create an adaptor using the Image Acquisition Toolbox Adaptor Kit.

The Image Acquisition Toolbox Adaptor Kit is a C++ framework that you can use to implement an adaptor. An adaptor is a dynamic link library (DLL) that implements the connection between the Image Acquisition Toolbox engine and a device driver via the vendor's SDK API. When you use the Adaptor Kit framework, you can take advantage of many prepackaged toolbox features such as disk logging, multiple triggering modes, and a standardized interface to the image acquisition device.

After you create your adaptor DLL and register it with the toolbox using the `imaqregister` function, you can create a video input object to connect with a device through your adaptor. In this way, adaptors enable the dynamic loading of support for hardware without requiring recompilation and linking of the toolbox.

To build an adaptor requires familiarity with C++, knowledge of the application programming interface (API) provided by the manufacturer of your hardware, and familiarity with Image Acquisition Toolbox concepts, functionality, and terminology. To learn more about creating an adaptor, see “Creating Custom Adaptors”. For detailed information about the adaptor kit framework classes, see the *Image Acquisition Toolbox Adaptor Kit Class Reference*, which is available in

```
matlabroot\toolbox\imaq\imaqadaptors\kit\doc\adaptorkit.chm
```

where *`matlabroot`* represents your MATLAB installation directory.

Troubleshooting

This chapter provides information about solving common problems you might encounter with the Image Acquisition Toolbox software and the video acquisition hardware it supports.

Troubleshooting Overview

If, after installing the Image Acquisition Toolbox software and using it to establish a connection to your image acquisition device, you are unable to acquire data or encounter other problems, try these troubleshooting steps first. They might help fix the problem.

1. Verify that your image acquisition hardware is functioning properly.
2. If the hardware is functioning properly, verify that you are using a hardware device driver that is compatible with the Image Acquisition Toolbox software.

The following sections describe how to perform these steps for the vendors and categories of devices supported by the Image Acquisition Toolbox software.

If you are encountering problems with the preview window, see “Video Preview Window Troubleshooting” on page 16-32.

Note To see the full list of hardware that the toolbox supports, visit the Image Acquisition Toolbox product page at the MathWorks website `www.mathworks.com/products/image-acquisition`.

Note With previous versions of the Image Acquisition Toolbox, the files for all of the adaptors were included in your installation. Starting with version R2014a, each adaptor is available separately through support packages. In order to use the Image Acquisition Toolbox, you must install the adaptor that your camera uses. See “Image Acquisition Support Packages for Hardware Adaptors” on page 4-2 for information about installing the adaptors using MATLAB Add-Ons.

DALSA Sapera Hardware

In this section...

Troubleshooting DALSA Sapera Devices

The Image Acquisition Toolbox software supports the use of DALSA Sapera hardware.

If you are having trouble using the Image Acquisition Toolbox software with a supported DALSA Sapera frame grabber, try the following:

1. Install the Image Acquisition Toolbox Support Package for DALSA Sapera Hardware.

   Starting with version R2014a, each adaptor is available separately through MATLAB Add-Ons. See "Image Acquisition Support Packages for Hardware Adaptors" on page 4-2 for information about installing the adaptors.
2. Install the third-party hardware device driver separately and verify that it is compatible with Image Acquisition Toolbox. For the correct driver information, check the list of supported drivers for your MATLAB release under **Third-Party Requirements** on Teledyne DALSA Support from Image Acquisition Toolbox.

   Note Image Acquisition Toolbox is compatible only with specific driver versions provided with the DALSA Sapera hardware and is not guaranteed to work with any other versions.

   Find out the driver version you are using on your system. To learn how to get this information, see "Determining the Driver Version for DALSA Sapera Devices" on page 16-4.

   If you discover that you are using an unsupported driver version, visit the Teledyne DALSA website to download the correct driver.
3. Use the `imaqhwinfo` function to verify if `'dalsa'` is listed.

   ```
   imaqhwinfo
     ans =
       struct with fields:
       InstalledAdaptors: {'dalsa'  'winvideo'}
           MATLABVersion: '9.4 (R2018a)'
             ToolboxName: 'Image Acquisition Toolbox'
          ToolboxVersion: '5.3 (R2018a)'
   ```

   In the example shown, the DALSA and WinVideo adaptors are present, meaning the DALSA and OS Generic Video Interface support packages are installed.
4. Verify that you have a supported frame grabber. See a list of supported frame grabbers on Teledyne DALSA Support from Image Acquisition Toolbox.
5. Verify that your image acquisition hardware is functioning properly.

   For DALSA Sapera devices, run the application that came with your hardware, the Sapera CamExpert, and verify that you can view a live video stream from your camera.

If you are using a camera file to configure the device, verify that the toolbox can locate your camera file. Make sure that your camera appears in the **Camera** list in the Sapera CamExpert. To test the camera, select the camera in the list, and click the **Grab** button.

6 Make sure no other application is using the camera.

Determining the Driver Version for DALSA Sapera Devices

To determine the DALSA Sapera Library version you are using, view the release notes for the driver. You can access the release notes through the Windows **Start** menu.

1 Click the **Start** button to open the **Start** menu.

2 Select **Programs > DALSA Coreco Imaging > Sapera LT** to open the **Sapera LT** menu.

3 Select **Readme** to view the Sapera release notes.

DCAM IEEE 1394 (FireWire) Hardware on Windows

In this section...

Troubleshooting DCAM IEEE 1394 Hardware on Windows

If you are having trouble using the Image Acquisition Toolbox software with an IEEE 1394 (FireWire) camera using the toolbox's `dcam` adaptor, try the following:

1. Install the Image Acquisition Toolbox Support Package for DCAM Hardware.

   Starting with version R2014a, each adaptor is available separately through MATLAB Add-Ons. See “Image Acquisition Support Packages for Hardware Adaptors” on page 4-2 for information about installing the adaptors.
2. When installing the CMU 1394 Digital Camera Driver Setup, on the first page of the installation wizard, under **Select components to install**, select these three items in the installation list, and click **Next**.

   - Program Group and Desktop Shortcuts
   - Disable Default Windows Driver
   - Update Driver for Attached Devices
3. Verify that your IEEE 1394 (FireWire) camera is plugged into the IEEE 1394 (FireWire) port on your computer and is powered up.
4. Confirm that another imaging application is not connected to the camera.
5. Confirm that the camera is detected by "1394Camera Demo 64-bit" installed with the CMU DCAM driver. See “Running the CMU Camera Demo Application on Windows” on page 16-6.

   Verify that your IEEE 1394 (FireWire) camera can be accessed through the `dcam` adaptor.

   - Make sure the camera is compliant with the IIDC 1394-based Digital Camera (DCAM) specification. Vendors typically include this information in documentation that comes with the camera. If your digital camera is not DCAM compliant, you might be able to use the `winvideo` adaptor. See “Windows Video Hardware” on page 16-25 for information.
   - Make sure the camera outputs data in uncompressed format. Cameras that output data in Digital Video (DV) format, such as digital camcorders, cannot use the `dcam` adaptor. To access these devices, use the `winvideo` adaptor. See “Windows Video Hardware” on page 16-25 for information.
   - Make sure you specified the `dcam` adaptor when you created the video input object. Some IEEE 1394 (FireWire) cameras can be accessed through either the `dcam` or `winvideo` adaptors. If you can connect to your camera from the toolbox but cannot access some camera features, such as hardware triggering, you might be accessing the camera through a DirectX® driver. See “Creating a Video Input Object” on page 5-7 for more information about specifying adaptors.
   - If the demo application does not recognize the camera, install the CMU DCAM driver. See “Manually Installing the CMU DCAM Driver on Windows” on page 16-6 for instructions.

6 Verify that your CMU 1394 Digital Camera driver is version 6.4.6. The "CMU 1394 Digital Camera Driver" installed version can be found from **Windows Control Panel > Programs > Programs and Features**.

7 Confirm that the IEEE 1394 (FireWire) card is detected and drivers are correctly installed in Windows Device Manager.

Manually Installing the CMU DCAM Driver on Windows

The Image Acquisition Toolbox software supports acquiring data from IEEE 1394 (FireWire) cameras that support the IIDC 1394-based Digital Camera (DCAM) specification. To use a DCAM-compliant camera, you must use the DCAM driver created by Carnegie Mellon University (CMU) to connect to these devices.

Note The CMU DCAM driver is the only DCAM driver supported by the toolbox. You cannot use vendor-supplied drivers, even if they are compliant with the DCAM specification.

Installing the Driver

To install the CMU DCAM driver on your system, follow this procedure:

1 Obtain the CMU DCAM driver files. The Image Acquisition Toolbox software includes the CMU DCAM installation file, `1394camera646.exe`, in the directory

```
matlabroot\toolbox\imaq\imaqextern\drivers\win64\dcam
```

where *`matlabroot`* represents the name of your MATLAB installation directory.

You can also download the DCAM driver directly from CMU. Go to the website `www.cs.cmu.edu/~iwan/1394` and click the download link.

2 Start the installation by double-clicking the .exe file.

On the first page of the installation wizard under **Select components to install**, select these three items in the installation list and click **Next**.

- Program Group and Desktop Shortcuts
- Disable Default Windows Driver
- Update Driver for Attached Devices

On the second page of the wizard, accept the default location or browse to a new one, and click **Install**.

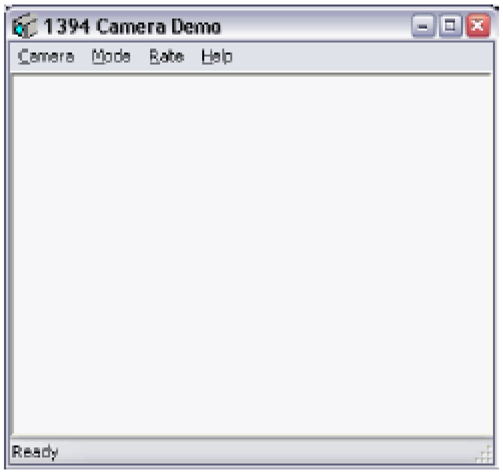
Note You may need to have your camera recognized after installing the driver. If this happens, open **Device Manager** and select the camera software. Right-click it, and choose **Update Driver Software**. Browse for the vendor driver software, and install it.

Running the CMU Camera Demo Application on Windows

The Carnegie Mellon University (CMU) DCAM driver distribution includes a camera demo application, named `1394CameraDemo.exe`. The demo application is among the files you installed in the previous section.

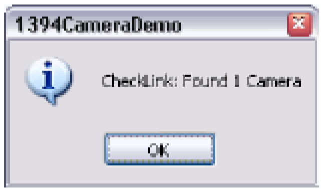
You can use this demo application to verify whether your camera is using the CMU DCAM driver. To access a camera through this demo application:

1. Select **Start > Programs > CMU 1394 Camera > 1394 Camera Demo**.
2. The application opens a window, shown in the following figure.



3. From the Camera Demo application, select **Camera > Check Link**. This option causes the demo application to look for DCAM-compatible cameras that are available through the IEEE 1394 (FireWire) connection.

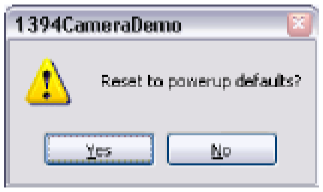
   The demo application displays the results of this search in a pop-up message box. In the following example, the demo application found a camera. Click **OK** to continue.



1394CameraDemo

CheckLink: Found 1 Camera

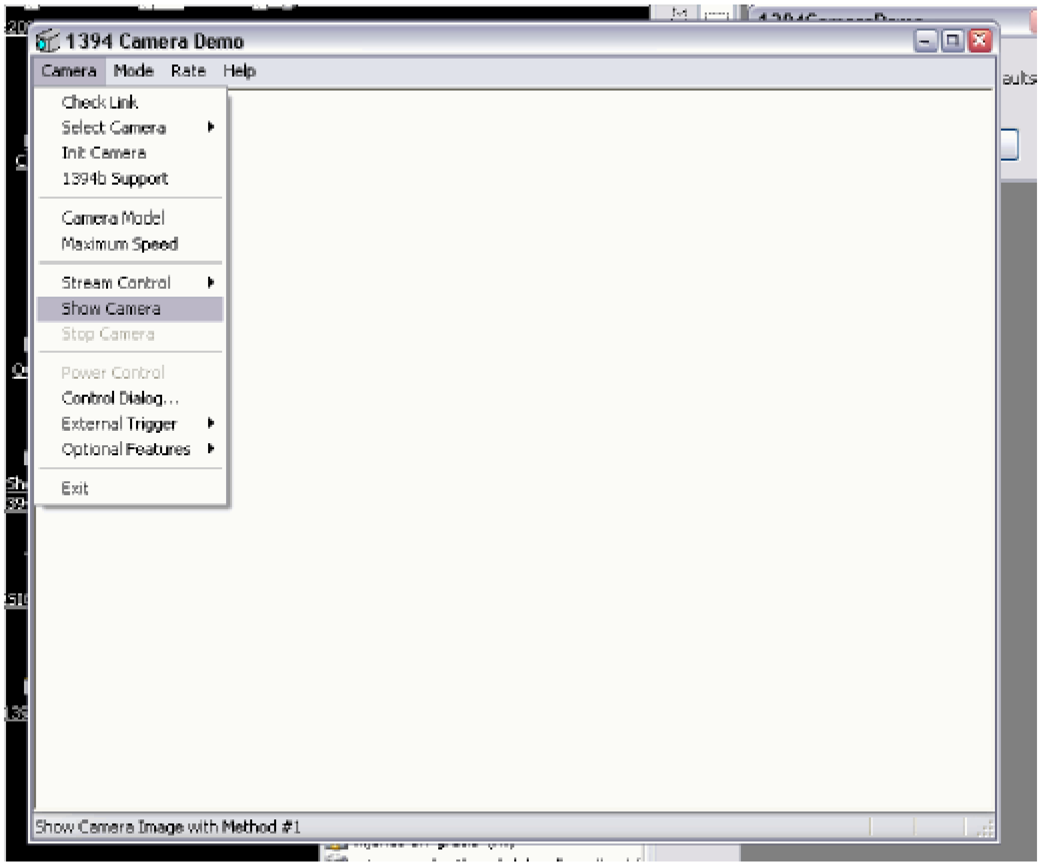
4. Select **Camera > Select Camera**, and select the camera you want to use. The **Select Camera** option is not enabled until after the **Check Link** option has successfully found cameras.
5. Select **Camera > Init Camera**. In this step, the demo application checks the values of various camera properties. The demo application might resize itself to fit the video format of the specified camera. If you see the following dialog box message, click **Yes**.



1394CameraDemo

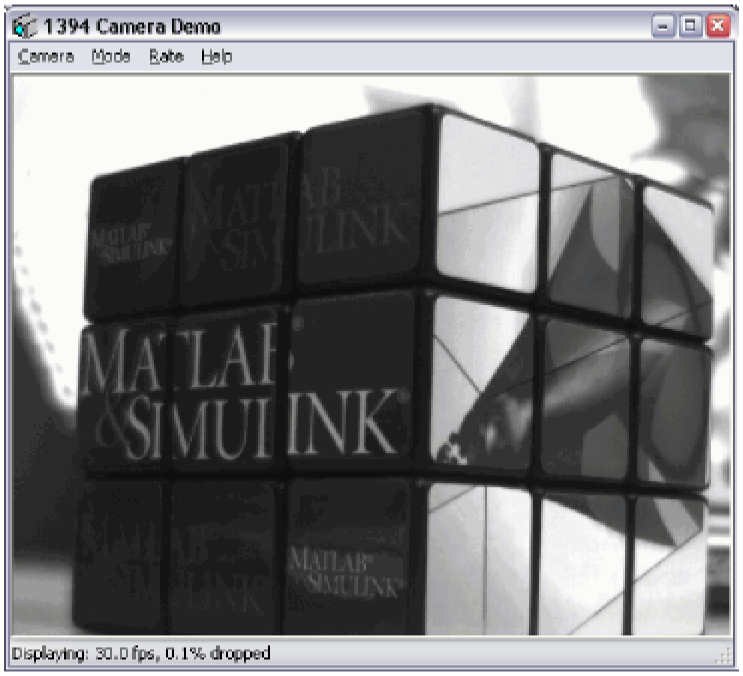
Reset to powerup defaults?

6. Select **Camera > Show Camera** to start acquiring video.



Show Camera Image with Method #1

The demo application starts displaying live video in the window.



Displaying: 30.0 fps, 0.1% dropped

7 To exit, select **Stop Camera** from the Camera menu, and click **Exit**.

Matrox Hardware

In this section...
"Troubleshooting Matrox Devices" on page 16-10
"Determining the Driver Version for Matrox Devices" on page 16-10

Troubleshooting Matrox Devices

Device Discovery

If you are having trouble using the Image Acquisition Toolbox software with a supported Matrox frame grabber, try the following:

1. Install the Image Acquisition Toolbox Support Package for Matrox Hardware.

   Starting with version R2014a, each adaptor is available separately through MATLAB Add-Ons. See "Image Acquisition Support Packages for Hardware Adaptors" on page 4-2 for information about installing the adaptors.
2. Install the Matrox Imaging Library (MIL) separately and verify that it is the supported version for your MATLAB release. Check the supported versions in the **Third-Party Requirements** section on Matrox Frame Grabber Support from Image Acquisition Toolbox. If you discover that you are using an unsupported driver version, visit the Matrox website to download the correct drivers.

   Note that the `matroxcam` and `matroxlist` functions that are available with the `matroxcam` object starting in release R2015a work with MIL or MIL-Lite 10.x only. Use of the `matrox` adaptor with the `videoinput` object can use either MIL 9 or 10.

   Find out the driver version you are using on your system. To learn how to get this information, see "Determining the Driver Version for Matrox Devices" on page 16-10.
3. Verify that you are using supported image acquisition hardware and that it is functioning properly. See a list of supported Matrox frame grabbers on Matrox Frame Grabber Support from Image Acquisition Toolbox.

   For Matrox devices, run the application that came with your hardware, Matrox Intellicam, and verify that you can receive live video.
4. Make sure no other application is accessing the frame grabber.

Note There is no difference between MIL and MIL-Lite software inside of MATLAB. They both work with Matrox Imaging devices.

Determining the Driver Version for Matrox Devices

To determine the Matrox Imaging Library version you are using, run the Matrox MIL Configuration utility. You can access this software through the Windows **Start** button.

Select **Start > Programs > Matrox Imaging Products > MIL Configuration**.

The software version is listed on the **Information** tab.

Note As of version R2014b, the Image Acquisition Toolbox supports MIL 10, and that is the recommended version to use.

National Instruments Hardware

In this section...
"Troubleshooting National Instruments Devices" on page 16-12
"Determining the Driver Version for National Instruments Devices" on page 16-13

Troubleshooting National Instruments Devices

If you are having trouble using the Image Acquisition Toolbox software with a supported National Instruments frame grabber, try the following:

1. Install the Image Acquisition Toolbox Support Package for National Instruments Hardware.

   Starting with version R2014a, each adaptor is available separately through MATLAB Add-Ons. See "Image Acquisition Support Packages for Hardware Adaptors" on page 4-2 for information about installing the adaptors.
2. Verify that your image acquisition hardware is functioning properly.

   For National Instruments devices, run the application that came with your hardware, NI MAX (Measurement & Automation Explorer), and verify that you can receive live video.

   Launch NI MAX from **Start > Programs > National Instruments > NI MAX**.

   To test that the hardware is working properly, in Measurement & Automation Explorer, expand **Devices and Interfaces**, and then expand **NI-IMAQ Devices**, and expand the node that represents the board you want to use.

   With the board expanded, select the channel or port that you have connected a camera to.
3. Click the **Grab** button to verify that your camera is working. If it is not, see the National Instruments device documentation.
4. Confirm that the NI-IMAQ vendor driver version installed is supported by Image Acquisition Toolbox.

   Note The Image Acquisition Toolbox software is compatible only with specific driver versions provided with the National Instruments software and is not guaranteed to work with any other versions.

   - Find out the driver version you are using on your system. To learn how to get this information, see Determining the Driver Version on page 16-13.
   - Verify that the version is compatible with the Image Acquisition Toolbox software. Supported NI-IMAQ driver versions are listed on the following hardware support web page: (`https://www.mathworks.com/hardware-support/national-instruments.html`).
5. If you discover that you are using an unsupported driver version:

   - Uninstall any existing NI-IMAQ driver version from **Windows Control Panel > Programs and Features > National Instruments Software**.
   - Reinstall the Image Acquisition Toolbox Support Package for National Instruments Hardware.

Determining the Driver Version for National Instruments Devices

To determine the National Instruments driver version you are using, run the Measurement & Automation Explorer.

Select **Help > System Information**, and then see the **NI-IMAQ Software** field for the driver version number.

Point Grey Hardware

In this section...
“Device Discovery” on page 16-14
“Troubleshooting Point Grey Devices” on page 16-15
“Determining the Driver Version for Point Grey Devices” on page 16-16

Device Discovery

1. Install the Image Acquisition Toolbox Support Package for Point Grey Hardware.

   Starting with version R2014a, each adaptor is available separately through MATLAB Add-Ons. See “Image Acquisition Support Packages for Hardware Adaptors” on page 4-2 for information about installing the adaptors.
2. Verify that you have the correct FlyCapture version for the release.

   Check the list of supported drivers on the Point Grey Camera Support from Image Acquisition Toolbox product page.

   Uninstall any unsupported versions of FlyCapture SDK/Viewer using **Control Panel > Add/Remove Programs**.
3. Verify that the camera is working with the FlyCapture Viewer.

   Select **Start > Programs > Point Grey FlyCapture2 > Point Grey FlyCap2**.

   In FlyCapture, select your device, and click **OK** to open the dialog box that shows the video feed to test that the hardware is working properly.
4. If you are using a Point Grey camera that is a GigE Vision device, do not try to use both the Point Grey adaptor and the GigE Vision adaptor at the same time. You should use the Point Grey adaptor.
5. Ensure that the FlyCapture driver is on the PATH Environment variable. To view the PATH:

   - Click the **Start** button, right-click the **Computer** option in the **Start** menu, and select **Properties**.
   - Click the **Advanced System Settings** link in the left column.
   - In the **System Properties** window, click on the **Advanced** tab, and click the **Environment Variables** button near the bottom of that tab.
   - In the **Environment Variables** window, highlight the **Path** variable in the **System variables** section, and click the **Edit** button. Ensure that `'%FC2PATH%'` and `'%FC2PATH\vs2013'` are on the path.
   - If you are having trouble detecting your device in MATLAB, put both variables in the start of your PATH.
6. Make sure that the FlyCapture DirectShow drivers are not installed.
7. If you are using a GigE Point Grey camera, you must configure it using FlyCapture Viewer before using it with MATLAB.

   - Connect the GigE Point Grey camera to your system.

- Launch FlyCapture Viewer.
- Select the camera that you connected in the list of devices.
- Click the **Force IP** button.

Troubleshooting Point Grey Devices

The Point Grey adaptor includes support for the following types of Point Grey devices:

- USB 3
- FireWire
- GigE Vision
- USB 2
- Bumblebee 2

If you are having trouble using the Image Acquisition Toolbox software with a supported Point Grey camera, try the following:

1. Verify that your image acquisition hardware is functioning properly.

   For Point Grey devices, run the application that came with your hardware, FlyCapture, and verify that you can receive live video.
2. Select **Start > Programs > Point Grey FlyCapture2 > Point Grey FlyCap2**.
3. In FlyCapture, select your device, and click **OK** to open the dialog box that shows the video feed to test that the hardware is working properly.
4. Install the Image Acquisition Toolbox Support Package for Point Grey Hardware.
5. If your hardware is functioning properly, verify that you are using a hardware device driver that is compatible with the toolbox.

   Note The Image Acquisition Toolbox software is compatible only with specific driver versions provided with the Point Grey software and is not guaranteed to work with any other versions.

   - Find out the driver version you are using on your system. To learn how to get this information, see “Determining the Driver Version for Point Grey Devices” on page 16-16.
   - Verify that the version is compatible with the Image Acquisition Toolbox software. For the correct driver information, check the list of supported drivers on the Point Grey Camera Support from Image Acquisition Toolbox

   If you discover that you are using an unsupported driver version, visit the FlyCapture SDK website to download the correct drivers.

Note If you are using a Point Grey camera that is a GigE Vision device, do not try to use both the Point Grey adaptor and the GigE Vision adaptor at the same time. You should use the Point Grey adaptor.

Note When using the Bumblebee 2 cameras, certain video formats may be suppressed. To see the available video formats for your Bumblebee camera, open the **Image Acquisition Explorer** (using the `imageAcquisitionExplorer` function), select your camera, and check the options available for the **Video Format** parameter in the app toolstrip.

Determining the Driver Version for Point Grey Devices

To determine the Point Grey driver version you are using, run the Point Grey FlyCapture utility.

To see the driver version number:

1. Select **Start > Programs > Point Grey FlyCapture2 > Point Grey FlyCap2** to open FlyCapture.
2. The driver number appears on the banner of the FlyCapture dialog box.

Kinect for Windows Hardware

The Kinect adaptor is supported on 64-bit Windows.

Device Discovery

If you are having trouble using the Image Acquisition Toolbox software with a supported Kinect for Windows sensor, try the following:

1. Install the Image Acquisition Toolbox Support Package for Kinect for Windows Sensor.

   Starting with version R2014a, each adaptor is available separately through MATLAB Add-Ons. See “Image Acquisition Support Packages for Hardware Adaptors” on page 4-2 for information about installing the adaptors.

   If you have problems installing the Kinect support package, see the section below, “Troubleshooting the Support Package Installation.”
2. Verify that you are using one of the supported Kinect hardware models. See Microsoft Kinect for Windows Support from Image Acquisition Toolbox for a list of supported hardware. The Kinect for Xbox 360 is not supported.
3. Verify that you have the supported Kinect driver version. You must have version 1.6 to use V1 devices, and version 2.0 for Kinect V2 and Xbox One devices.
4. Verify that your image acquisition hardware is functioning properly.

   For Kinect for Windows devices, you can run the Kinect Explorer application for Kinect V1 devices, and verify that you can receive live video. You can run Kinect Evolution for Kinect V2 or Xbox One and adapter.
5. For Kinect V1, make sure no other application is accessing the Kinect v1.
6. For Kinect V2, make sure it is plugged into a USB 3.0 port, and you are using an OS that is Windows 8 or later.

System Requirements for the Kinect V2 Sensor

The Kinect V2 sensor requires the following:

- 64-bit (x64) processor
- Physical dual-core 3.1 GHz (2 logical cores per physical system) or faster processor
- USB 3.0 controller dedicated to the Kinect for Windows v2 sensor or the Kinect Adapter for Windows for use with the Kinect for Xbox One sensor
- 4 GB of RAM
- Graphics card that supports DirectX 11
- Windows 8 or 8.1, Windows Embedded 8, or Windows 10

Troubleshooting the Support Package Installation

If the setup fails, it could be caused by an internet security setting. If you get an error message such as “KINECT Setup Failed – An error occurred while installing,” try the following, and then run the installer again.

1. In Internet Explorer, go to **Tools > Internet Options**.

2. In Internet Options, select the **Advanced** tab.
3. Under the **Security** subsection, uncheck **Check for publisher's certificate revocation** to temporarily disable it, and click **OK**.
4. Run the installer again.
5. After you have installed the support package, re-enable the security option in Internet Explorer.

GigE Vision Hardware

In this section...

Troubleshooting GigE Vision Devices on Windows

If you are having trouble using the Image Acquisition Toolbox with a GigE Vision camera on a Windows machine, using the toolbox's `gige` adaptor, try the following:

1. Install the Image Acquisition Toolbox Support Package for GigE Vision Hardware.

   Starting with version R2014a, each adaptor is available separately through MATLAB Add-Ons. See "Image Acquisition Support Packages for Hardware Adaptors" on page 4-2 for information about installing the adaptors.
2. Go through the configuration steps of the "GigE Vision Image Acquisition Quick Start Guide" (Image Acquisition Toolbox Support Package for GigE Vision Hardware).

   In particular, confirm that:

   The installed Ethernet network adapter driver is provided by the network adapter manufacturer (and is not a custom high-performance driver installed for use with a third-party imaging application).

   Any packet filtering drivers from a third-party imaging application or an antivirus program are disabled (unchecked) in the camera connection Network Settings.

   A firewall is not blocking communication with the camera.
3. Confirm that another imaging application is not connected to the camera.
4. To refresh the list of detected devices, execute `imaqreset` followed by `imaqhwinfo`.

   ```
   imaqreset
   imaqhwinfo('gige')
   ```

   When using the `gigecam` interface, use the `gigecamlist` command to show a list of the detected GigE Vision cameras:

   ```
   gigecamlist
   ```
5. Confirm that the camera is detected with other GigE Vision compliant imaging applications.
6. Confirm that there are no issues with the GenICam runtime libraries installation (such as a conflict with a third-party imaging application) by executing the `imaqsupport` command and checking for any error messages in the GENICAM section.

   ```
   imaqsupport
   ```
7. Certain camera vendor software setup programs also install DirectShow drivers for use with GigE Vision cameras. Uninstall the DirectShow drivers by using the vendor's software setup program, as these DirectShow drivers might cause issues with the camera being detected when using the `gige` adaptor.

Troubleshooting GigE Vision Devices on Linux

If you are having trouble using the Image Acquisition Toolbox with a GigE Vision camera on a Linux machine, using the toolbox's `gige` adaptor, try the following:

1. Install the Image Acquisition Toolbox Support Package for GigE Vision Hardware.

   Starting with version R2014a, each adaptor is available separately through MATLAB Add-Ons. See “Image Acquisition Support Packages for Hardware Adaptors” on page 4-2 for information about installing the adaptors.
2. Verify that the adaptor loads. You can use the `imaqhwinfo` command to list installed adaptors. The `gige` adaptor should be included on the list.

   If it does not load, make sure that GenICam is configured correctly using the `imaqsupport` function.

   If your camera requires a GenICam XML file on a local drive (most do not), and the adaptor loads but no devices are shown, check the `MWIMAQ_GENICAM_XML_FILES` environment variable, and make sure it contains the directory where your camera’s XML file is located.

   For information on installing GenICam and checking your environment variables, see “Software Configuration” on page 10-10.
3. Make sure you did not install your camera vendor's filtering or performance networking driver. If you did, you should uninstall it.
4. Make sure the Ethernet card is configured properly.

   For more information on this, see “Network Hardware Configuration Notes” on page 10-3 and “Network Adaptor Configuration Notes” on page 10-4.

   Also, if you have multiple cameras connected to multiple Ethernet cards, you cannot have them all set to automatic IP configuration. You must specify the IP address for each card and each card must be on a different subnet.
5. Examine the connectivity of your device separately from using the Image Acquisition Toolbox. You may find using `ping -b`, `arp`, `route`, and `ifconfig` helpful with this.
6. If your acquisition stops due to a dropped frame, you can set the `IgnoreDroppedFrames` property to `'on'` to continue your acquisition with dropped frames. When this property is `'on'`, the `NumDroppedFrames` property keeps track of the number of frames dropped while the acquisition is running.
7. You might receive an error message such as:

   “Block 23 is being dropped because packets are unavailable for resend.”

   If it does not mention buffer size, it is likely that packets are being dropped due to overload of the CPU. To lower the CPU load, raise the value of the `PacketSize` device-specific (`source`) property. In order to do this, you must be using hardware that supports jumbo frames.

   You might also want to calculate and set the `PacketDelay` device-specific (`source`) property.

   Also, if you are using a CPU that is older than an Intel Core® 2 Quad or equivalent AMD®, you might experience this type of error.

   If you have a slower computer and experience packet loss using the GigE Vision adaptor, you can set a packet delay to avoid overloading the computer. This action is useful in solving the performance issue if you cannot achieve your camera's frame rate. The `PacketDelay` property is initially set to use your camera's default value. You can then adjust the value if needed. The `TimeStampTickFrequency` property is read-only, but is available for calculating the actual packet delay value is being used.

For more information on the new `PacketDelay` property and how to calculate packet delay, see this solution:

https://www.mathworks.com/support/solutions/en/data/1-F36R0R/index.html

8. If you are able to start acquisition without error but do not receive any frames, and if you are using a larger `PacketSize`, make sure that your hardware and the network between the computer and the camera support jumbo frames, and also that your Ethernet interface is set to allow them at the size that you are attempting to use.
9. If you receive an error that says a block or frame is being dropped because a packet is unavailable for resend, one likely cause is that the buffer size of the socket could not be set to the reported value, for example `1000000`.

   See your system administrator about using `sysctl` for `net.core.rmem_max`. For example, the system administrator could try:

   ```
   sysctl -w net.inet.udp.recvspace=1000000
   ```
10. If your camera does not start a new acquisition at block `1`, the toolbox attaches the block ID (frame ID) as metadata to the frame. If you want to know if you lost initial frames, you can check the metadata. If the first frame's block ID is not `1`, you may have some missing frames. For example, use this command in MATLAB:

    ```
    [d t m]=getdata(vid,2);
    m(1)
    ```

    The answer includes the `Block ID` and the `FrameNumber`.
11. Run the `imaqsupport` function for further troubleshooting information.

Troubleshooting GigE Vision Devices on macOS

If you are having trouble using the Image Acquisition Toolbox software with a GigE Vision camera on a macOS machine using the toolbox's `gige` adaptor, try the following:

1. Install the Image Acquisition Toolbox Support Package for GigE Vision Hardware.

   Starting with version R2014a, each adaptor is available separately through MATLAB Add-Ons. See "Image Acquisition Support Packages for Hardware Adaptors" on page 4-2 for information about installing the adaptors.
2. Verify that the adaptor loads. You can use the `imaqhwinfo` command to list installed adaptors. The `gige` adaptor should be included on the list.

   If it does not load, make sure that GenICam is installed and the environment variables exist. You can check this using the `imaqsupport` function.

   If your camera requires a GenICam XML file on a local drive (most do not), and the adaptor loads but no devices are shown, check the `MWIMAQ_GENICAM_XML_FILES` environment variable, and make sure it contains the directory where your camera's XML file is located.

For information on installing GenICam and checking your environment variables, see “Software Configuration” on page 10-10.

3 Make sure you did not install your camera vendor's filtering or performance networking driver. If you did, uninstall it.

4 Make sure the Ethernet card is configured properly.

For more information on this, see “Network Hardware Configuration Notes” on page 10-3 and “Network Adaptor Configuration Notes” on page 10-4.

Also, if you have multiple cameras connected to multiple Ethernet cards, you cannot have them all set to automatic IP configuration. You must specify the IP address for each card, and each card must be on a different subnet.

5 Examine the connectivity of your device separately from using the Image Acquisition Toolbox. You may find using `ping -b`, `arp`, `route`, and `ifconfig` helpful with this process.

6 If your acquisition stops due to a dropped frame, you can set the `IgnoreDroppedFrames` property to `'on'` to continue your acquisition with dropped frames. When this property is `'on'`, the `NumDroppedFrames` property keeps track of the number of frames dropped while the acquisition is running.

7 You might receive an error message such as the following:

“Block 23 is being dropped because packets are unavailable for resend”.

If it does not mention buffer size, it is likely that packets are being dropped due to overload of the CPU. To lower the CPU load, raise the value of the `PacketSize` device-specific (`source`) property. In order to do this, you must be using hardware that supports jumbo frames.

You might also want to calculate and set the `PacketDelay` device-specific (`source`) property.

Also, if you are using a CPU that is older than an Intel Core 2 Quad or equivalent AMD, you might experience this type of error.

If you have a slower computer and experience packet loss using the GigE Vision adaptor, you can set a packet delay to avoid overloading the computer. This setting is useful in solving the performance issue if you cannot achieve your camera’s frame rate. The `PacketDelay` property is initially set to use your camera’s default value. You can then adjust the value if needed. The `TimeStampTickFrequency` property is read-only but is available for calculating the actual packet delay value is being used.

For more information on the new `PacketDelay` property and how to calculate packet delay, see this solution:

https://www.mathworks.com/support/solutions/en/data/1-F36R0R/index.html

8 If you are able to start acquisition without error but do not receive any frames, and if you are using a larger `PacketSize`, make sure that your hardware and the network between the computer and the camera support jumbo frames, and also that your Ethernet interface is set to allow them at the size that you are attempting to use.

9 If you receive an error that says a block or frame is being dropped because a packet is unavailable for resend, one likely cause is that the buffer size of the socket could not be set to the reported value, for example `1000000`.

See your system administrator about using `sysctl` for `net.core.rmem_max`. For example, the system administrator could try:

```
sysctl -w net.inet.udp.recvspace=1000000
```

10 If your camera does not start a new acquisition at block `1`, the toolbox attaches the block ID (frame ID) as metadata to the frame. If you want to know if you lost initial frames, you can check the metadata – if the first frame's block ID is not `1`, you may have some missing frames. For example, use this command in MATLAB:

```
[d t m]=getdata(vid,2);
m(1)
```

The answer includes the `Block ID` and the `FrameNumber`.

11 Run the `imaqsupport` function for further troubleshooting information.

GenICam GenTL Hardware

Device Discovery

If you are having trouble using the Image Acquisition Toolbox with a GenICam GenTL camera driver using the toolbox's `gentl` adaptor, try the following:

1. Install the Image Acquisition Toolbox Support Package for GenICam Interface.

   Starting with version R2014a, each adaptor is available separately through MATLAB Add-Ons. See “Image Acquisition Support Packages for Hardware Adaptors” on page 4-2 for information about installing the adaptors.
2. Verify that the `gentl` adaptor loads. You can use the `imaqhwinfo` command to list installed adaptors. The `gentl` adaptor should be included on the list.

   If it does not load, make sure that GenICam is configured correctly using the `imaqsupport` function.

   For information on installing GenICam, see “Software Configuration” on page 10-10.
3. Install the manufacturer-provided GenTL producer. During setup, make sure to uncheck the DirectShow driver installation. If a device is configured for DirectShow, it might not be available to GenTL.
4. Make sure your environment variables are set. For example, depending on the GenTL producers you have installed, on a 64-bit Windows system, it could be something like:

   ```
   GENICAM_GENTL64_PATH=C:\Program Files\LeutronVision\Simplon\bin\cti;C:\Program
       Files\MATRIX VISION\mvIMPACT acquire\bin;C:\XIMEA\GenTL Producer\x86
   ```
5. Each directory that you list in the environment variables must contain a DLL file that has a `.cti` extension and that exports the standard C functions that are expected for a GenTL producer. The Image Acquisition Toolbox `gentl` adaptor scans these directories for all the CTI files and then checks whether they export the correct minimum set of functions.
6. Test the connectivity of your device separately from using the Image Acquisition Toolbox. Use the vendor program included with your device to see if you can detect and acquire images from the camera.
7. If you are using the GenICam GenTL adaptor with a GigE Vision camera, it may be that the producers for GigE Vision cameras do not send a `ForceIP` command. So sometimes, after plugging in a new camera, it is not found. Using the toolbox’s `gige` adaptor first can resolve this.
8. Run the `imaqsupport` function for further troubleshooting information. The GenTL section should list the detected producers, as follows.

   ```
   --------------------GENTL--------------------

   GENICAM_GENTL64_PATH = <C:\Program Files\MATRIX VISION\mvIMPACT Acquire\bin\x64;C:\Program> Files (x86)\Common Files

   Producer found = <C:\Program Files\MATRIX VISION\mvIMPACT Acquire\bin\x64\mvGenTLProducer.cti>

   No producers found in C:\Program Files (x86)\Common Files\MVS\Runtime\Win64_x64

   Producer found = <C:\XIMEA\GenTL Producer\x64\ximea.gentlX64.cti>
   ```

Windows Video Hardware

Troubleshooting Windows Video Devices

If you are having trouble using the Image Acquisition Toolbox software with a supported Windows video acquisition device, try the following:

1. Install the Image Acquisition Toolbox OS Generic Video Interface Support Package. It includes the necessary files to use the `winvideo` adaptor.

   Starting with version R2014a, each adaptor is available separately through MATLAB Add-Ons. See “Image Acquisition Support Packages for Hardware Adaptors” on page 4-2 for information about installing the adaptors.
2. Confirm that another imaging application is not connected to the camera.
3. Confirm that the camera and its supported video formats are detected by the standalone hardware detection utility `detectDevices.exe` (64-bit), which is installed in

   ```
   MATLABROOT\toolbox\imaq\imaqextern\utilities\detectDevices\win64
   ```

   where MATLABROOT is the MATLAB installation folder. For example:

   ```
   C:Program Files\MATLAB\R2018a\toolbox\imaq\imaqextern\utilities\detectDevices\win64
   ```

   The following is sample output when you run `detectDevices.exe` in this folder:

   ```
   Hardware detection application for the R2018a version of Image Acquisition Toolbox.
   Detecting hardware for the winvideo adaptor.

       Found device: IPEVO Point 2 View
                   Found format: YUY2_640x480
                   Found format: YUY2_320x240
                   Found format: YUY2_800x600
                   Found format: YUY2_1024x768
                   Found format: YUY2_1280x1024
                   Found format: YUY2_1600x1200
                   Found format: YUY2_640x480
                   Found format: MJPG_640x480
                   Found format: MJPG_320x240
                   Found format: MJPG_800x600
                   Found format: MJPG_1024x768
                   Found format: MJPG_1280x1024
                   Found format: MJPG_1600x1200
                   Found format: MJPG_640x480
   ```
4. Confirm that the camera is detected in other DirectShow compliant applications, such as VLC Media Player. Utilities such as GraphEdit or AmCap Sample, which are included with Microsoft Windows Software Development Kit (SDK), are also useful for troubleshooting DirectShow driver related issues.
5. Some camera and video capture hardware manufacturers provide DirectShow drivers that need to be installed in order to use the hardware with imaging applications on Windows. For example, Thorlabs and IDS Imaging are both hardware manufacturers that provide software packages that include DirectShow drivers. Make sure you install the 64-bit version of the DirectShow drivers, as MATLAB is a 64-bit application.

6 Some vendor-provided DirectShow drivers might need to be registered using a vendor-provided utility in order to use the hardware with imaging applications on Windows. Refer to the vendor-provided instructions.

Linux Video Hardware

Device Discovery for Linux Video Devices

If you have trouble using the Image Acquisition Toolbox with a supported Linux Video acquisition device, try the following:

1. Install the Image Acquisition Toolbox OS Generic Video Interface Support Package. It includes the necessary files to use the `linuxvideo` adaptor.

   Starting with version R2014a, each adaptor is available separately through MATLAB Add-Ons. See “Image Acquisition Support Packages for Hardware Adaptors” on page 4-2 for information about installing the adaptors.
2. Verify that you have a supported GStreamer version on the Third-Party Requirements section on the MathWorks website (`www.mathworks.com/hardware-support/gstreamer.html`).

   To find the version of GStreamer drivers installed on your system, run this command:

   ```
   dpkg -l libgst* | grep ^i
   ```

   If you had the GStreamer 1.0 libraries, it would return results similar to the following examples.

   GStreamer development files for libraries from the "bad" set:

   ```
   ii libgstreamer-plugins-bad1.0-0:amd64 1.4.4-2.1+b1 amd64
   ```

   GStreamer libraries from the "base" set:

   ```
   ii libgstreamer-plugins-base1.0-0:amd64 1.4.4-2 amd64
   ```

   GStreamer development files for libraries from the "base" set :

   ```
   ii libgstreamer-plugins-base1.0-dev 1.4.4-2 amd64
   ```

   Core GStreamer libraries and elements:

   ```
   ii libgstreamer1.0-0:amd64 1.4.4-2 amd64
   ```

   Core GStreamer libraries and elements:

   ```
   ii libgstreamer1.0-0-dbg:amd64 1.4.4-2 amd64
   ```

   Core GStreamer development files:

   ```
   ii libgstreamer1.0-dev 1.4.4-2 amd64
   ```
3. Verify that your camera can be detected and images can be acquired by running applications such as Cheese or guvcview. If you can start the utility, run the utility, and close it without encountering any errors, the toolbox should be able to operate with your image acquisition device. If you encounter errors, resolve them before attempting to use the toolbox with the device.
4. If the camera has a USB interface, make sure `lsusb` on a Linux terminal can detect your camera.
5. For releases R2017a and later, MathWorks supports the GStreamer 1.0 drivers.

Note The Linux Video driver is a generic interface, and you should only use it if you do not have a more specific driver to use with your device. If your device is an IEEE 1394 IIDC compliant device, use the DCAM (`dcam`) adaptor. If your device is GigE Vision compliant, use the GigE (`gige`) adaptor. If your device manufacturer provides a GenTL producer, use the GenTL (`gentl`) adaptor. Use the Linux Video driver only if there is no more specific option for your device.

Linux DCAM IEEE 1394 Hardware

Troubleshooting Linux DCAM Devices

If you are having trouble using the Image Acquisition Toolbox with a supported Linux DCAM IEEE 1394 hardware acquisition device, try the following:

1. Verify that your IEEE 1394 (FireWire) camera can be accessed through the `dcam` adaptor.

   - Make sure the camera is compliant with the IIDC 1394-based Digital Camera (DCAM) specification. Vendors typically include this information in documentation or data sheet that comes with the camera. If your digital camera is not DCAM compliant, you should be able to use the Linux Video adaptor.
2. Install the Image Acquisition Toolbox Support Package for DCAM Hardware.

   Starting with version R2014a, each adaptor is available separately through MATLAB Add-Ons. See "Image Acquisition Support Packages for Hardware Adaptors" on page 4-2 for information about installing the adaptors.
3. Verify that your image acquisition hardware is functioning properly and that you have permission to access it.

   Be sure that your system and login have the proper permissions to access the hardware. See your system administrator if you need help.

   You can verify that your hardware is functioning properly by running Coriander. See your system administrator if you need help installing Coriander.

   If you can start the utility, run the utility, and close the utility without encountering any errors, the toolbox should be able to operate with your image acquisition device. If you encounter errors, resolve them before attempting to use the toolbox with the device.
4. To use DCAM on Linux, you need to have installed the libdc1394-22 package, as well as the libraw1394-11.

Macintosh Video Hardware

Troubleshooting Macintosh Video Devices

Device Discovery

If you are having trouble using the Image Acquisition Toolbox with a supported Macintosh video acquisition device, try the following:

1. Install the Image Acquisition Toolbox OS Generic Video Interface Support Package. It includes the necessary files to use the `macvideo` adaptor.

   Starting with version R2014a, each adaptor is available separately through MATLAB Add-Ons. See “Image Acquisition Support Packages for Hardware Adaptors” on page 4-2 for information about installing the adaptors.
2. Verify that you have supported libraries that work with the camera. Make sure you have Apple AV Foundation Libraries installed on your computer. If you do not have it installed, you can download it.

   The OS Generic Video Adaptor on Mac (`macvideo`) uses Apple AV Foundation Libraries starting with R2016b. The use of the Macintosh Video adaptor previously required that you have QuickTime installed on your computer. QuickTime is no longer required.
3. Verify that your camera can be detected and images can be acquired by running applications that come with Macintosh OS, such as Photo Booth, iMovie or FaceTime. If you can start the utility, run the utility, and close the utility without encountering any errors, then the toolbox should be able to operate with your image acquisition device. If you encounter errors, resolve them before attempting to use the toolbox with the device.

Note The Macintosh Video Adaptor is a generic interface and should only be used if you do not have a more specific adaptor to use with your device. If your device is a DCAM or FireWire IIDC device, use the DCAM (`dcam`) adaptor. If your device is GigE Vision compliant, use the GigE (`gige`) adaptor. Use the Macintosh Video Adaptor only if there is no more specific option for your device.

Macintosh DCAM IEEE 1394 Hardware

Troubleshooting Macintosh DCAM Devices

If you are having trouble using the Image Acquisition Toolbox with a supported Macintosh DCAM IEEE 1394 hardware acquisition device, try the following:

1. Verify that your IEEE 1394 (FireWire) camera can be accessed through the `dcam` adaptor.

   - Make sure the camera complies with the IIDC 1394-based Digital Camera (DCAM) specification. Vendors typically include this information in documentation that comes with the camera. If your digital camera is not DCAM compliant, you might be able to use the Macintosh Video Adaptor.

2. Install the Image Acquisition Toolbox Support Package for DCAM Hardware.

   Starting with version R2014a, each adaptor is available separately through MATLAB Add-Ons. See “Image Acquisition Support Packages for Hardware Adaptors” on page 4-2 for information about installing the adaptors.

3. Verify that your image acquisition hardware is functioning properly.

   You can verify that your hardware is functioning properly by running an external webcam application, for example, Photo Booth or iMovie.

   If you can start the utility, run the utility, and close the utility without encountering any errors, then the toolbox should be able to operate with your image acquisition device. If you encounter errors, resolve them before attempting to use the toolbox with the device.

Video Preview Window Troubleshooting

When previewing the video stream, if you encounter a problem, try one of the following solutions.

Problem	Possible Solutions
Video Preview window stops running.	• Close the preview window and reopen it. • Verify that your image acquisition device is working properly. Close MATLAB and run the application that came with your device. • Make sure no other application is using the device.
Video Preview window displays a white window with a red X.	• Close the preview window and reopen it. • Make sure no other application is using the device. • If you are using a GigE Vision camera (either the `gige` adaptor with the `videoinput` object or the `gigecam` object), you may need to disable your firewall. See “GigE Vision Hardware” on page 16-19 for more information.
Video Preview window displays dropped frames message.	• Close the preview window and reopen it.

Contacting MathWorks and Using the imaqsupport Function

If you need support from MathWorks, see Contact Support.

Before contacting MathWorks, you should run the `imaqsupport` function. This function returns diagnostic information such as:

- The versions of MathWorks products you are using
- Your MATLAB path
- The characteristics of your hardware
- Information about your adaptors

The output from `imaqsupport` is automatically saved to a text file, `imaqsupport.txt`, which you can use to help troubleshoot your problem.

To have MATLAB generate this file for you, type

```
imaqsupport
```

Image Acquisition Toolbox Examples

Identifying Available Devices

This example shows how to identify the available devices on your system and obtain device information.

Identifying Installed Adaptors

The `imaqhwinfo` function provides a structure with an `InstalledAdaptors` field that lists all adaptors on the current system that the toolbox can access.

```
imaqInfo = imaqhwinfo
```

```
imaqInfo =

    InstalledAdaptors: {'dcam'  'winvideo'}
        MATLABVersion: '7.1 (R14SP3)'
          ToolboxName: 'Image Acquisition Toolbox'
       ToolboxVersion: '1.9 (R14SP3)'
```

```
imaqInfo.InstalledAdaptors
```

```
ans =

    'dcam'    'winvideo'
```

Obtaining Device Information

Calling `imaqhwinfo` with an adaptor name returns a structure that provides information on all accessible image acquisition devices.

```
hwInfo = imaqhwinfo('winvideo')
```

```
hwInfo =

       AdaptorDllName: [1x68 char]
    AdaptorDllVersion: '1.9 (R14SP3)'
          AdaptorName: 'winvideo'
            DeviceIDs: {[1]  [3]}
           DeviceInfo: [1x2 struct]
```

```
hwInfo.DeviceInfo
```

```
ans =

1x2 struct array with fields:
    DefaultFormat
    DeviceFileSupported
    DeviceName
    DeviceID
    VideoInputConstructor
    VideoDeviceConstructor
    SupportedFormats
```

Information on a specific device can be obtained by simply indexing into the device information structure array.

```
device1 = hwInfo.DeviceInfo(1)
```

```
device1 = 

             DefaultFormat: 'RGB555_320x240'
       DeviceFileSupported: 0
                DeviceName: 'Veo PC Camera'
                  DeviceID: 1
     VideoInputConstructor: 'videoinput('winvideo', 1)'
    VideoDeviceConstructor: 'imaq.VideoDevice('winvideo', 1)'
          SupportedFormats: {1x30 cell}
```

The `DeviceName` field contains the image acquisition device name.

```
device1.DeviceName
```

```
ans =

Veo PC Camera
```

The `DeviceID` field contains the image acquisition device identifier.

```
device1.DeviceID
```

```
ans =

     1
```

The `DefaultFormat` field contains the image acquisition device's default video format.

```
device1.DefaultFormat
```

```
ans =

RGB555_320x240
```

The `SupportedFormats` field contains a cell array of all valid video formats supported by the image acquisition device.

```
device1.SupportedFormats
```

```
ans =
Columns 1 through 4

  'I420_128x96'    'I420_160x120'    'I420_176x144'    'I420_320x240'

Columns 5 through 8

  'I420_352x240'    'I420_352x288'    'RGB24_128x96'    'RGB24_160x120'

Columns 9 through 11

  'RGB24_176x144'    'RGB24_320x240'    'RGB24_352x240'

Columns 12 through 14

  'RGB24_352x288'    'RGB555_128x96'    'RGB555_160x120'

Columns 15 through 17

  'RGB555_176x144'    'RGB555_320x240'    'RGB555_352x240'

Columns 18 through 21

  'RGB555_352x288'    'UYVY_128x96'    'UYVY_160x120'    'UYVY_176x144'

Columns 22 through 25

  'YUY2_128x96'    'YUY2_160x120'    'YUY2_176x144'    'YV12_128x96'

Columns 26 through 29

  'YV12_160x120'    'YV12_176x144'    'YV12_320x240'    'YV12_352x240'

Column 30

  'YV12_352x288'
```

Accessing Devices and Video Sources

This example shows how to access and connect to a video device.

Accessing an Image Acquisition Device

A video input object represents the connection between MATLAB® and an image acquisition device. To create a video input object, use the VIDEOINPUT function and indicate what device the object is to be associated with.

```
% Access an image acquisition device.
vidobj = videoinput('dt', 1, 'RS170')
```

```
Summary of Video Input Object Using 'Dt313xK'.

   Acquisition Source(s):  VID0, VID1, and VID2 are available.

  Acquisition Parameters:  'VID0' is the current selected source.
                           10 frames per trigger using the selected source.
                           'RS170' video data to be logged upon START.
                           Grabbing first of every 1 frame(s).
                           Log data to 'memory' on trigger.

      Trigger Parameters:  1 'immediate' trigger(s) on START.

                  Status:  Waiting for START.
                           0 frames acquired since starting.
                           0 frames available for GETDATA.
```

Identifying a Device's Video Source Object

A video source object represents a collection of one or more physical data sources that are treated as a single entity. For example, one video source object could represent the three physical connections of an RGB source (red, green, and blue).

The Source property of a video input object provides an array of the device's available video source objects.

```
% Access the device's video sources that can be used for acquisition.
sources = vidobj.Source
```

```
   Display Summary for Video Source Object Array:

      Index:   SourceName:   Selected:
      1        'VID0'        'on'
      2        'VID1'        'off'
      3        'VID2'        'off'
```

```
whos sources
```

```
  Name           Size                      Bytes  Class

  sources        1x3                         872  videosource object
```

```
Grand total is 47 elements using 872 bytes
```

Selecting a Video Source Object for Acquisition

A video source object can be selected for acquisition by specifying its name.

```
vidobj.SelectedSourceName = 'VID2'

% Notice that the corresponding video source has been selected.
sources
```

```
   Display Summary for Video Source Object Array:

      Index:   SourceName:   Selected:
      1        'VID0'        'off'
      2        'VID1'        'off'
      3        'VID2'        'on'
```

To obtain the video source object that is currently selected, use the GETSELECTEDSOURCE function.

```
selectedsrc = getselectedsource(vidobj)
```

```
   Display Summary for Video Source Object:

      Index:   SourceName:   Selected:
      1        'VID2'        'on'
```

Accessing a Video Source Object's Properties

Each video source object provides a list of general and device specific properties.

```
% List the video source object's properties and their current values.
get(selectedsrc)
```

```
  General Settings:
    Parent = [1x1 videoinput]
    Selected = on
    SourceName = VID2
    Tag = 
    Type = videosource
    UserData = []

  Device Specific Properties:
    FirstActiveLine = 21
    FirstActivePixel = 140
    FrameType = interlacedEvenFieldFirst
    StrobeOutput = off
    StrobeOutputDuration = 3.3ms
    StrobeOutputPolarity = activeHigh
    StrobeOutputType = afterFrame
    SyncInput = composite
    TriggerTimeout = 0
```

Note: Each video source object maintains its own property configuration. Modifying the selected video source is equivalent to selecting a new video source configuration.

```
% Once the video input object is no longer needed, delete
% it and clear it from the workspace.
delete(vidobj)
clear vidobj
```

Working with Properties

This example shows how to access and configure video acquisition properties.

Accessing Properties

To access a complete list of an object's properties and their current values, use the `get` function with the object.

```
% Create a video input object.
vidobj = videoinput('dcam', 1);

% List the video input object's properties and their current values.
get(vidobj)
```

```
  General Settings:
    DeviceID = 1
    DiskLogger = []
    DiskLoggerFrameCount = 0
    EventLog = [1x0 struct]
    FrameGrabInterval = 1
    FramesAcquired = 0
    FramesAvailable = 0
    FramesPerTrigger = 10
    Logging = off
    LoggingMode = memory
    Name = RGB24_640x480-dcam-1
    NumberOfBands = 3
    Previewing = off
    ReturnedColorSpace = rgb
    ROIPosition = [0 0 640 480]
    Running = off
    Tag = 
    Timeout = 10
    Type = videoinput
    UserData = []
    VideoFormat = RGB24_640x480
    VideoResolution = [640 480]

  Callback Function Settings:
    ErrorFcn = @imaqcallback
    FramesAcquiredFcn = []
    FramesAcquiredFcnCount = 0
    StartFcn = []
    StopFcn = []
    TimerFcn = []
    TimerPeriod = 1
    TriggerFcn = []

  Trigger Settings:
    InitialTriggerTime = []
    TriggerCondition = none
    TriggerFrameDelay = 0
    TriggerRepeat = 0
    TriggersExecuted = 0
    TriggerSource = none
    TriggerType = immediate

  Acquisition Sources:
    SelectedSourceName = input1
    Source = [1x1 videosource]


% Access the currently selected video source object
src = getselectedsource(vidobj);

% List the video source object's properties and their current values.
get(src)

  General Settings:
    Parent = [1x1 videoinput]
    Selected = on
    SourceName = input1
    Tag =
    Type = videosource

  Device Specific Properties:
    AutoExposure = 511
    AutoExposureMode = auto
    Brightness = 304
    BrightnessMode = auto
    FrameRate = 15
    Gain = 87
    Gamma = 1
    Saturation = 90
    Sharpness = 80
    Shutter = 6
    WhiteBalance = [95 87]
    WhiteBalanceMode = auto
```

To access a specific property value, use dot notation with the object and property name.

```
framesPerTriggerValue = vidobj.FramesPerTrigger;


framesPerTriggerValue =

    10


brightnessValue = src.Brightness;


brightnessValue =

   304
```

Configuring Properties

Enumerated properties have a defined set of possible values. To list the enumerated values of a property, use the `set` function with the object and property name. The property's default value is listed in braces.

```
set(vidobj, 'LoggingMode')
```

```
[ {memory} | disk | disk&memory ]
```

To access a complete list of an object's configurable properties, use the `set` function with the object.

```
% List the video input object's configurable properties.
set(vidobj)
```

```
  General Settings:
    DiskLogger
    FrameGrabInterval
    FramesPerTrigger
    LoggingMode: [ {memory} | disk | disk&memory ]
    Name
    ReturnedColorSpace: [ {rgb} | grayscale | YCbCr ]
    ROIPosition
    Tag
    Timeout
    UserData

  Callback Function Settings:
    ErrorFcn: string -or- function handle -or- cell array
    FramesAcquiredFcn: string -or- function handle -or- cell array
    FramesAcquiredFcnCount
    StartFcn: string -or- function handle -or- cell array
    StopFcn: string -or- function handle -or- cell array
    TimerFcn: string -or- function handle -or- cell array
    TimerPeriod
    TriggerFcn: string -or- function handle -or- cell array

  Trigger Settings:
    TriggerFrameDelay
    TriggerRepeat

  Acquisition Sources:
    SelectedSourceName: [ {input1} ]
```

```
% List the video source object's configurable properties.
set(src)
```

```
  General Settings:
    Tag

  Device Specific Properties:
    AutoExposure
    AutoExposureMode: [ {auto} | manual ]
    Brightness
    BrightnessMode: [ {auto} | manual ]
    FrameRate: [ {15} | 7.5 | 3.75 ]
    Gain
    Gamma
    Saturation
    Sharpness
    Shutter
    WhiteBalance
    WhiteBalanceMode: [ {auto} | manual ]
```

To configure an object's property value, use dot notation with the object, property name, and property value.

```
vidobj.TriggerRepeat = 2;
src.Saturation = 100;
```

Getting Property Help and Information

To obtain a property's description, use the `imaqhelp` function with the object and property name. `imaqhelp` can also be used for function help.

```
imaqhelp(vidobj, 'LoggingMode')
```

```
    LOGGINGMODE  [ {memory} | disk | disk&memory ]  (Read-only: whileRunning)

    LoggingMode specifies the destination for acquired data.

    LoggingMode can be set to disk, memory,or disk&Memory.

    If LoggingMode is set to disk, then acquired data is streamed to a disk file
    as specified by the DiskLogger property.

    If LoggingMode is set to memory, acquired data is stored in a memory buffer.

    If LoggingMode is set to disk&Memory, then acquired data is stored in memory
    and is streamed to a disk file as specified by the DiskLogger property.

    When logging to memory, you must extract the data in a timely manner with the
    GETDATA function. If the data is not extracted in a timely manner, memory
    resources may be used up.

    The value of LoggingMode cannot be modified while the object is running.

    See also DiskLogger, IMAQDEVICE/GETDATA.
```

To obtain information on a property's attributes, use the `propinfo` function with the object and property name.

```
propinfo(vidobj, 'LoggingMode')
```

```
ans = 

               Type: 'string'
         Constraint: 'enum'
    ConstraintValue: {'memory'  'disk'  'disk&memory'}
       DefaultValue: 'memory'
           ReadOnly: 'whileRunning'
     DeviceSpecific: 0
```

When an image acquisition object is no longer needed, remove it from memory and clear the MATLAB® workspace of the associated variable.

```
delete(vidobj);
clear vidobj
```

Managing Video Input Objects

This example shows how to find video input objects and remove video input objects from memory.

Finding Video Input Objects in Memory

To find video input objects in memory, use the IMAQFIND function. IMAQFIND returns an array of video input objects.

```
objects = imaqfind
```

```

objects =

     []

```

```
% Create video input objects.
vidobj1 = videoinput('matrox', 1, 'M_CCIR');
vidobj2 = videoinput('matrox', 1, 'M_PAL_RGB');
vidobj3 = videoinput('matrox', 1, 'M_NTSC_RGB');

% Find all valid objects.
objects = imaqfind
```

```

   Video Input Object Array:

   Index:   Type:          Name:
   1        videoinput     M_CCIR-matrox-1
   2        videoinput     M_PAL_RGB-matrox-1
   3        videoinput     M_NTSC_RGB-matrox-1
```

Removing Objects from Memory

To delete a video input object from memory, use the DELETE function with the object.

```
% Delete the first object in the array.
delete(objects(1))
```

Find all remaining valid objects.

```
objects = imaqfind
```

```

   Video Input Object Array:

   Index:   Type:          Name:
   1        videoinput     M_PAL_RGB-matrox-1
   2        videoinput     M_NTSC_RGB-matrox-1
```

Using the DELETE function with the object will remove the object from memory but not from the MATLAB® workspace. To remove an object from the MATLAB workspace use the CLEAR function. To see what objects are in the MATLAB workspace, use the WHOS function.

```
% Display the current workspace.
whos
```

```
   Name            Size                       Bytes  Class

   objects         1x2                         1200  videoinput object
   vidobj1         1x1                         1060  videoinput object
   vidobj2         1x1                         1060  videoinput object
   vidobj3         1x1                         1060  videoinput object

Grand total is 185 elements using 4380 bytes
```

Since an object was deleted, it is no longer valid.

```
vidobj1
```

```
Invalid Image Acquisition object.
This object is not associated with any hardware and
should be removed from your workspace using CLEAR.
```

Clear the associated variable.

```
clear vidobj1
```

Display the current workspace.

```
whos
```

```
  Name            Size                       Bytes  Class

  objects         1x2                         1200  videoinput object
  vidobj2         1x1                         1060  videoinput object
  vidobj3         1x1                         1060  videoinput object

Grand total is 142 elements using 3320 bytes
```

To remove all video input objects from memory and to reset the toolbox to its initial state, use the IMAQRESET function.

Note: Using the IMAQRESET function will only delete objects from memory, not clear them from the MATLAB workspace.

```
imaqreset
```

Verify no objects remain.

```
objects = imaqfind
```

```
objects =

     []
```

Variables associated with deleted objects still remain.

```
whos
```

```
   Name            Size                       Bytes  Class
```

```
  objects       0x0                          0  double array
  vidobj2       1x1                       1060  videoinput object
  vidobj3       1x1                       1060  videoinput object

Grand total is 86 elements using 2120 bytes
```

Clear any remaining variables associated with deleted objects.

```
clear vidobj2 vidobj3
```

Logging Data to Memory

This example shows how to log image data and view logged data.

Previewing Data

Before logging data, images from an image acquisition device can be previewed live using the PREVIEW function. Calling the PREVIEW function, will open a preview window. To close the preview window, use the CLOSEPREVIEW function.

```
% Access an image acquisition device.
vidobj = videoinput('winvideo', 1);

% Open the preview window.
preview(vidobj)
```

Single Frame Acquisition

To acquire a single frame, use the GETSNAPSHOT function.

```
snapshot = getsnapshot(vidobj);

% Display the frame in a figure window.
imagesc(snapshot)
```

Multi-Frame Acquisition

To specify the number of frames to log upon triggering, configure the video input object's FramesPerTrigger property.

```
% Configure the number of frames to log upon triggering.
vidobj.FramesPerTrigger = 50;
```

An image acquisition object must be running before data can be logged. To initiate an acquisition, use the START function.

```
start(vidobj)

% Notice that the number of frames being logged to memory ...
numAvail = vidobj.FramesAvailable


numAvail =

     7


% ... is increasing ...
numAvail = vidobj.FramesAvailable


numAvail =

    14


% ... over time.
numAvail = vidobj.FramesAvailable


numAvail =

    21
```

To retrieve logged data from memory, use the GETDATA function with the video input object and the number of frames to retrieve.

```
% Retrieve some of the logged frames.
imageData = getdata(vidobj, 30);

% Notice the number of frames remaining in memory.
numAvail = vidobj.FramesAvailable


numAvail =

    20


% Display the last frame extracted from memory.
imagesc(imageData(:,:,:,30))
```

```
% Wait for the acquisition to finish.
wait(vidobj);
```

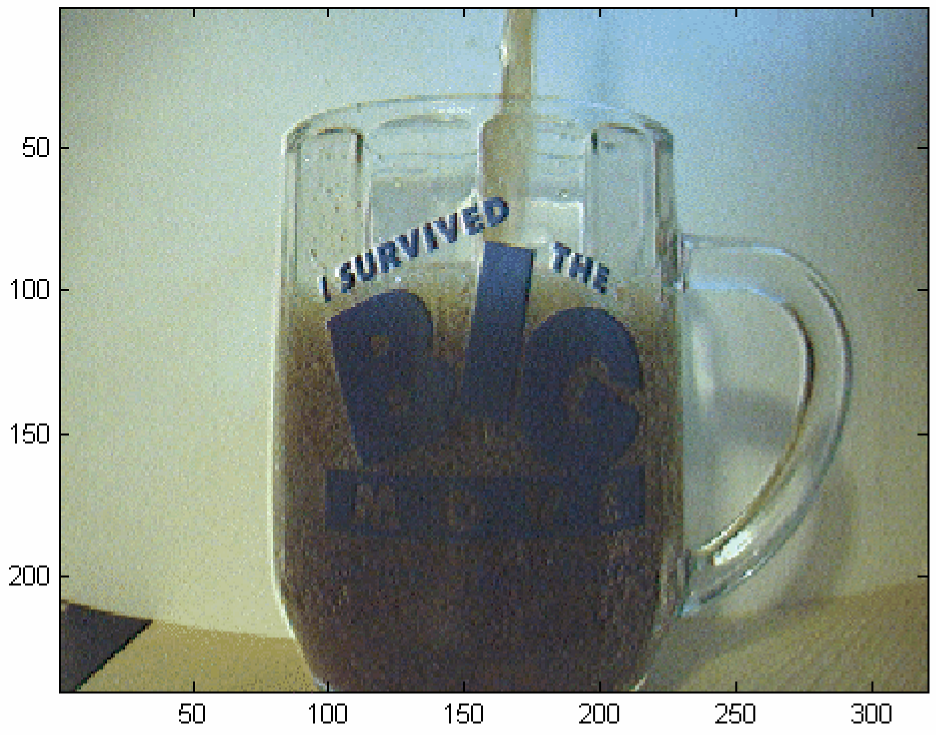
To acquire data continuously, configure the FramesPerTrigger property to infinity. Upon triggering, data will be logged until the video input object stops running. To stop an object from running, use the STOP function.

```
vidobj.FramesPerTrigger = inf;

% Initiate the acquisition.
start(vidobj)

% Notice the number of frames in memory.
numAvail = vidobj.FramesAvailable
```

```
numAvail =

     6
```

```
% Loop through till 10 frames are acquired
while(numAvail<=10)
    numAvail = vidobj.FramesAvailable;
end

% Stop the acquisition.
stop(vidobj)
```

```
% View the total number of frames that were logged before stopping.
numAcquired = vidobj.FramesAcquired;


numAcquired =

    10


% Retrieve all logged data.
imageData = getdata(vidobj, numAcquired);

% Display one of the logged frames.
imagesc(imageData(:,:,:,numAcquired))
```



Viewing Logged Data.

To view the most recently logged image data without extracting it from memory, use the PEEKDATA function with the video input object and the number of frames to view. Viewing logged data using PEEKDATA will not remove any logged data from memory.

```
% Configure the number of frames to log upon triggering.
vidobj.FramesPerTrigger = 35;

% Initiate the acquisition.
start(vidobj)

% Wait for the acquisition to finish.
wait(vidobj, 3);
```

```
% Verify the number of frames logged to memory.
numAvail = vidobj.FramesAvailable


numAvail =

    35


% Access the logged data without extracting them from memory.
imageData = peekdata(vidobj, numAvail);

% Verify that all logged frames are still available in memory.
numFramesAvailable = vidobj.FramesAvailable


numFramesAvailable =

    35
```

Once the video input object is no longer needed, delete and clear the associated variable.

```
delete(vidobj)
clear vidobj
```

Logging Data to Disk

This example shows how to configure logging properties for disk logging and then initiate an acquisition to log.

Configuring Logging Mode

Data acquired from an image acquisition device may be logged to memory, to disk, or both. By default, data is logged to memory. To change the logging mode, configure the video input object's `LoggingMode` property.

```
% Access an image acquisition device, using a grayscale video format with
% 10 bits per pixel.
vidobj = videoinput('gige', 1, 'Mono10');

% View the default logging mode.
currentLoggingMode = vidobj.LoggingMode;
```

```
currentLoggingMode =

memory
```

```
% List all possible logging modes.
set(vidobj, 'LoggingMode')
```

```
[ {memory} | disk | disk&memory ]
```

```
% Configure the logging mode to disk.
vidobj.LoggingMode = 'disk';

% Verify the configuration.
currentLoggingMode = vidobj.LoggingMode;
```

```
currentLoggingMode =

disk
```

Configuring Disk Logging Properties

Logging to disk requires a MATLAB® `VideoWriter` object. `VideoWriter` is a MATLAB function, not a toolbox function. After you create and configure a `VideoWriter` object, provide it to the video input object's `DiskLogger` property.

`VideoWriter` provides a number of different profiles that log the data in different formats. This example uses the Motion JPEG 2000 profile which can log single-banded (grayscale) data as well as multi-byte data. The complete list of profiles provided by `VideoWriter` can be found in the documentation.

```
% Create a VideoWriter object.
logfile = VideoWriter('logfile.mj2', 'Motion JPEG 2000')
```

```
  VideoWriter
  General Properties:
      Filename: 'logfile.mj2'
          Path: 'C:\Temp'
    FileFormat: 'mj2'
      Duration: 0

  Video Properties:
             ColorChannels:
                    Height:
                     Width:
                FrameCount: 0
                 FrameRate: 30
         VideoBitsPerPixel:
               VideoFormat:
    VideoCompressionMethod: 'Motion JPEG 2000'
          CompressionRatio: 10
       LosslessCompression: 0
               MJ2BitDepth:


% Configure the video input object to use the VideoWriter object.
vidobj.DiskLogger = logfile;
```

Initiating the Acquisition

Now that the video input object is configured for logging data to a Motion JPEG 2000 file, initiate the acquisition.

```
% Start the acquisition.
start(vidobj)

% Wait for the acquisition to finish.
wait(vidobj, 5)
```

When logging large amounts of data to disk, disk writing sometimes lags behind the acquisition. To determine whether all frames have been written to disk, use the `DiskLoggerFrameCount` property.

```
while (vidobj.FramesAcquired ~= vidobj.DiskLoggerFrameCount)
    pause(.1)
end
```

Verify that the `FramesAcquired` property and the `DiskLoggerFrameCount` property have the same value.

```
vidobj.FramesAcquired


ans =

    10


vidobj.DiskLoggerFrameCount


ans =
    10
```

```
% When the video input object is no longer needed, delete
% it and clear it from the workspace.
delete(vidobj)
clear vidobj
```

Working with Triggers

This example shows how to use the different types of triggering and how to configure other trigger properties.

Configuring Trigger Properties

To obtain a list of configurable trigger settings, use the TRIGGERINFO function with the video input object. TRIGGERINFO will return all possible trigger configurations supported by the image acquisition device associated with the video input object. Possible configurations consists of a valid trigger type, trigger condition, and trigger source combination.

Note: All image acquisition devices support immediate and manual trigger types. A hardware trigger type is available only if it is supported by the image acquisition device.

```
% Access an image acquisition device.
vidobj = videoinput('matrox', 1);

% Display all valid trigger configurations.
triggerinfo(vidobj)
```

```
   Valid Trigger Configurations:

      TriggerType:   TriggerCondition:   TriggerSource:
      'immediate'    'none'              'none'
      'manual'       'none'              'none'
      'hardware'     'fallingEdge'       'digitalTrigger'
      'hardware'     'fallingEdge'       'optoTrigger'
      'hardware'     'fallingEdge'       'timer1'
      'hardware'     'fallingEdge'       'timer2'
      'hardware'     'risingEdge'        'digitalTrigger'
      'hardware'     'risingEdge'        'optoTrigger'
      'hardware'     'risingEdge'        'timer1'
      'hardware'     'risingEdge'        'timer2'
```

To configure the trigger settings for an image acquisition device, use the TRIGGERCONFIG function with the desired trigger type, trigger condition, and trigger source.

```
triggerconfig(vidobj, 'hardware', 'fallingEdge', 'optoTrigger')

% View the current trigger configuration.
currentConfiguration = triggerconfig(vidobj)
```

```
currentConfiguration = 

         TriggerType: 'hardware'
    TriggerCondition: 'fallingEdge'
       TriggerSource: 'optoTrigger'
```

Note: Configuring trigger settings requires a unique configuration to be specified. If specifying the trigger type uniquely identifies a configuration, no further arguments need to be provided to TRIGGERCONFIG.

Hardware triggers are the only trigger type that typically have multiple valid configurations.

Immediate Triggering

By default, a video input object's trigger type is configured for immediate triggering. Immediate triggering indicates that data logging is to begin as soon as the START function is issued.

```
% Configure the trigger type.
triggerconfig(vidobj, 'immediate')

% Initiate the acquisition.
start(vidobj)

% Wait for acquisition to end.
wait(vidobj, 2)

% Determine the number frames acquired.
frameslogged = vidobj.FramesAcquired;
```

```
frameslogged =

    10
```

Manual Triggering

Manual triggering requires that the TRIGGER function be issued before data logging is to begin.

```
% Configure the trigger type.
triggerconfig(vidobj, 'manual')

% Initiate the acquisition.
start(vidobj)

% Verify no frames have been logged.
frameslogged = vidobj.FramesAcquired;
```

```
frameslogged =

     0
```

```
% Trigger the acquisition.
trigger(vidobj)

% Wait for the acquisition to end.
wait(vidobj, 2);

% Determine the number frames acquired.
frameslogged = vidobj.FramesAcquired;
```

```
frameslogged =

     10
```

Hardware Triggering

Hardware triggering begins logging data as soon as a trigger condition has been met via the trigger source.

In this example, we have connected an opto-isolated trigger source from a function generator to our image acquisition device. The image acquisition device will begin logging data upon detecting a falling edge signal from the source.

```
% Configure the trigger settings.
triggerconfig(vidobj, 'hardware', 'fallingEdge', 'optoTrigger')
```

Initially, no signal is sent from the source to the image acquisition device.

```
% Initiate the acquisition.
start(vidobj)

% Verify nothing has been acquired.
frameslogged = vidobj.FramesAcquired;
```

```
frameslogged =

     0
```

A square wave signal will now be sent from the trigger source to the image acquisition device.

```
% Wait for the acquisition to end.
wait(vidobj, 10)

% Verify frames were acquired.
frameslogged = vidobj.FramesAcquired;
```

```
frameslogged =

    10
```

```
% Once the video input object is no longer needed, delete
% it and clear it from the workspace.
delete(vidobj)
clear vidobj
```

Acquiring a Single Image in a Loop

This example shows how to use the GETSNAPSHOT function. The GETSNAPSHOT function allows for quick acquisition of a single video frame.

Set up the Acquisition Object

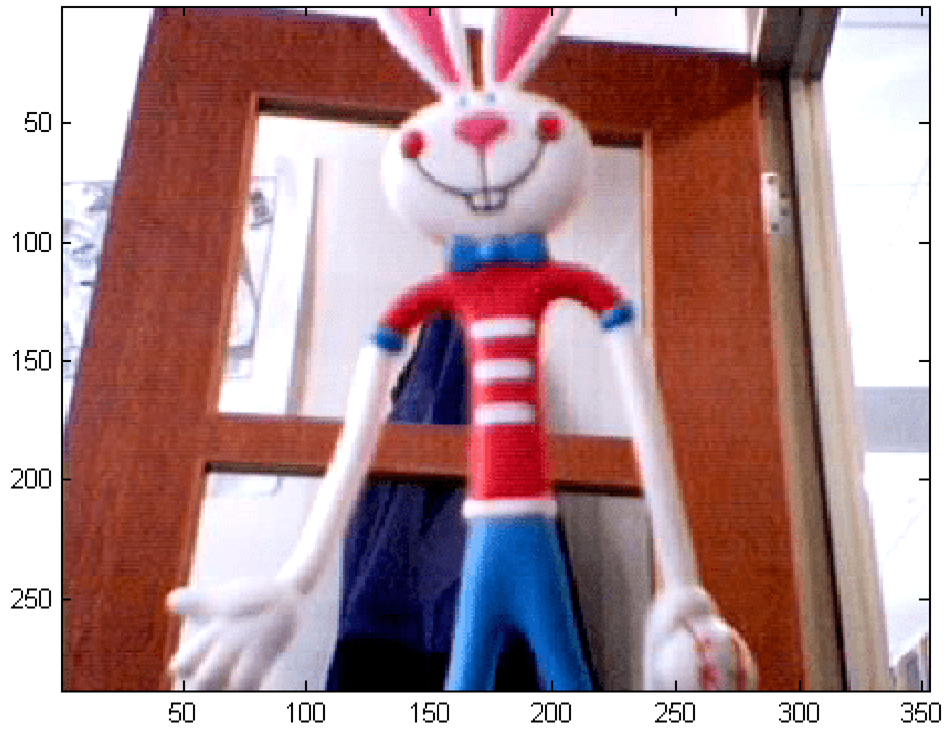
Most interaction with Image Acquisition Toolbox is done through a video input object. These objects are created with the VIDEOINPUT command. This example uses a webcam that is accessed through the toolbox's "winvideo" adaptor.

```
vidobj = videoinput('winvideo');
```

Acquire a Frame

To acquire a single frame, use the GETSNAPSHOT function.

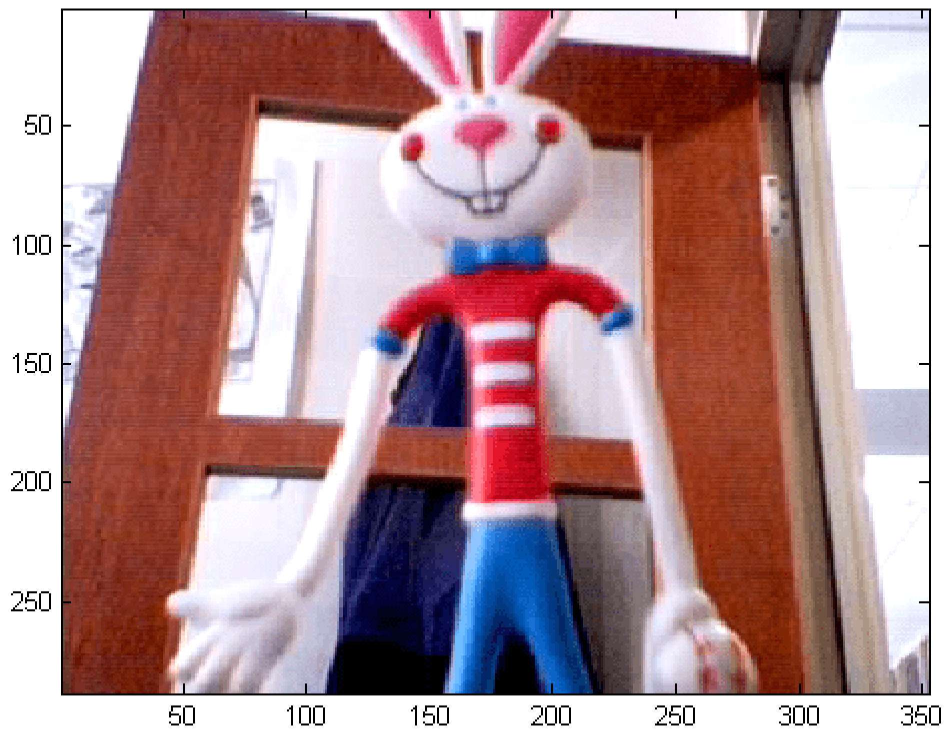
```
snapshot = getsnapshot(vidobj);

% Display the frame in a figure window.
imagesc(snapshot)
```



Acquire Multiple Frames

A common task is to repeatedly acquire a single image, process it, and then store the result. To do this, GETSNAPSHOT can be called in a loop.

```
for i = 1:5
    snapshot = getsnapshot(vidobj);
    imagesc(snapshot);
end
```



Timing Implications

The GETSNAPSHOT function performs a lot of work when it is called. It must connect to the device, configure it, start the acquisition, acquire one frame, stop the acquisition, and then close the device. This means that the acquisition of one frame can take significantly longer than would be expected based on the frame rate of the camera. To illustrate this, call GETSNAPSHOT in a loop.

```
% Measure the time to acquire 20 frames.
tic
for i = 1:20
    snapshot = getsnapshot(vidobj);
end

elapsedTime = toc

% Compute the time per frame and effective frame rate.
timePerFrame = elapsedTime/20
effectiveFrameRate = 1/timePerFrame
```

```
elapsedTime =

   21.2434
```

```
timePerFrame =

    1.0622


effectiveFrameRate =

    0.9415
```

The next example illustrates a more efficient way to perform the loop.

Using Manual Trigger Mode

You can avoid the overhead of GETSNAPSHOT described in the previous setting by using the manual triggering mode of the videoinput object. Manual triggering mode allows the toolbox to connect to and configure the device a single time without logging data to memory. This means that frames can be returned to MATLAB® with less of a delay.

```
% Configure the object for manual trigger mode.
triggerconfig(vidobj, 'manual');

% Now that the device is configured for manual triggering, call START.
% This will cause the device to send data back to MATLAB, but will not log
% frames to memory at this point.
start(vidobj)

% Measure the time to acquire 20 frames.
tic
for i = 1:20
    snapshot = getsnapshot(vidobj);
end

elapsedTime = toc

% Compute the time per frame and effective frame rate.
timePerFrame = elapsedTime/20
effectiveFrameRate = 1/timePerFrame

% Call the STOP function to stop the device.
stop(vidobj)
```

```
elapsedTime =

    1.4811


timePerFrame =

    0.0741


effectiveFrameRate =
   13.5031
```

You can see that the elapsed time using manual triggering is much smaller than the previous example.

Cleanup

Once the video input object is no longer needed, delete the associated variable.

```
delete(vidobj)
```

Configuring Callback Properties

This example explains how callback functions work and shows how to use them.

Callback functions are executed when an associated event occurs. To configure a callback to execute for a particular event, configure one of the video input object's callback properties:

- ErrorFcn
- FramesAcquiredFcn
- StartFcn
- StopFcn
- TimerFcn
- TriggerFcn

This tutorial uses a callback function that displays the N'th frame, where N is provided as an input argument to the callback function.

Select a device to use for acquisition and configure it to acquire data upon executing a manual trigger.

```
% Access an image acquisition device.
vidobj = videoinput('winvideo', 1);

% Acquire an infinite number of frames when manually triggered.
triggerconfig(vidobj, 'manual');
vidobj.FramesPerTrigger = inf;
```

Configure the video input object to execute a callback function when the acquisition is stopped.

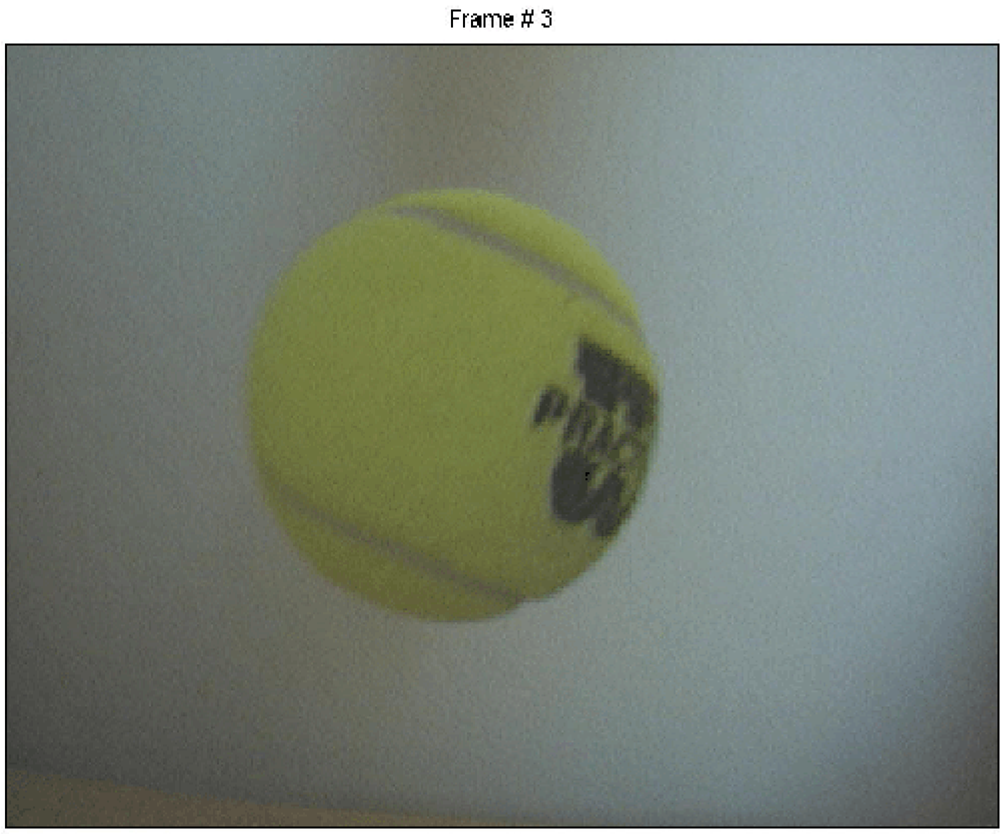
```
% Specify the N'th frame the callback function will display.
frameNumber = 3;

% Have the callback function executed when the acquisition ends.
vidobj.StopFcn = {'util_showframe', frameNumber};

% Initiate the acquisition.
start(vidobj)
```

Upon triggering the image acquisition device, a tennis ball is dropped within the camera's view.

```
% Trigger the object for logging and acquire data for a few seconds.
trigger(vidobj)
pause(5);
```

When the acquisition is stopped, it will cause the callback function to execute and display the N'th frame.

```
% Stop the acquisition.
stop(vidobj)
```



Once the video input object is no longer needed, delete it and clear it from the workspace.

```
delete(vidobj)
clear vidobj
```

Viewing Events

Events occur during an acquisition at a particular time when a condition is met. These events include:

- Error
- FramesAcquired
- Start
- Stop
- Timer
- Trigger

All acquisitions consist of at least 3 events:

- Starting the device
- Triggering the device
- Stopping the device.

Executing an Acquisition

Initiate a basic acquisition using a video input object.

```
% Access an image acquisition device.
vidobj = videoinput('winvideo', 1);

% Use a manual trigger to initiate data logging.
triggerconfig(vidobj, 'manual');

% Start the acquisition.
start(vidobj)

% Trigger the object to start logging and allow the acquisition to run for
% couple of seconds.
trigger(vidobj)
pause(2);

% Stop the acquisition
stop(vidobj)
```

Viewing Event Information

To view event information for the acquisition, access the `EventLog` property of the video input object. Events are recorded in chronological order.

```
% View the event log.
events = vidobj.EventLog
```

```
events = 

1x3 struct array with fields:
    Type
    Data
```

Each event provides information on the state of the object at the time the event occurred.

```
% Display first event.
event1 = events(1)


event1 = 

    Type: 'Start'
    Data: [1x1 struct]


data1 = events(1).Data


data1 = 

             AbsTime: [2005 6 5 23 53 14.1680]
    FrameMemoryLimit: 341692416
     FrameMemoryUsed: 0
         FrameNumber: 0
       RelativeFrame: 0
        TriggerIndex: 0


% Display second event.
event2 = events(2)


event2 = 

    Type: 'Trigger'
    Data: [1x1 struct]


data2 = events(2).Data


data2 = 

             AbsTime: [2005 6 5 23 53 14.7630]
    FrameMemoryLimit: 341692416
     FrameMemoryUsed: 0
         FrameNumber: 0
       RelativeFrame: 0
        TriggerIndex: 1


% Display third event.
event3 = events(3)


event3 = 

    Type: 'Stop'
    Data: [1x1 struct]


data3 = events(3).Data
```

```
data3 = 

             AbsTime: [2005 6 5 23 53 16.9970]
    FrameMemoryLimit: 341692416
     FrameMemoryUsed: 768000
         FrameNumber: 5
       RelativeFrame: 5
        TriggerIndex: 1


% Once the video input object is no longer needed, delete
% it and clear it from the workspace.
delete(vidobj)
clear vidobj
```

Alpha Blending Streamed Image Pairs

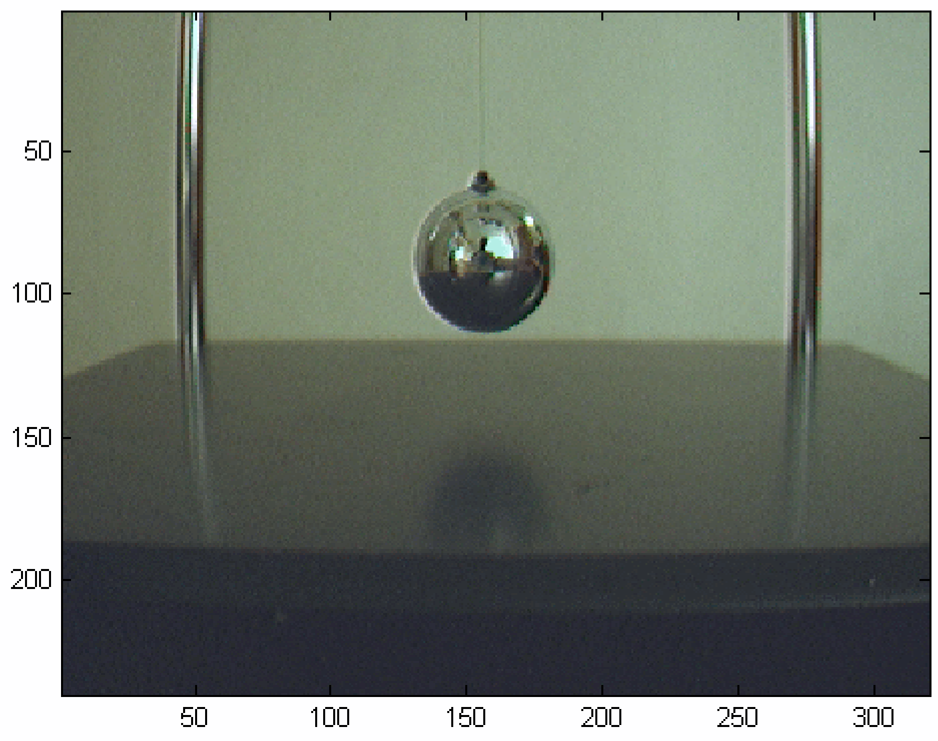
This example shows how to capture streaming images from an image acquisition device, perform on-line image processing on each frame and display the processed frames.

The result is an alpha blend of two images, one a stationary pendulum, the other a pendulum in motion, making moving features appear transparent.

Step 1: Capture A Background Image

Log and display a snapshot of the background with no moving features.

```
% Access an image acquisition device.
vidobj = videoinput('winvideo', 1, 'RGB24_320X240');

% Using the preview window, properly position the camera.
preview(vidobj)
pause(1)

% Capture an image with no moving features.
background = getsnapshot(vidobj);

% Convert the background from uint8 to double.
background = double(background)/255;

% Display the background image in a figure window.
imagesc(background);
```

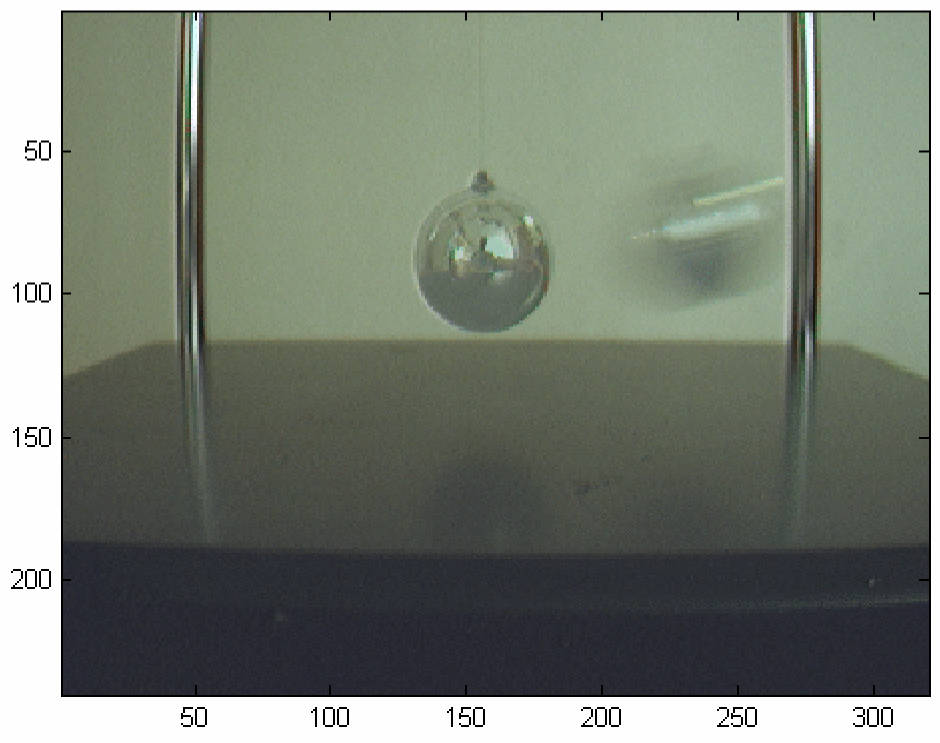


Step 2: Process Logged Data

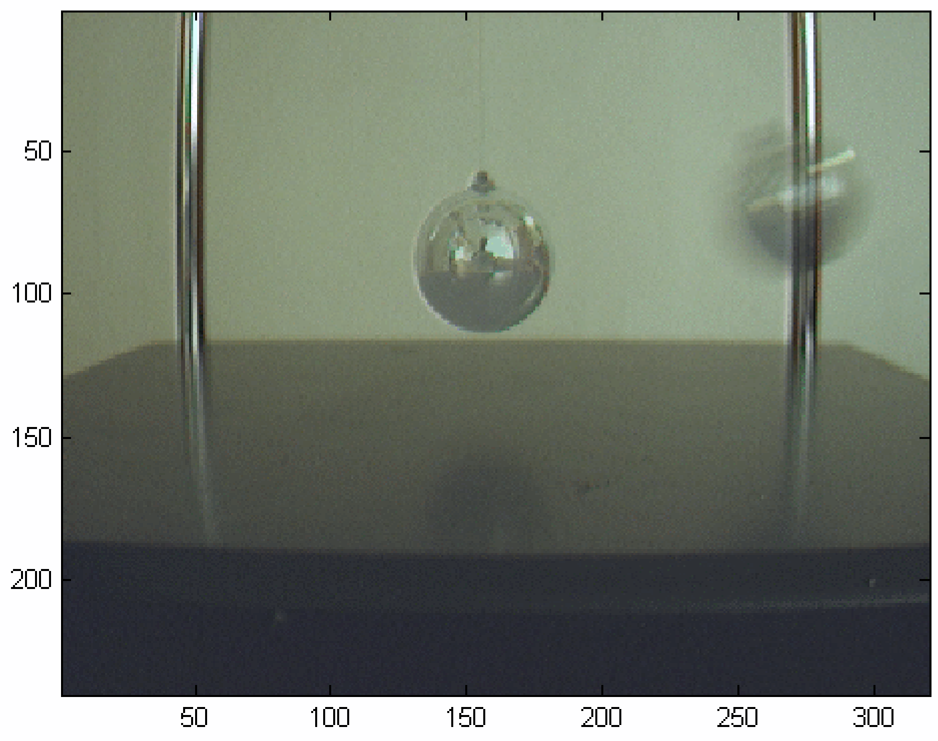
Using the acquired image data, perform on-line image processing, and display the processed images in a figure window.

For each streamed image frame, calculate the linear combination between that frame and the background image. The linear combination effectively alpha blends the two images so any moving features appear transparent.

```
% Set the object into motion.
pause(2);

% Configure the acquisition.
vidobj.FramesPerTrigger = 20;

% Start the acquisition.
start(vidobj)

% While logging data, perform a linear combination between
% the current and background images.
current = getdata(vidobj, 1, 'double');
transparent = (current * 0.5) + (background * 0.5);

% Display the processed image.
imagesc(transparent);
```



```
% Repeat for all remaining images.
while (vidobj.FramesAvailable > 0),
    % Perform a linear combination between the current and background images.
    current = getdata(vidobj, 1, 'double');
    transparent = (current * 0.5) + (background * 0.5);

    % Display the processed image.
    imagesc(transparent);
end
```



```
% Once the video input object is no longer needed, delete
% it and clear it from the workspace.
delete(vidobj)
clear vidobj
```

Alpha Blending Streamed Image Pairs

This example shows how to capture streaming images from an image acquisition device, perform on-line image processing on each frame and display the processed frames.

The result is an alpha blend of two images, one a stationary pendulum, the other a pendulum in motion, making moving features appear transparent.

This example requires Image Processing Toolbox™.

Step 1: Capture A Background Image

Log and display a snapshot of the background with no moving features.

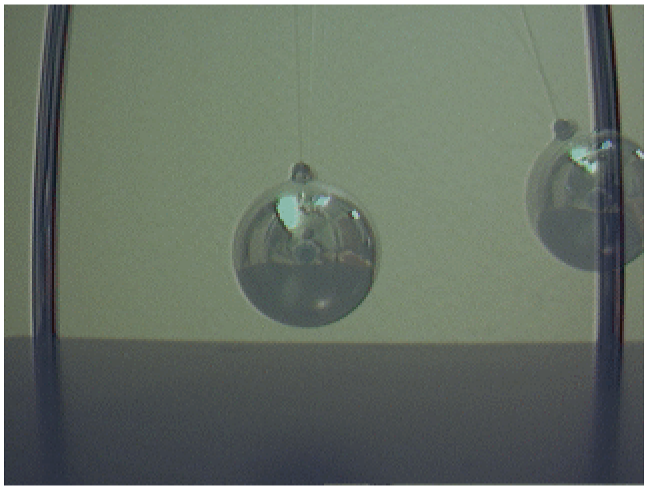
```
% Access an image acquisition device.
vidobj = videoinput('winvideo', 1, 'RGB24_320X240');

% Using the preview window, properly position the camera.
preview(vidobj)
pause(1)

% Capture an image with no moving features.
background = getsnapshot(vidobj);

% Display the background image in a figure window.
imshow(background);
```

Step 2: Process Logged Data

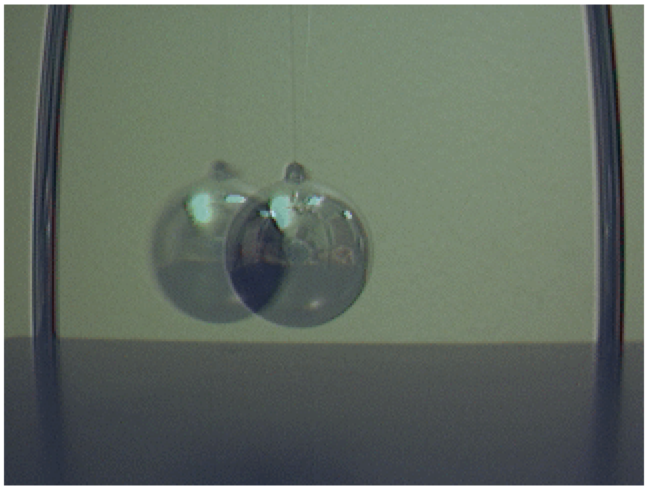
Using the acquired image data, perform on-line image processing, and display the processed images in a figure window.

For each streamed image frame, calculate the linear combination between that frame and the background image. The linear combination effectively alpha blends the two images so any moving features appear transparent.

```
% Set the object into motion.
pause(2);

% Configure the acquisition.
vidobj.FramesPerTrigger = 20;

% Start the acquisition.
start(vidobj)

% While logging data, perform a linear combination between
% the current and background images.
current = getdata(vidobj, 1);
transparent = imlincomb(0.5, current, 0.5, background);

% Display the processed image.
imshow(transparent);
```



```
% Repeat for all remaining images.
while (vidobj.FramesAvailable > 0),
    % Perform a linear combination between the current and background images.
    current = getdata(vidobj, 1);
    transparent = imlincomb(0.5, current, 0.5, background);

    % Display the processed image.
    imshow(transparent);
end
```



```
% Once the video input object is no longer needed, delete
% it and clear it from the workspace.
delete(vidobj)
clear vidobj
```

Averaging Images over Time

This example shows how to average images acquired over time.

For some advanced applications, the acquisition process may require that images be processed as they are acquired, while your processing results are recorded to disk.

Using Image Acquisition Toolbox™ callbacks, triggering, and logging features, this example illustrates how to accomplish the following task:

- acquire 5 frames every 10 seconds
- repeat the acquisition 10 times
- while acquiring images, average the acquired frames and save results to disk.

The experimental setup consists of an hourglass with white sand trickling down over time. The example uses a callback function that averages acquired image frames using Image Processing Toolbox™ functions.

Configuring the Acquisition

Create and configure a video input object for the acquisition.

```
% Access a device using a 24 bit RGB format.
vid = videoinput('winvideo', 1, 'RGB24_320x240');

% Assuming data logging can begin immediately upon START,
% an immediate trigger is used.
triggerconfig(vid, 'immediate');

% Configure the acquisition to collect 5 frames...
framesPerTrigger = 5;
vid.FramesPerTrigger = framesPerTrigger;

% ...and repeat the trigger 9 additional times
% (for a total of 10 trigger executions).
nAdditionalTrigs = 9;
vid.TriggerRepeat = nAdditionalTrigs;
```

To control the rate at which frames will be logged, there are 2 options available:

- configure the device frame rate
- use a TimerFcn to execute a callback

First, a solution using the device's frame rate will be shown, followed by an alternative solution using a timer callback.

Using the frame rate option will provide acquisition results that are most closely aligned with the device's actual video stream rate, whereas using the timer approach provides acquisition results independent of the device's streaming rate.

Frame Rate Based Acquisition (Solution 1)

The device frame rate can only be configured if it is supported by the device. As it is a device specific property, it can be found on the video source object.

```
% Access the video source selected for acquisition.
src = getselectedsource(vid);

% Notice this device provides a FrameRate property.
get(src)


  General Settings:
    Parent = [1x1 videoinput]
    Selected = on
    SourceName = input1
    Tag = 
    Type = videosource
    UserData = []

  Device Specific Properties:
    BacklightCompensation = on
    Brightness = 255
    BrightnessMode = auto
    Contrast = 127
    Exposure = 511
    ExposureMode = auto
    Focus = 58
    FrameRate = 15.1500
    Gamma = 0
    Iris = 4
    Saturation = 108
    Sharpness = 127
    WhiteBalance = 100
    WhiteBalanceMode = auto

% Using the FrameRate property, one can configure the acquisition source
% to provide the toolbox 30 frames per second.
fps = 30;
src.FrameRate = num2str(fps);

% Since the goal is to acquire 5 frames every 10 seconds, the toolbox
% should not acquire any frames until the device provides the 300'th
% frame:
acqPeriod = 10;
frameDelay = fps * acqPeriod


frameDelay =

   300

% If the trigger is delayed by this value, the toolbox will not buffer
% any frames until the 300'th frame is provided by the device.
vid.TriggerFrameDelay = frameDelay;

% To ensure the acquisition does not come close to timing out, configure
% the time out value slightly above the expected acquisition duration.
```

```
totalTrigs = nAdditionalTrigs + 1;
acqDuration = (acqPeriod * totalTrigs) + 3
```

```
acqDuration =

   103
```

```
vid.Timeout = acqDuration;
```

Image Averaging

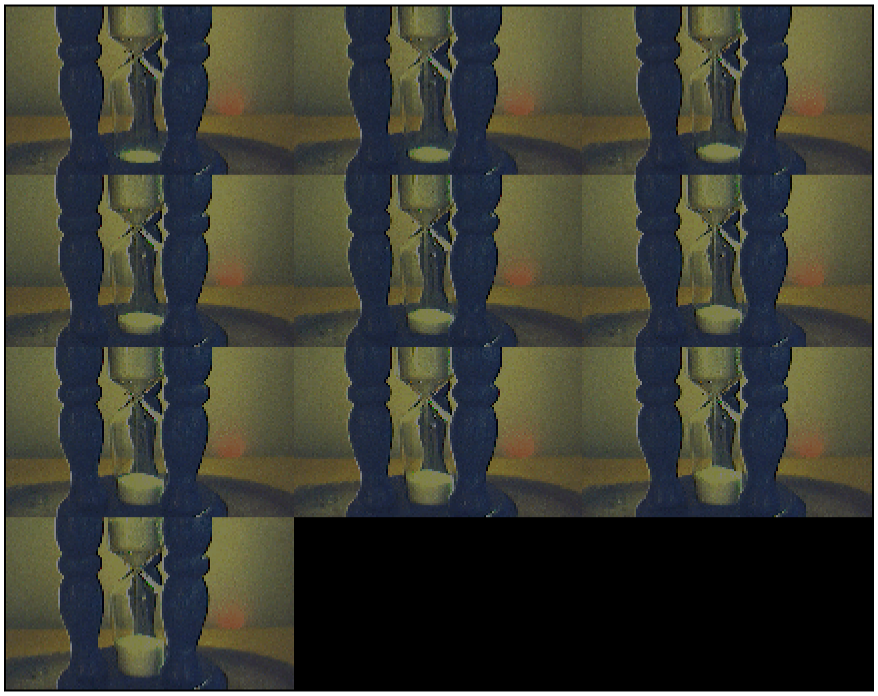
In order to save processed images to disk, a VIDEOWRITER object is used. Each set of acquired frames is averaged using Image Processing Toolbox functions, and then written to disk.

```
% Create an AVI file and configure it.
vwObj = VideoWriter('imaverages.avi', 'Uncompressed AVI');
vwObj.FrameRate = fps;

% Use the video input object's UserData to store processing information.
userdata.average = {};
userdata.avi = vwObj;
vid.UserData = userdata;

% Configure the video input object to process every 5 acquired frames by
% specifying a callback routine that is executed upon every trigger.
vid.TriggerFcn = {'util_imaverage', framesPerTrigger};

% Now that the image acquisition and processing configuration is complete,
% the acquisition is started.
start(vid)

% Wait for the acquisition to complete. This provides the acquisition
% time to complete before the object is deleted.
wait(vid, acqDuration);

% Verify the averaged frames were saved to the AVI file.
userdata = vid.UserData;
vwObj = userdata.avi;
framesWritten1 = vwObj.FrameCount
```

```
framesWritten1 =

    10
```

```
% Display the resulting averages of the acquired frames.
% Notice the change in the lower chamber of the hourglass over time.
imaqmontage(userdata.average);
title('Averaging Results - Frame Rate Based');
```

Averaging Results - Frame Rate Based



```
% Once the video input object is no longer needed, delete
% it and clear it from the workspace. Also delete and clear the VideoWriter object.
delete(vid)
delete(vwObj)
clear vid vwObj
```

Timer Based Acquisition (Solution 2)

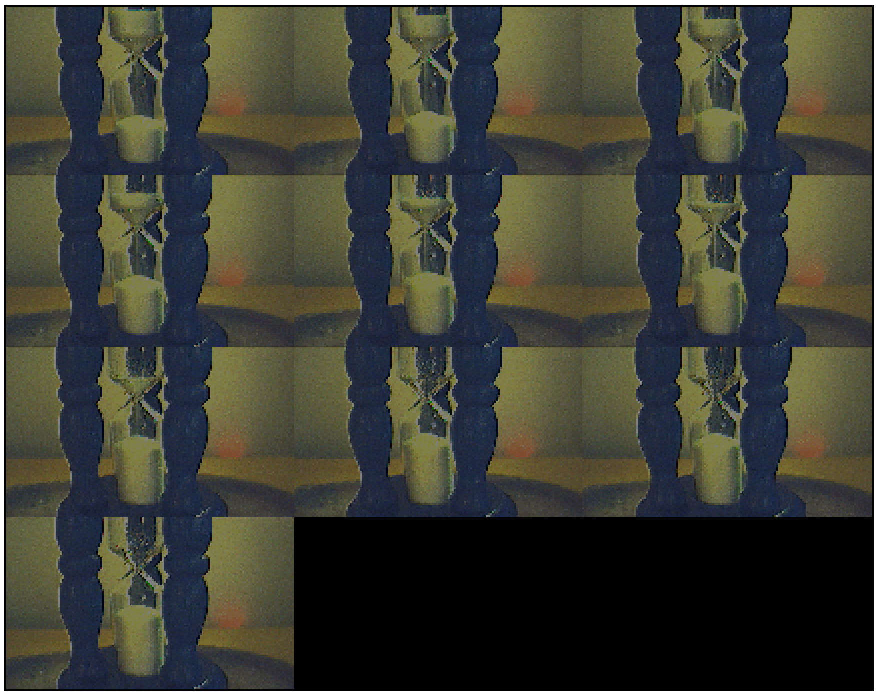
An alternative solution for this task is to use a TimerFcn. The TimerFcn can be executed every 10 seconds, at which point 5 frames are acquired and averaged. In order to initiate the acquisition at the correct moment, manual triggers are used.

Note, this approach is independent of configuring the device's frame rate.

```
% Access a device and configure the acquisition. Have
% the TimerFcn trigger the acquisition every 10 seconds.
vid = videoinput('winvideo', 1, 'RGB24_320x240');
triggerconfig(vid, 'manual');
vid.TimerFcn = @trigger;
vid.TimerPeriod = acqPeriod;

% Configure the acquisition to collect 5 frames each time the
% device is triggered. Repeat the trigger 9 additional times.
vid.FramesPerTrigger = framesPerTrigger;
vid.TriggerRepeat = nAdditionalTrigs;

% Configure the processing routine and AVI file.
vid.TriggerFcn = {'util_imaverage', framesPerTrigger};
```

```
vwObj2 = VideoWriter('imaverages2.avi', 'Uncompressed AVI');
vwObj2.FrameRate = fps;


% Use the video input object's UserData to store processing information.
userdata2.average = {};
userdata2.avi = vwObj2;
vid.UserData = userdata2;

% Start the acquisition.
start(vid);
wait(vid, acqDuration);

% Verify the averaged frames were saved to the AVI file.
userdata2 = vid.UserData;
vwObj2 = userdata2.avi;
framesWritten2 = vwObj2.FrameCount


framesWritten2 =

    10

% Display the resulting averages of the acquired frames.
% Notice the change in the lower chamber of the hourglass over time.
imaqmontage(userdata2.average);
title('Averaging Results - Timer Based');
```

Averaging Results - Timer Based



```
% Once the video input object is no longer needed, delete
% it and clear it from the workspace. Also delete and clear the VideoWriter object.
delete(vid)
delete(vwObj2)
clear vid vwObj2
```

Calculating the Length of a Pendulum in Motion

This example shows how to capture and analyze images of an object in motion.

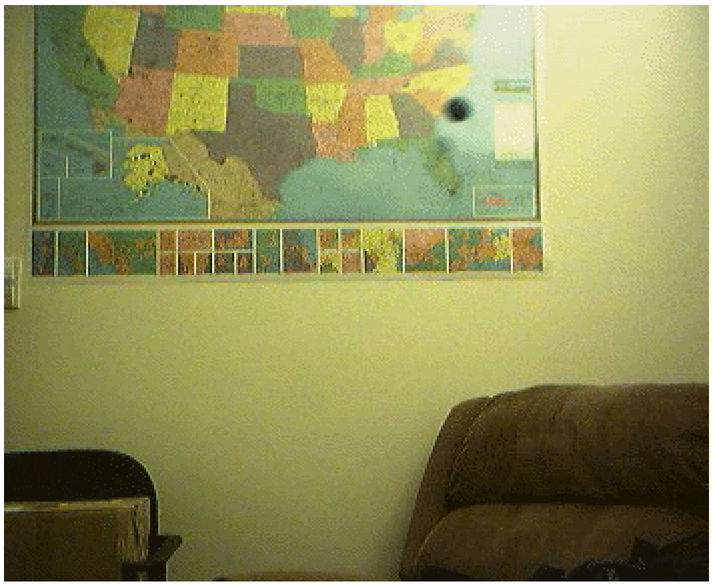
This example captures images of a pendulum in motion. The pendulum consists of a blue ball attached to a nylon string. Image data is captured as the pendulum is swung. Once captured, the images are processed to determine the length of the pendulum.

Images are acquired using the Image Acquisition Toolbox™ and analyzed with the Image Processing Toolbox™.

Acquire Images

Acquire a series of images to analyze.

```
% Access an image acquisition device.
vid = videoinput('winvideo', 1, 'RGB24_352x288');
vid.Timeout = 12;

% Configure object to capture every fifth frame.
vid.FrameGrabInterval = 5;

% Configure the number of frames to be logged.
vid.FramesPerTrigger = 50;

% Access the device's video source object selected for acquisition.
src = getselectedsource(vid);

% Configure the device to provide 30 frames per second.
src.FrameRate = '30';

% Open a live preview window. Focus camera onto a moving pendulum.
preview(vid);

% Initiate the acquisition.
start(vid);

% Extract frames from memory.
frames = getdata(vid);

% Remove video input object from memory.
delete(vid)
clear vid

% Display the first frame in the series.
imshow( frames(:, :, :, 1) );
```



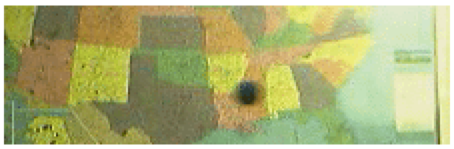
```
% Display all acquired images.
imaqmontage(frames);
```

Select Region Of Interest

Since the pendulum's motion is confined to the upper half of the image series. Create a new series of frames that contain the region of interest.

To crop a series of frames using `imcrop`, first perform `imcrop` on one frame and store its output. Then create a series of frames having the size of the previous output.

```
% Determine the total number of frames acquired.
nFrames = size(frames, 4);

% Crop the first frame.
roi = [50 16 222 68];
firstFrame = frames(:,:,:,1);
frameRegion = imcrop(firstFrame, roi);

% Create a storage for the modified image series.
regions = repmat(uint8(0), [size(frameRegion) nFrames]);
for count = 1:nFrames,
    regions(:,:,:,count) = imcrop(frames(:,:,:,count), roi);
    imshow(regions(:,:,:,count));
end
```



Segment the Pendulum in Each Frame

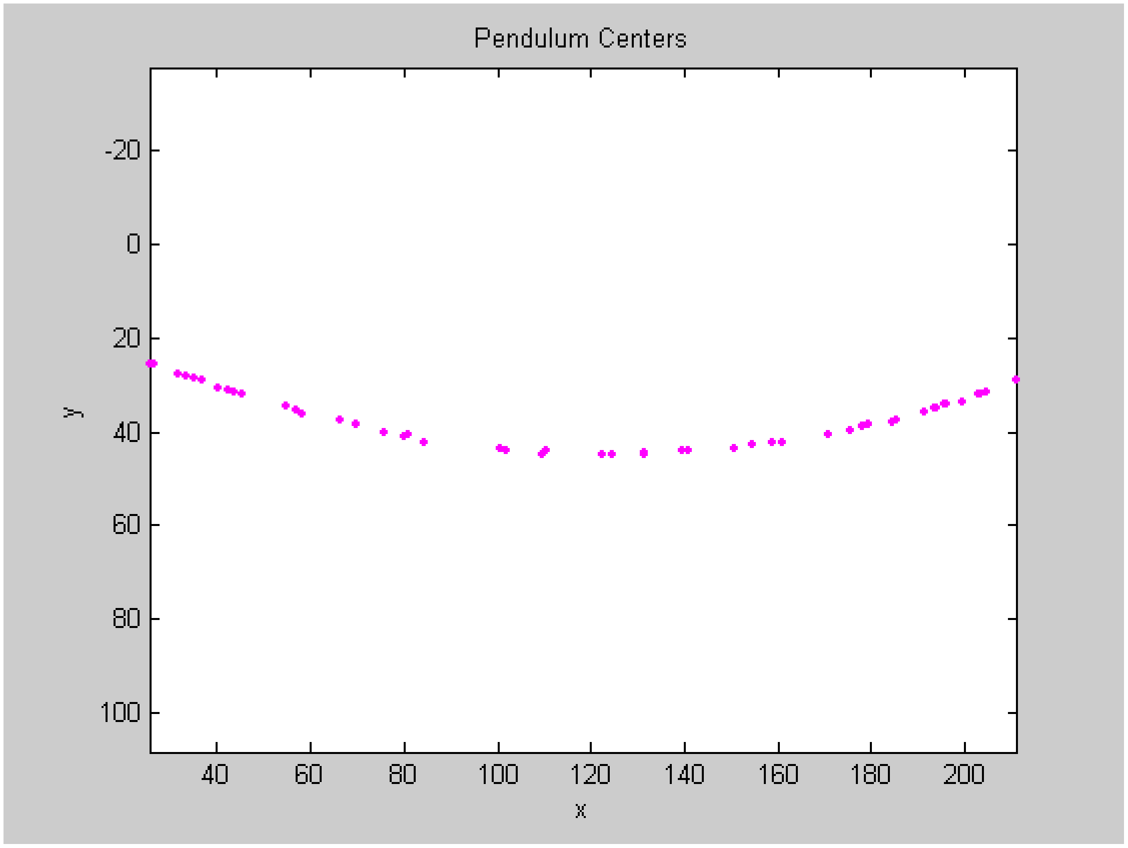
Since the pendulum is much darker than the background, the pendulum can be segmented in each frame by converting the frame to grayscale, thresholding it, and removing background structures.

```
% Initialize array to contain the segmented pendulum frames.
segPend = false([size(frameRegion, 1) size(frameRegion, 2) nFrames]);
centroids = zeros(nFrames, 2);
structDisk = strel('disk', 3);

for count = 1:nFrames,
    % Convert to grayscale.
    fr = regions(:,:,:,count);
    gfr = rgb2gray(fr);
    gfr = imcomplement(gfr);

    % Experimentally determine the threshold.
    bw = im2bw(gfr, .7);
    bw = imopen(bw, structDisk);
    bw = imclearborder(bw);
    segPend(:,:,count) = bw;
    imshow(bw);
end
```

Find the Centers of Each Segmented Pendulum

The shape of the segmented pendulum in each frame is not a serious issue because the pendulum's center is the only characteristic needed to determine the length of the pendulum.

```
% Calculate the pendulum centers.
for count = 1:nFrames,
    property = regionprops(segPend(:, :, count), 'Centroid');
    pendCenters(count,:) = property.Centroid;
end

% Display the pendulum centers and adjust the plot.
figure;
x = pendCenters(:, 1);
y = pendCenters(:, 2);
plot(x, y, 'm.');
axis ij;
axis equal;
hold on;
xlabel('x');
ylabel('y');
title('Pendulum Centers');
```



Calculate Pendulum Length

By fitting a circle through the pendulum centers, the pendulum length can be calculated. Rewrite the basic equation of a circle:

- (x-xc)^2 + (y-yc)^2 = radius^2

where (xc,yc) is the center, in terms of parameters a, b, and c:

- x^2 + y^2 + a*x + b*y + c = 0

where:

- a = -2*xc
- b = -2*yc
- c = xc^2 + yc^2 - radius^2.

Solving for parameters a, b, and c using the least squares method, the above equation can be rewritten as:

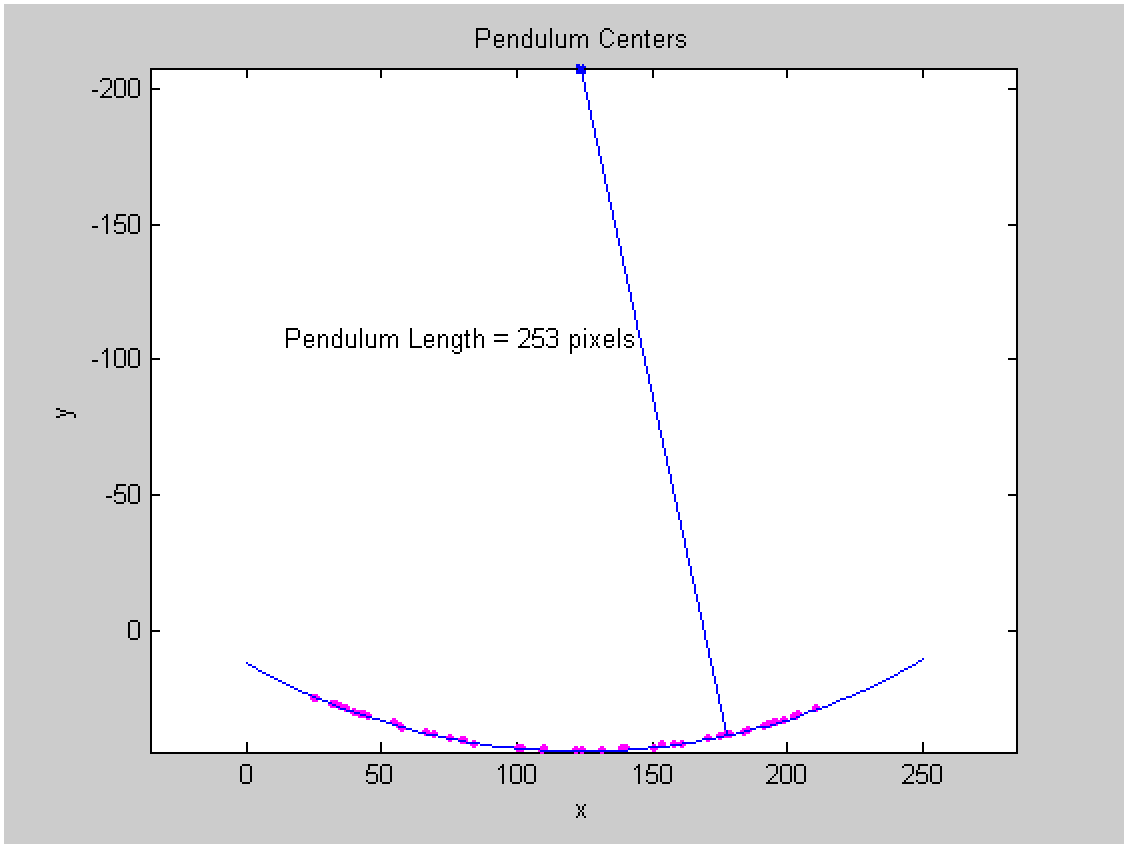
- a*x + b*y + c = -(x^2 + y^2)

which can also be rewritten as:

- [a; b; c] * [x y 1] = -x^2 - y^2

This equation can be solved in MATLAB® using the backslash(\) operator.

```
% Solve the equation.
abc = [x y ones(length(x),1)] \ [-(x.^2 + y.^2)];
a = abc(1);
b = abc(2);
c = abc(3);
xc = -a/2;
yc = -b/2;
circleRadius = sqrt((xc^2 + yc^2) - c);

% Circle radius is the length of the pendulum in pixels.
pendulumLength = round(circleRadius)


pendulumLength =

   253

% Superimpose results onto the pendulum centers
circle_theta = pi/3:0.01:pi*2/3;
x_fit = circleRadius*cos(circle_theta) + xc;
y_fit = circleRadius*sin(circle_theta) + yc;
plot(x_fit, y_fit, 'b-');
plot(xc, yc, 'bx', 'LineWidth', 2);
plot([xc x(1)], [yc y(1)], 'b-');
titleStr = sprintf('Pendulum Length = %d pixels', pendulumLength);
text(xc-110, yc+100, titleStr);
```



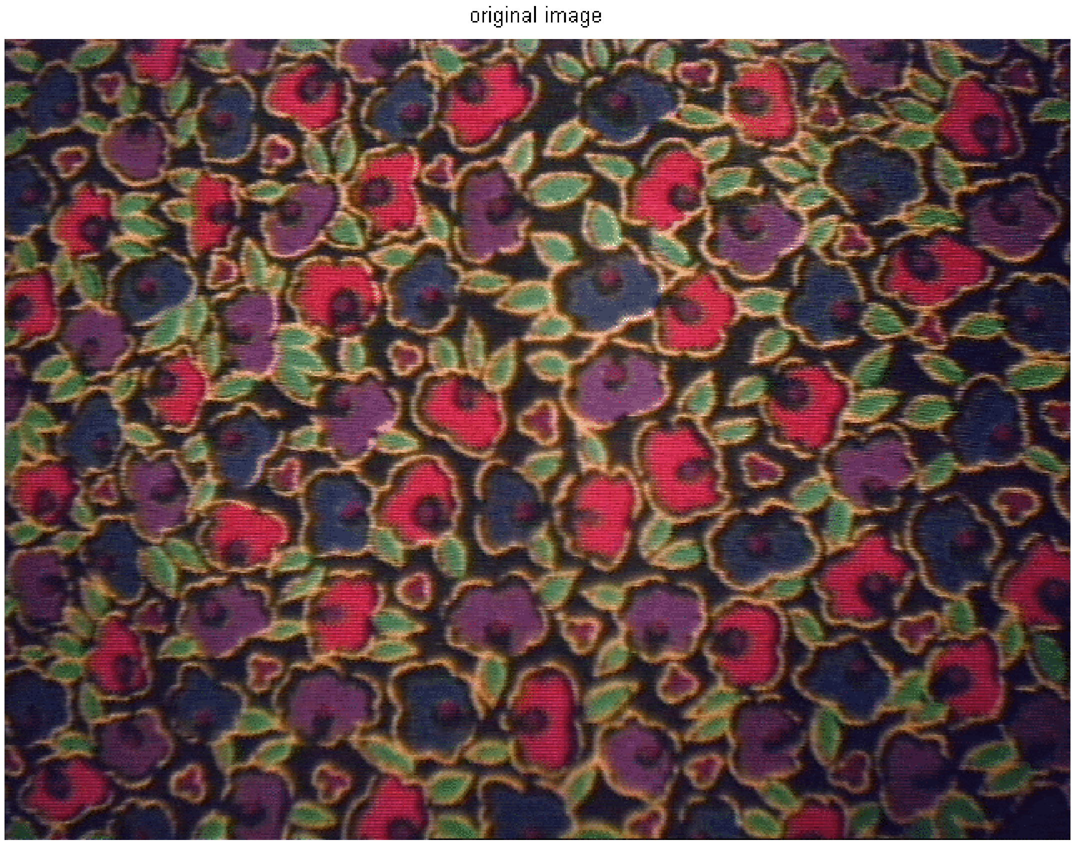
Color-Based Segmentation of Fabric Using the L*a*b Color Space

This example shows how to acquire a single image frame of a piece of colorful fabric. The different colors in the fabric are identified using the L*a*b color space.

This example requires the use of the Image Processing Toolbox™.

Step 1: Acquire Image

```
% Create a video input object to access the image acquisition device.
vid = videoinput('matrox', 1, 'M_NTSC');

% Capture one frame of data.
fabric = getsnapshot(vid);
figure(1)
imshow(fabric)
title('original image');
```



```
% Determine the image resolution.
imageRes = vid.VideoResolution;
imageWidth = imageRes(1);
imageHeight = imageRes(2);

% Once the video input object is no longer needed, delete
% it and clear it from the workspace.
delete(vid)
clear vid
```

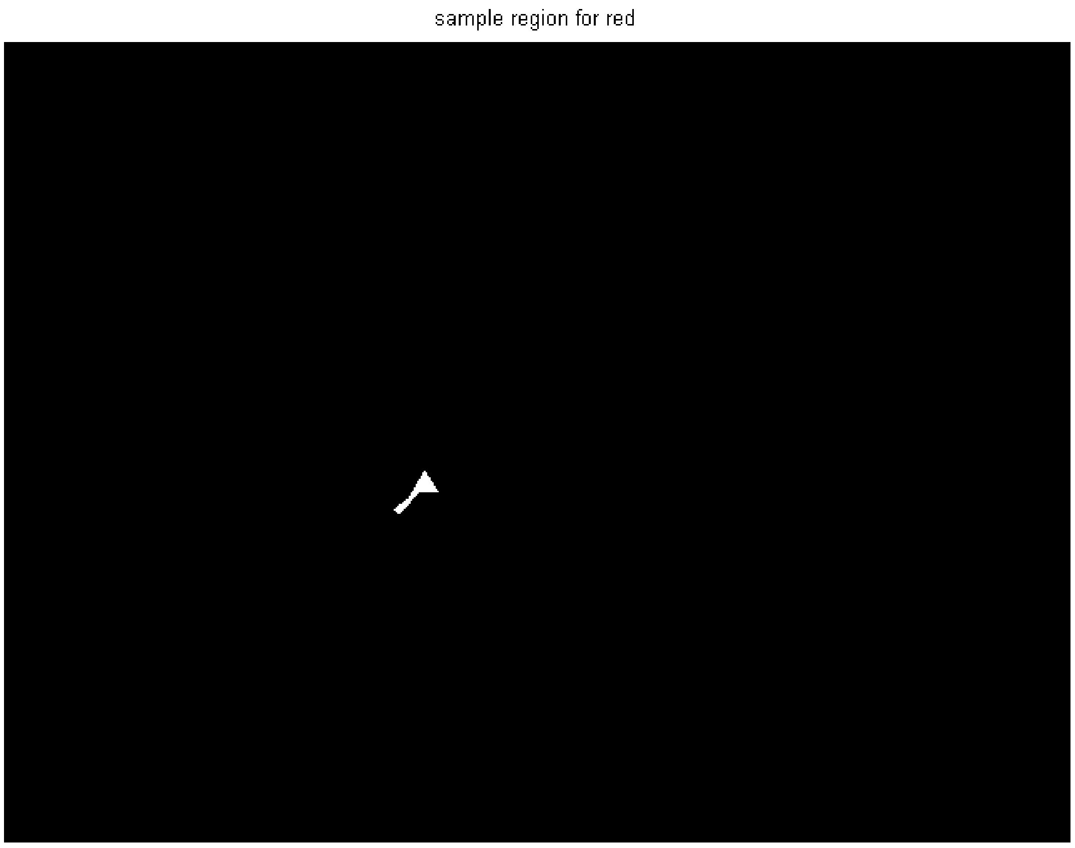
Step 2: Calculate Sample Colors in L*a*b Color Space for Each Region

Count the number of major colors visible in the image. Notice how easily you can visually distinguish these colors from one another. The L*a*b color space (also known as CIELAB or CIE L*a*b) enables you to quantify these visual differences.

The L*a*b color space is derived from the CIE XYZ tristimulus values. The L*a*b space consists of a luminosity ('L') or brightness layer, chromaticity layer 'a' indicating where color falls along the red-green axis, and chromaticity layer 'b' indicating where the color falls along the blue-yellow axis.

Your approach is to choose a small sample region for each color and to calculate each sample region's average color in 'a*b' space. You will use these color markers to classify each pixel.

```
% Initialize storage for each sample region.
colorNames = { 'red','green','purple','blue','yellow' };
nColors = length(colorNames);
sample_regions = false([imageHeight imageWidth nColors]);

% Select each sample region.
f = figure;
for count = 1:nColors
    f.Name = ['Select sample region for ' colorNames{count}];
    sample_regions(:,:,count) = roipoly(fabric);
end
close(f);

% Display a sample region.
imshow(sample_regions(:,:,1))
title(['sample region for ' colorNames{1}]);
```



```
% Convert the fabric RGB image into an L*a*b image.
cform = makecform('srgb2lab');
lab_fabric = applycform(fabric,cform);

% Calculate the mean 'a' and 'b' value for each area extracted.
% These values serve as your color markers in 'a*b' space.
a = lab_fabric(:,:,2);
b = lab_fabric(:,:,3);
color_markers = repmat(0, [nColors, 2]);

for count = 1:nColors
  color_markers(count,1) = mean2(a(sample_regions(:,:,count)));
  color_markers(count,2) = mean2(b(sample_regions(:,:,count)));
end

% For example, the average color of the second sample region in 'a*b' space is:
disp( sprintf('[%0.3f,%0.3f]', color_markers(2,1), color_markers(2,2)) );
```

```
[105.956,147.867]
```

Step 3: Classify Each Pixel Using the Nearest Neighbor Rule

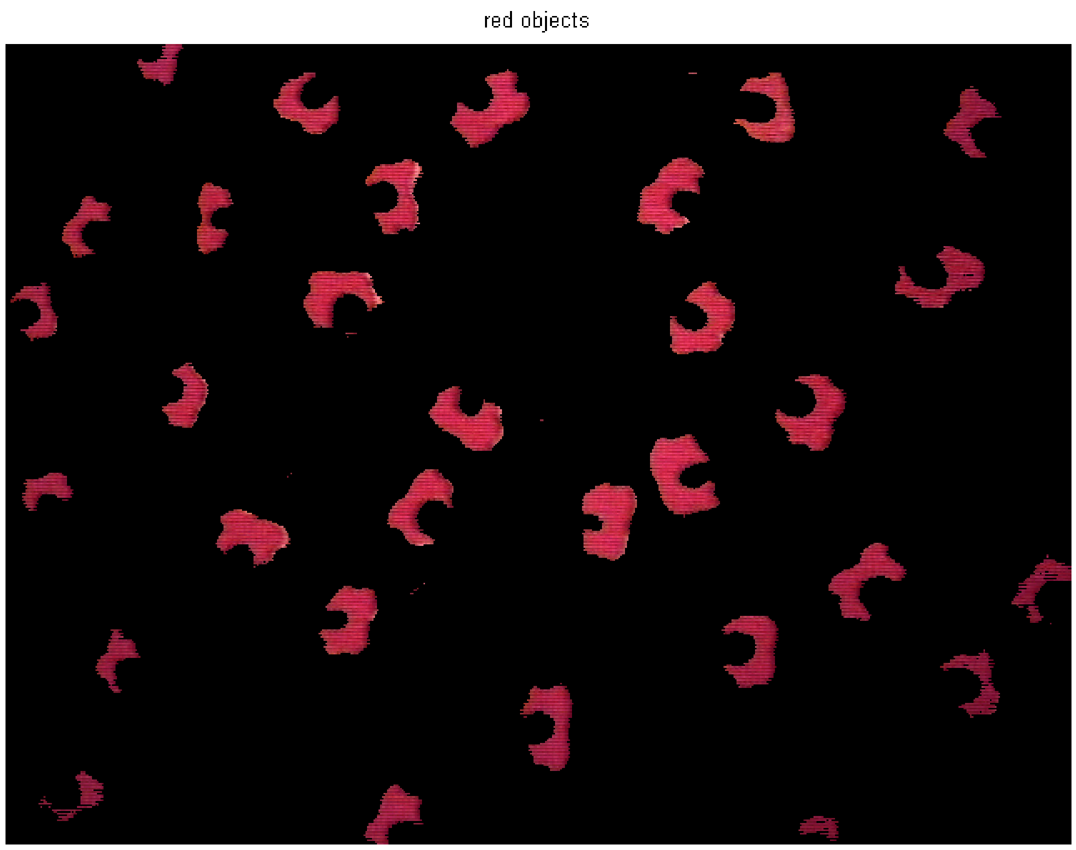
Each color marker now has an 'a' and a 'b' value. You can classify each pixel in the image by calculating the Euclidean distance between that pixel and each color marker. The smallest distance will tell you that the pixel most closely matches that color marker. For example, if the distance between a pixel and the second color marker is the smallest, then the pixel would be labeled as that color.

```
% Create an array that contains your color labels:
%     0 = background
%     1 = red
%     2 = green
%     3 = purple
%     4 = magenta
%     5 = yellow
color_labels = 0:(nColors-1);

% Initialize matrices to be used in the nearest neighbor classification.
a = double(a);
b = double(b);
distance = repmat(0,[size(a), nColors]);

% Perform classification.
for count = 1:nColors
  distance(:,:,count) = ( (a - color_markers(count,1)).^2 + ...
                      (b - color_markers(count,2)).^2 ).^0.5;
end

[value, label] = min(distance, [], 3);
label = color_labels(label);
clear value distance;
```

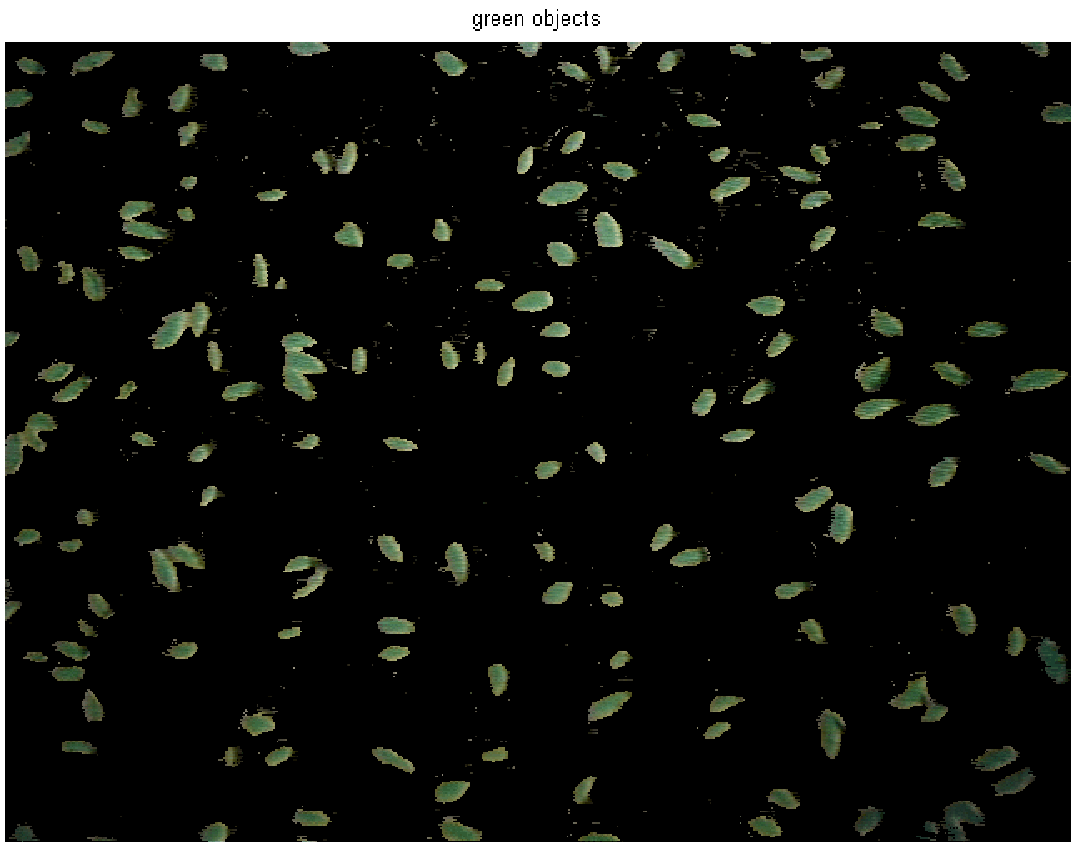
Step 4: Display Results of Nearest Neighbor Classification

The label matrix contains a color label for each pixel in the fabric image. Use the label matrix to separate objects in the original fabric image by color.

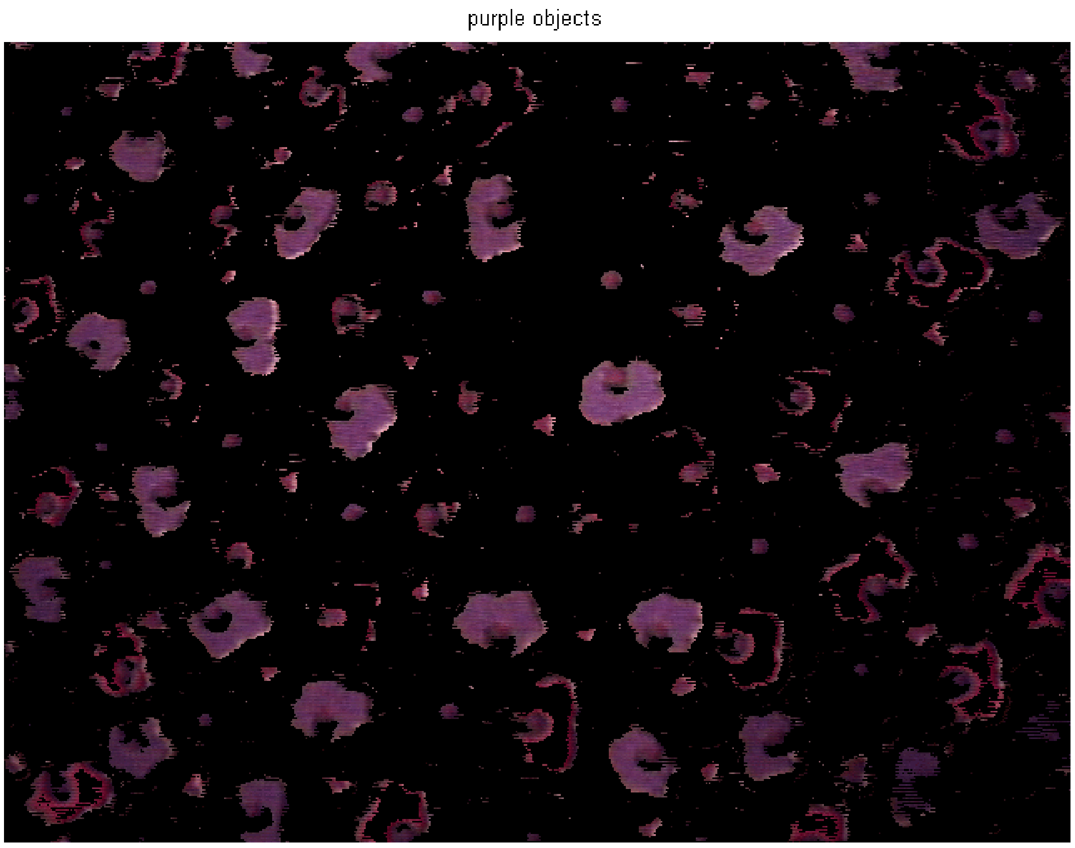
```
rgb_label = repmat(label, [1 1 3]);
segmented_images = repmat(uint8(0), [size(fabric), nColors]);

for count = 1:nColors
  color = fabric;
  color(rgb_label ~= color_labels(count)) = 0;
  segmented_images(:,:,:,count) = color;
end

imshow(segmented_images(:,:,:,1));
title([colorNames{1} ' objects'] );
```



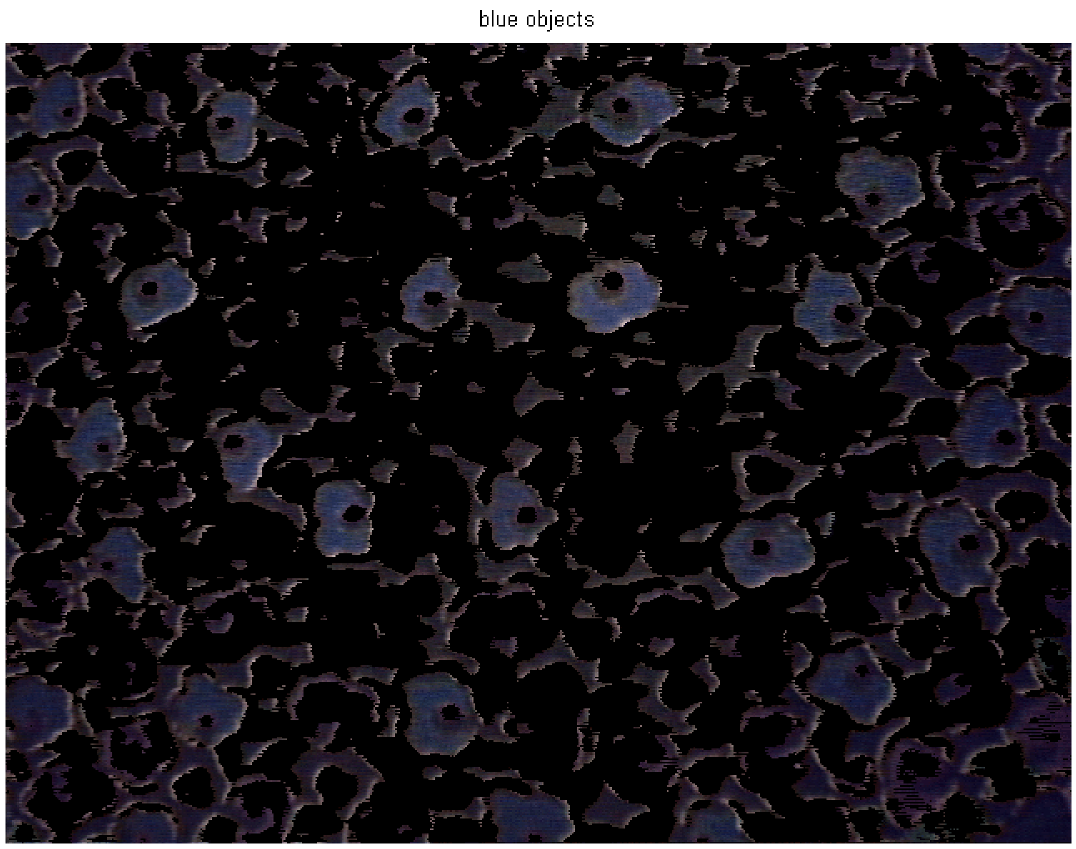
```
imshow(segmented_images(:,:,:,2));
title([colorNames{2} ' objects'] );
```



```
imshow(segmented_images(:,:,:,3));
title([colorNames{3} ' objects'] );
```

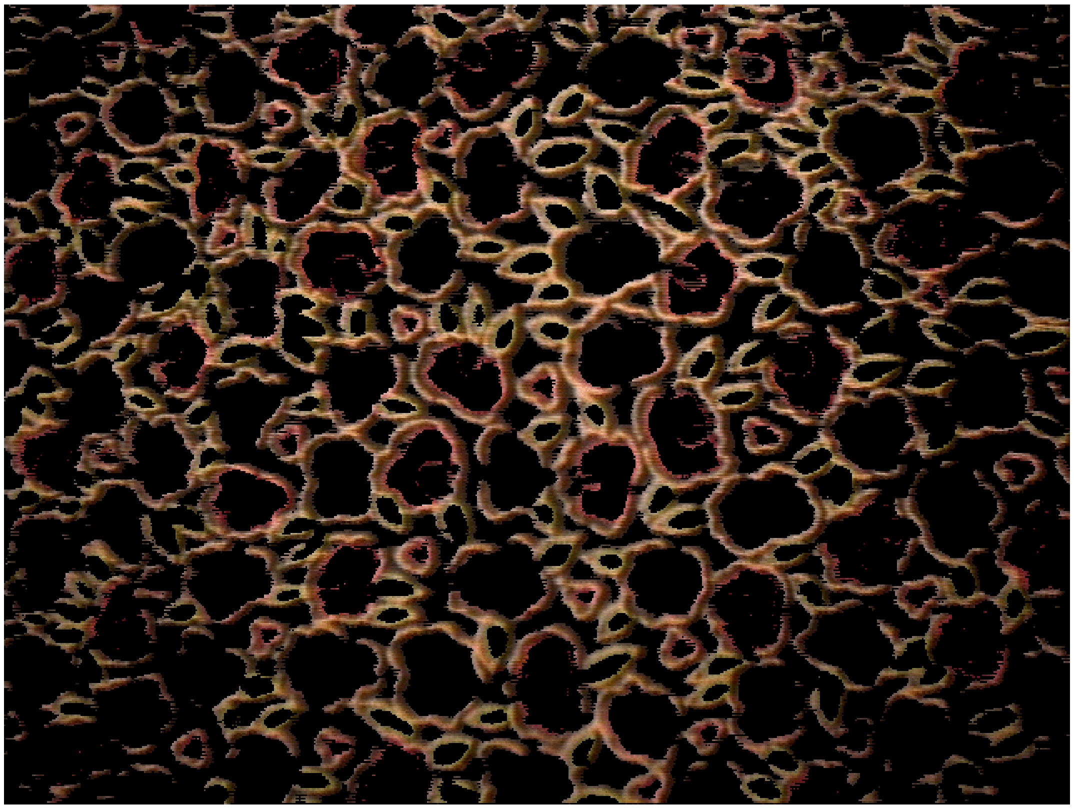


```
imshow(segmented_images(:,:,:,4));
title([colorNames{4} ' objects'] );
```



```
imshow(segmented_images(:,:,:,5));
title([colorNames{5} ' objects'] );
```

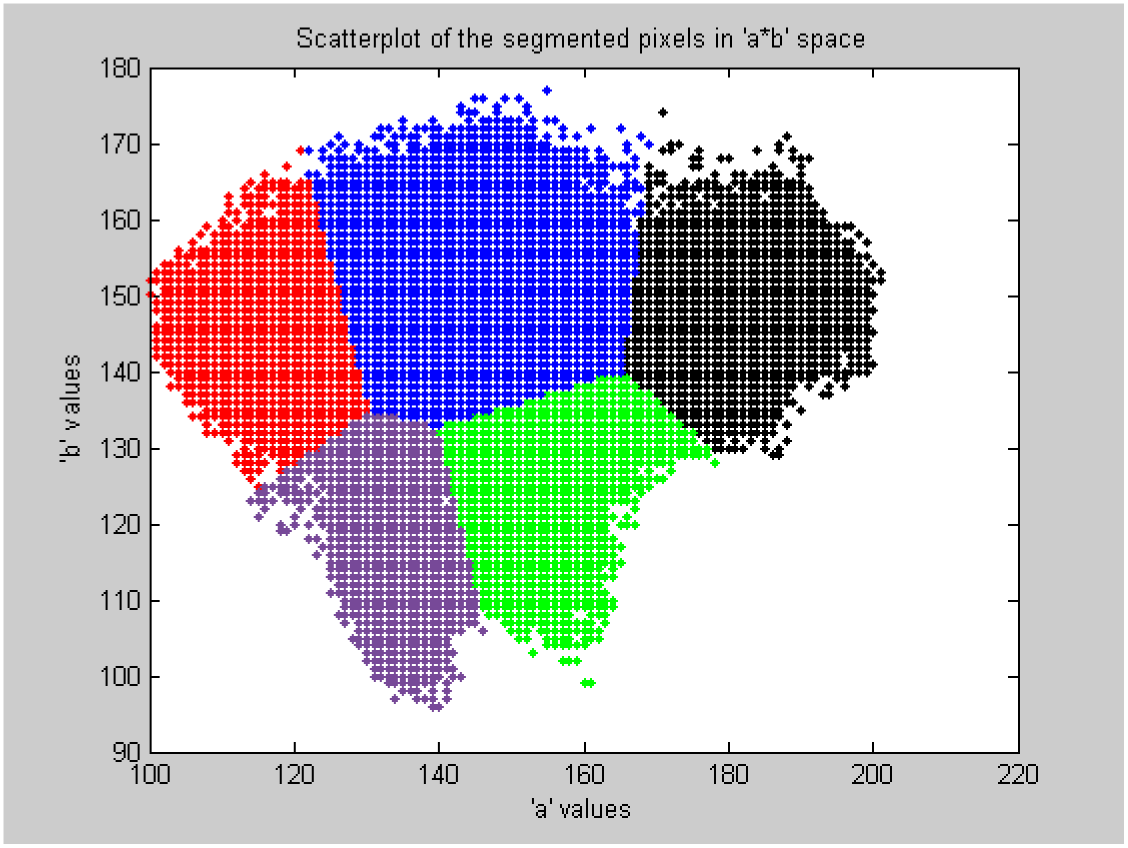
yellow objects



Step 5: Display 'a' and 'b' Values of the Labeled Colors

You can see how well the nearest neighbor classification separated the different color populations by plotting the 'a' and 'b' values of pixels that were classified into separate colors. For display purposes, label each point with its color label.

```
purple = [119/255 73/255 152/255];
plot_labels = {'k', 'r', 'g', purple, 'b', 'y'};

figure
for count = 1:nColors
  h(count) = plot(a(label==count-1),b(label==count-1),'.','MarkerEdgeColor', ...
                  plot_labels{count}, 'MarkerFaceColor', plot_labels{count});
  hold on;
end

title('Scatterplot of the segmented pixels in ''a*b'' space');
xlabel('''a'' values');
ylabel('''b'' values');
```


Scatterplot of the segmented pixels in 'a*b' space
180
170
160
150
140
130
120
110
100
90
'b' values
100
120
140
160
180
200
220
'a' values

Determining the Rate of Acquisition

This example shows how to use the timestamps provided by the GETDATA function, and estimate the device frame rate using MATLAB® functions.

Step 1: Access and Configure a Device.

Create a video input object and access its video source object to configure the desired acquisition rate. The acquisition rate is determined by the value of the device specific FrameRate property of the video source object.

Note, since FrameRate is a device specific property, not all devices may support it.

```
% Access an image acquisition device.
vidobj = videoinput('winvideo', 1);

% Configure the number of frames to log.
vidobj.FramesPerTrigger = 50;

% Skip the first few frames the device provides
% before logging data.
vidobj.TriggerFrameDelay = 5;

% Access the device's video source.
src = getselectedsource(vidobj);

% Determine the device specific frame rates (frames per second) available.
frameRates = set(src, 'FrameRate')
```

```
frameRates = 

    '30.0000'
    '24.0000'
    '8.0000'
```

```
% Configure the device's frame rate to the highest available setting.
src.FrameRate = frameRates{1};
actualRate = str2num( frameRates{1} )
```

```
actualRate =

    30
```

Step 2: Log and Retrieve Data.

Initiate the acquisition and retrieve the logged frames and timestamps.

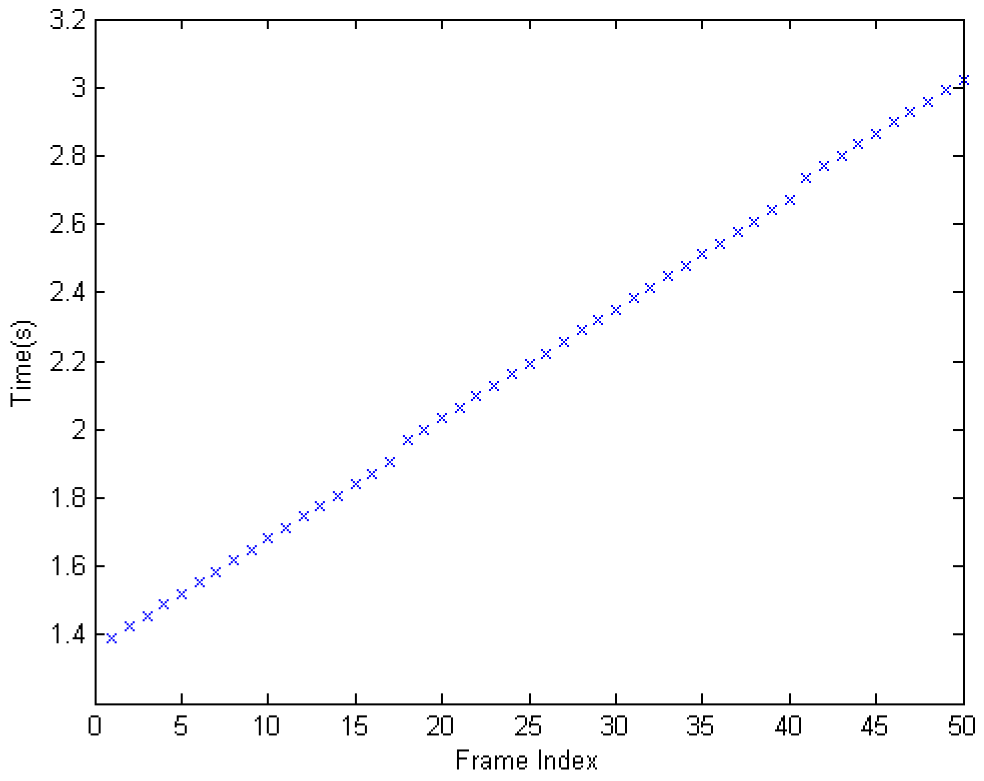
```
% Start the acquisition.
start(vidobj)

% Wait for data logging to end before retrieving data.  Set the wait time
% to be equal to the expected time to acquire the number of frames
% specified plus a little buffer time to accommodate  overhead.
waittime = actualRate * (vidobj.FramesPerTrigger + vidobj.TriggerFrameDelay) + 5;
wait(vidobj, waittime);
```

```
% Retrieve the data and timestamps.
[frames, timeStamp] = getdata(vidobj);
```

Step 3: Calculate the Acquisition Rate.

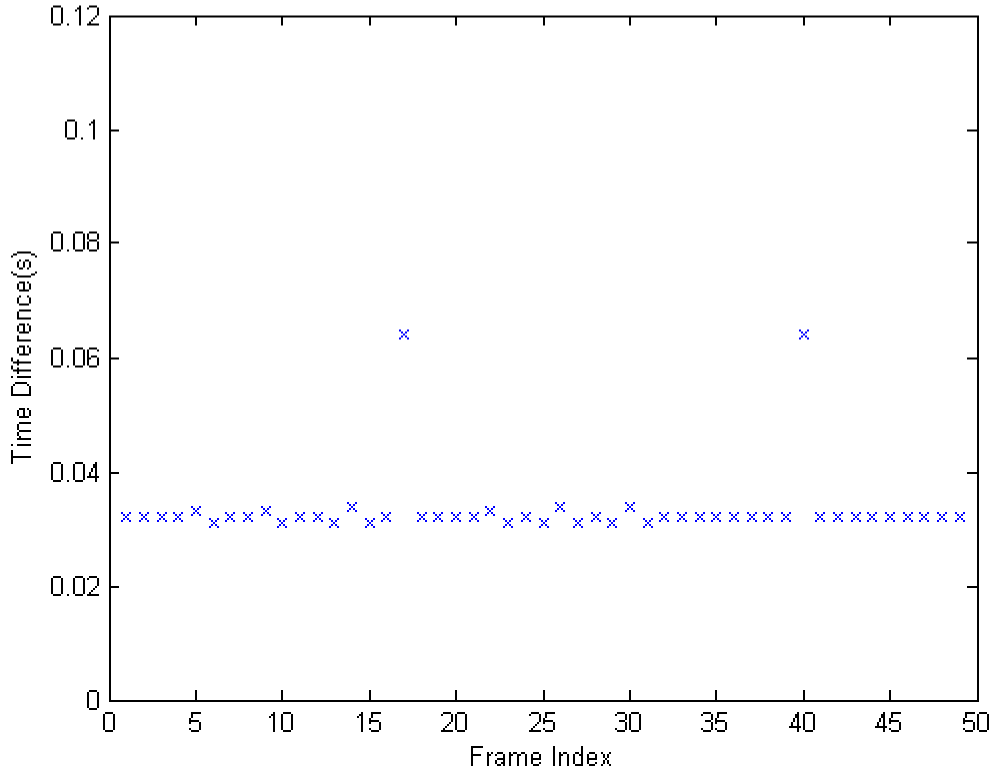
By plotting each frame's timestamp, one can verify that the rate of acquisition is constant.

```
% Graph frames vs time.
plot(timeStamp,'x')
xlabel('Frame Index')
ylabel('Time(s)')
```



The average time difference can also be determined to compare to the expected acquisition rate.

```
% Find the time difference between frames.
diffFrameTime = diff(timeStamp);

% Graph the time differences.
plot(diffFrameTime, 'x');
xlabel('Frame Index')
ylabel('Time Difference(s)')
ylim([0 .12])
```



```
% Find the average time difference between frames.
avgTime = mean(diffFrameTime)
```

```
avgTime =

    0.0333
```

```
% Determine the experimental frame rate.
expRate = 1/avgTime
```

```
expRate =

   30.0245
```

Comparing the time difference between the experimental and the known frame rate, the percent error can be calculated. Since a generic USB web camera is being used as the acquisition device, it is to be expected that the actual device frame rate will fluctuate.

```
% Determine the percent error between the determined and actual frame rate.
diffRates = abs(actualRate - expRate)
```

```
diffRates =

    0.0245
```

```
percentError = (diffRates/actualRate) * 100
percentError =

    0.0817

% Once the video input object is no longer needed, delete
% it and clear it from the workspace.
delete(vidobj)
clear vidobj
```

Laser Tracking

This example shows how to track a moving laser dot.

Using the Image Acquisition Toolbox™, image data streams from a camera are acquired directly into MATLAB®. These images are used to track objects in the camera's view. For this example, the object being tracked is the dot produced by a laser pointer.

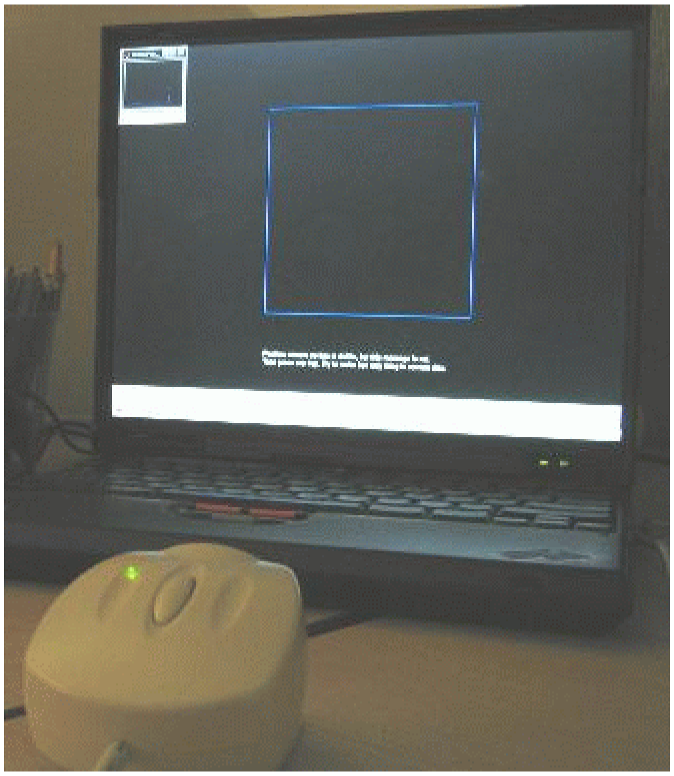
The monitor of a computer running MATLAB is placed in the camera's view while a laser pointer shines a red dot on a MATLAB figure window. The camera is used to acquire images of the MATLAB figure window while the laser pointer moves around. By tracking the movement of the laser dot, the laser pointer can be used as a pointer device similar to a mouse.

The first task involves calibrating the data to establish a relationship between the acquired image pixels and the MATLAB figure axes. Once this relationship is established, the laser dot can be tracked as it moves around within the MATLAB figure window.

This example uses a set of utility functions that aid in the processing of images. These utility functions require the Image Processing Toolbox™.

Physical Setup

Focus a camera onto the screen of a computer that MATLAB is running on.



It is best to have the ambient light in the room minimized. This example has been successfully run in auditoriums using a projector.

Configure the Acquisition

An image acquisition device will be used to acquire image data to perform the calibration and laser tracking. The device used will be a generic Windows® video WebCam.

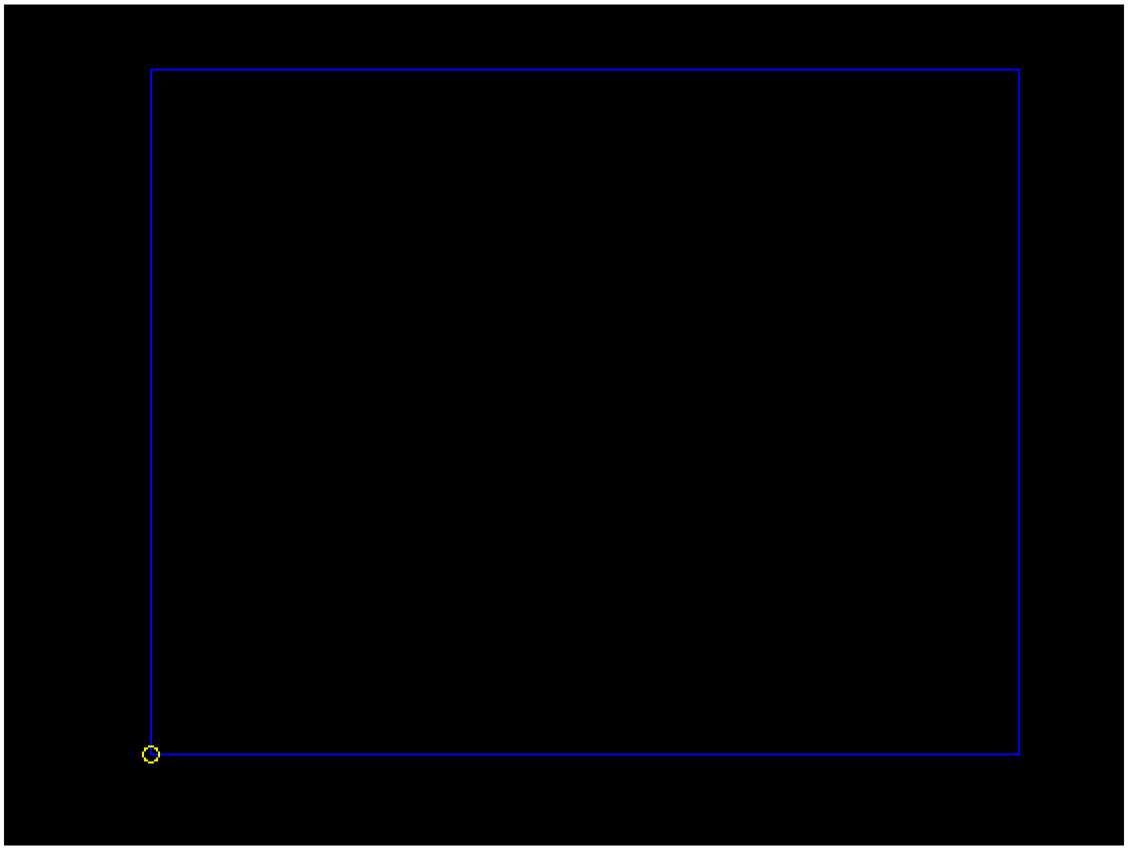
```
% Access and configure a device.
vid = videoinput('winvideo', 1, 'RGB24_320x240');
vid.FramesPerTrigger = 1;
vid.TriggerRepeat = Inf;
triggerconfig(vid,'manual')
```

Create the Calibration Screen

The calibration screen created is purposely set to black to get the best contrast for the laser pointer. Some systems work better when the window's colors are set dark for the title bars.

Since a red laser will be used, the red plane of the image is the only color plane of interest. The calibration square is made blue in order to make it appear "invisible" in the red plane.

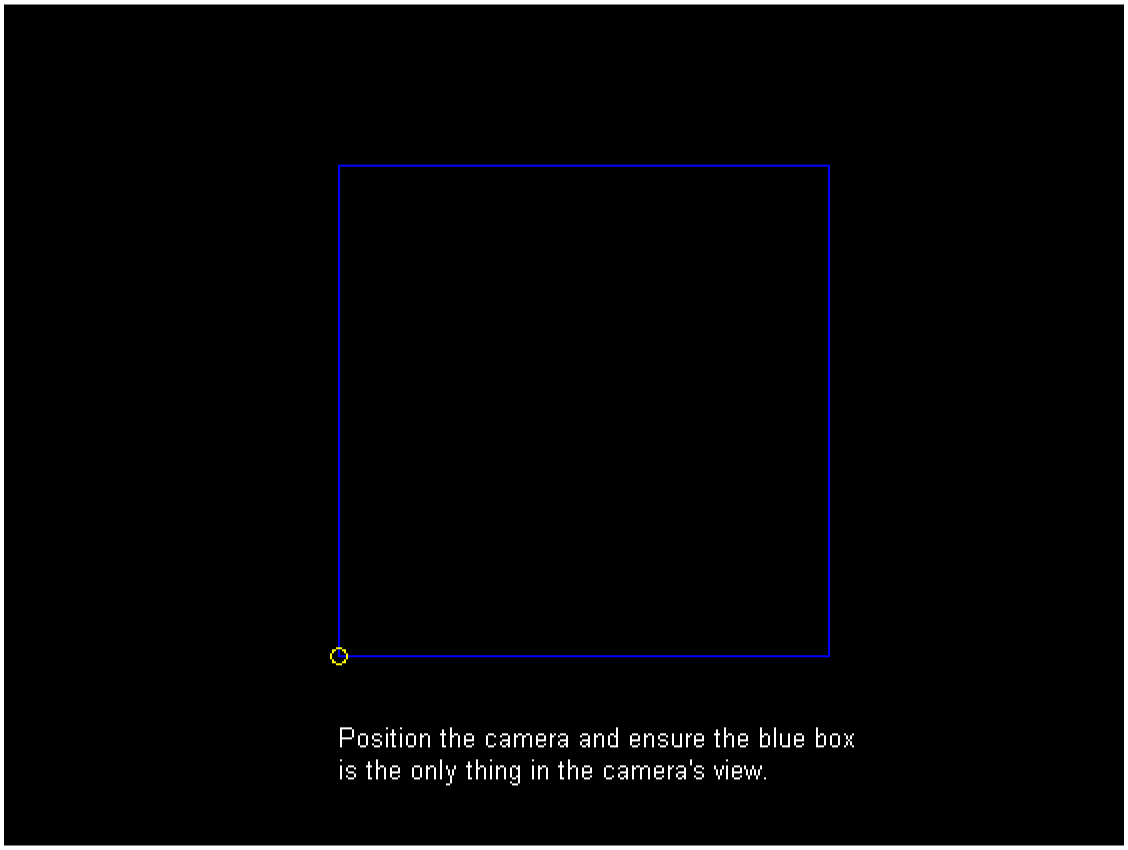
```
% Create the laser figure window.
laserFig = figure;
hBox = plot([0 0 1 1 0], [0 1 1 0 0], 'b-');
hold on

% Set up calibration screen. Modify the cursor so it does not
% interfere with the calibration.
hTarget = plot(0, 0, 'yo');
ax = gca;
ax.Color = [0, 0, 0];
laserFig.Color = [0, 0, 0];
laserFig.Menubar = 'none';
laserFig.Pointer = 'custom';
laserFig.PointerShapeCData = NaN(16, 16);
```



Position the Camera

Position the camera such that only the blue square is visible.

```
% Display positioning information.
posText = sprintf('%s\n%s', ...
    'Position the camera and ensure the blue box', ...
    'is the only thing in the camera''s view.');
infoText = text(0, -0.2,  posText, 'Color', [1 1 1]);
axis([-0.2 1.2 -0.2 1.2])
axis equal
```


Position the camera and ensure the blue box
is the only thing in the camera's view.

```
% Using the preview window, request that the camera be positioned such
% that the view is of the blue box and little else.
preview(vid)
smallFigPos = laserFig.Position;
laserFig.Position = get(0, 'ScreenSize');
disp('Waiting for camera to be positioned...press any key to continue.')
pause
```

```
Waiting for camera to be positioned...press any key to continue.
```

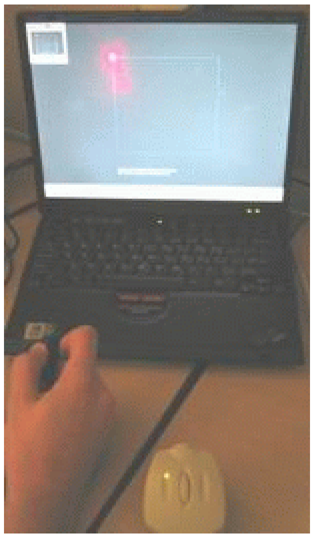
Perform Image Calibration

Now that the camera is focused on the right area, a target is drawn at each of the four corners of the box. The calibration is performed by aiming the laser on each corner of the blue square, allowing a relationship to be established between the camera pixel coordinates (the image) and MATLAB axis coordinates (the square). For each target displayed:

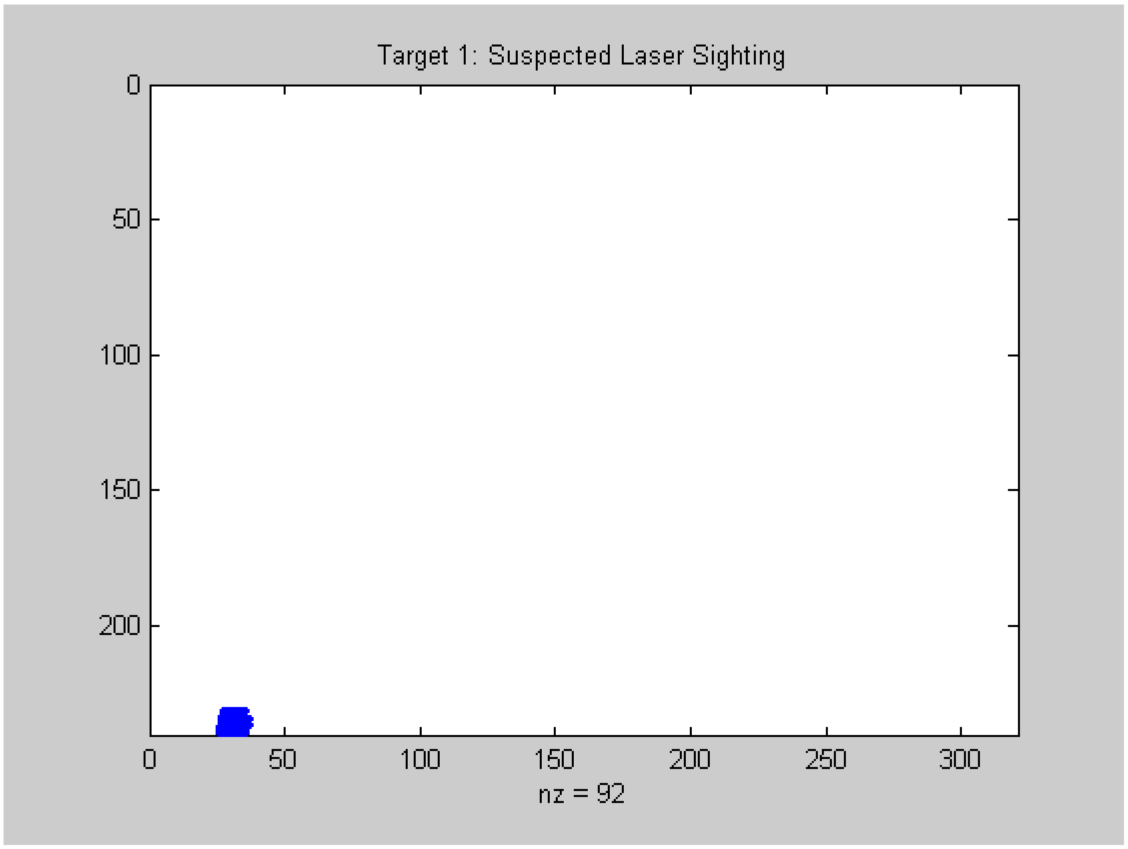
- output a sound indicating the laser should be aimed
- output a sound indicating a frame is about to be acquired
- trigger the acquisition device
- access the acquired image frame and determine the laser position in pixel coordinates

The laser position is determined by thresholding the red plane and looking for high intensity values. Some additional processing is performed to make sure the laser is not obscured by ghost images caused by poor optics in some WebCams. It is also verified that a laser dot was actually present on the screen.

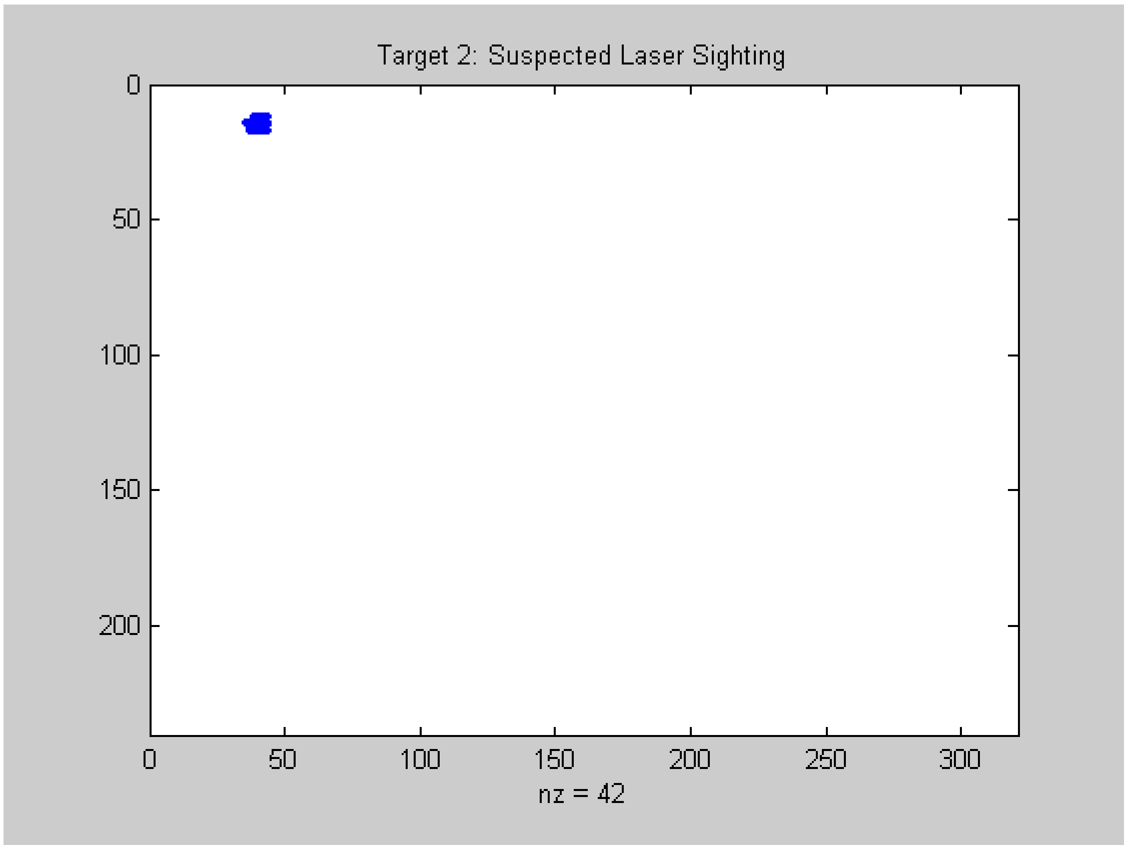
```
% Provide calibration instructions.
calibText = sprintf('%s\n%s', ...
    'Aim the laser pointer on each target as it appears.', ...
    'Hold the laser on the target until the target moves.');
infoText.String = calibText;
```



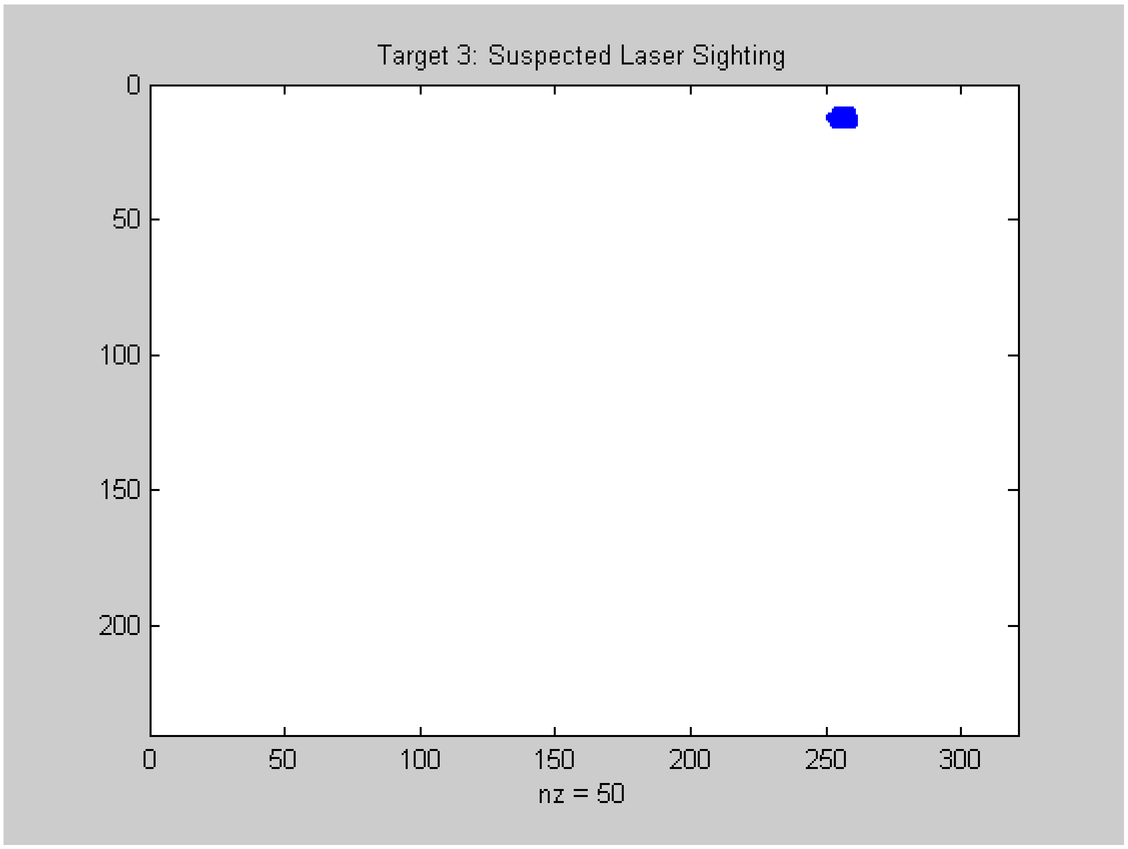
```
% Start the acquisition and create a new figure to display
% calibration results in a MATLAB SPY plot.
start(vid)
spyFig = figure;

% Target 1...
figure(laserFig);
hTarget.XData = 0;
hTarget.YData = 0;
sound(1), pause(2)
sound(1), trigger(vid);
acqResults{1} = getdata(vid, 1);

[xCalib(1), yCalib(1), laserSights] = util_findlaser(acqResults{1});
figure(spyFig);
spy(laserSights)
title('Target 1: Suspected Laser Sighting')
```



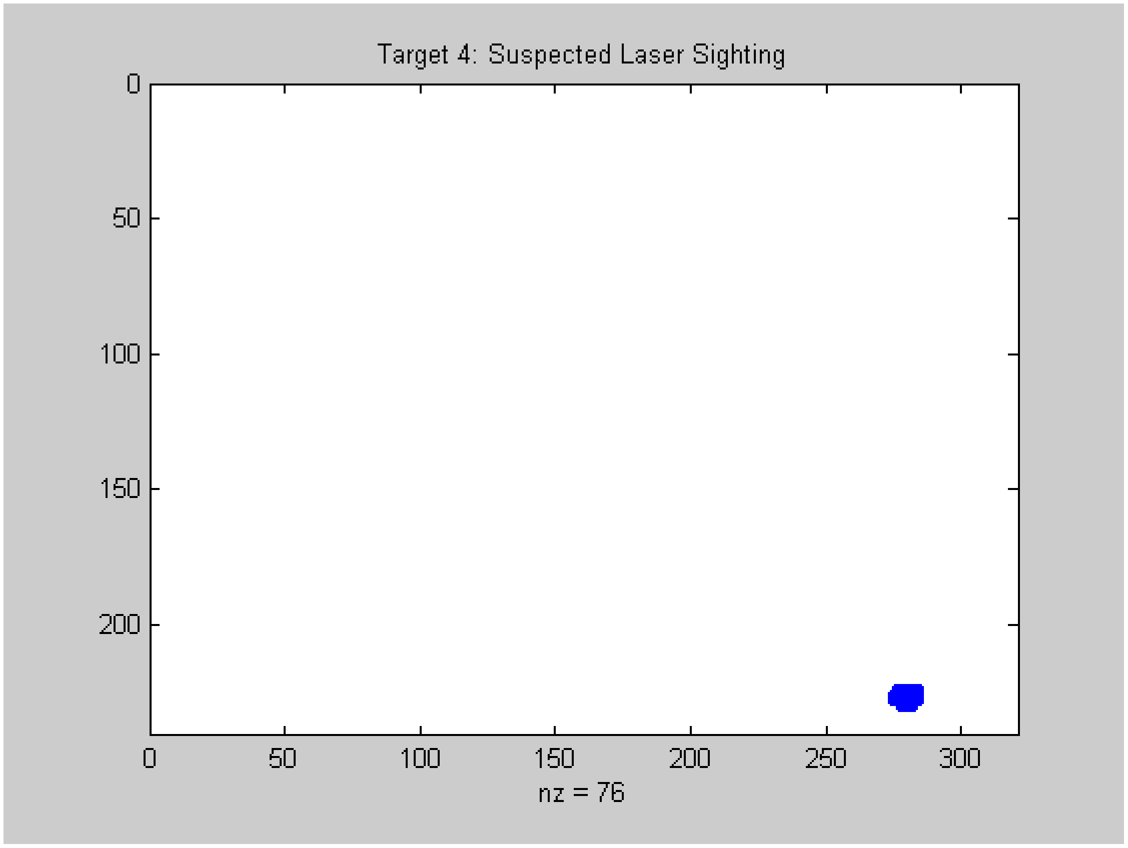
```
% Target 2...
figure(laserFig);
hTarget.XData = 0;
hTarget.YData = 1;
sound(1), pause(2)
sound(1), trigger(vid);
acqResults{2} = getdata(vid, 1);

[xCalib(2), yCalib(2), laserSights] = util_findlaser(acqResults{2});
figure(spyFig);
spy(laserSights)
title('Target 2: Suspected Laser Sighting')
```



```
% Target 3...
figure(laserFig);
hTarget.XData = 1;
hTarget.YData = 1;
sound(1), pause(2)
sound(1), trigger(vid);
acqResults{3} = getdata(vid, 1);

[xCalib(3), yCalib(3), laserSights] = util_findlaser(acqResults{3});
figure(spyFig);
spy(laserSights)
title('Target 3: Suspected Laser Sighting')
```



```
% Target 4...
figure(laserFig);
hTarget.XData = 1;
hTarget.YData = 0;
sound(1), pause(2)
sound(1), trigger(vid);
acqResults{4} = getdata(vid, 1);

[xCalib(4), yCalib(4), laserSights] = util_findlaser(acqResults{4});
figure(spyFig);
spy(laserSights)
title('Target 4: Suspected Laser Sighting')
```

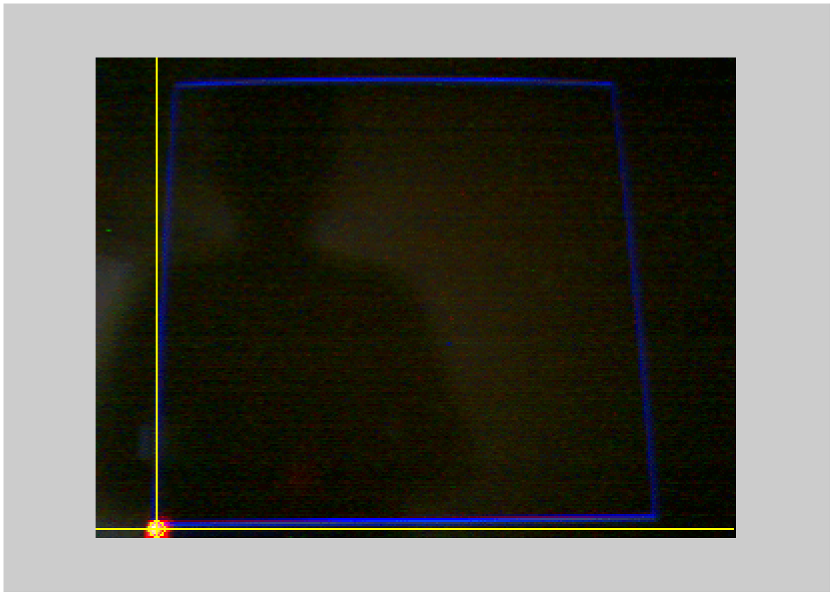


```
% Close the SPY plot and stop the acquisition.
close(spyFig)
stop(vid);
```

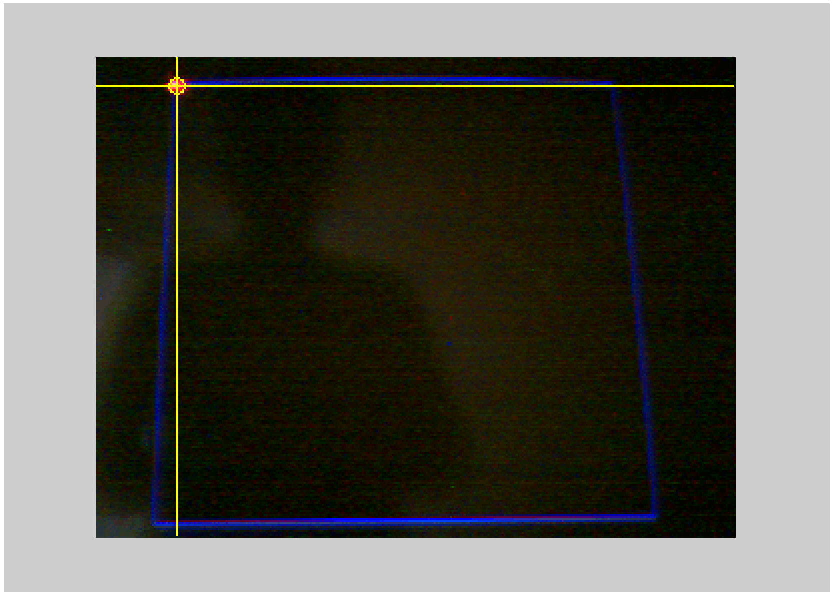
Calibration Results

Plot the acquired image and the calculated laser pointer coordinates for each target. Since the yellow crosshairs are positioned at the proper location in each image, the processing results are validated.

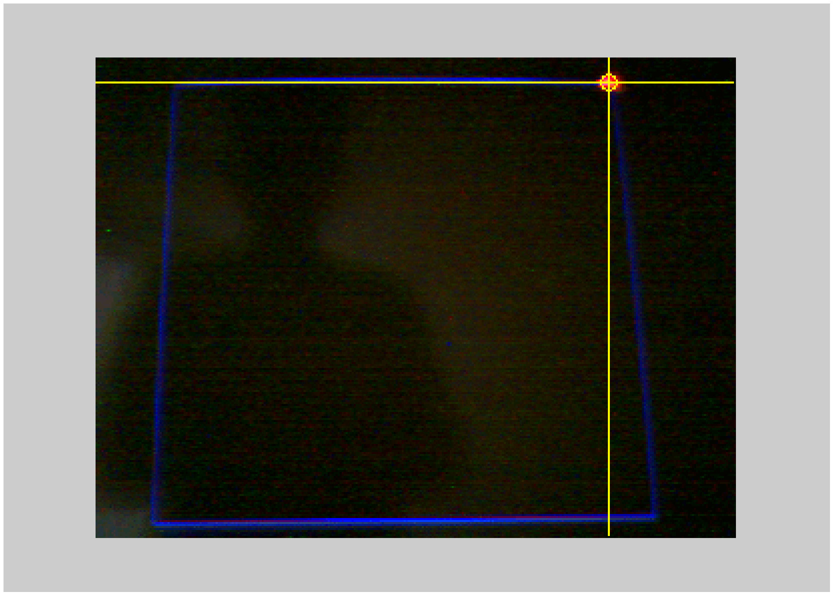
```
% Target 1 results...
calibFig = figure;
util_plotpos(acqResults{1}, xCalib(1), yCalib(1));
```



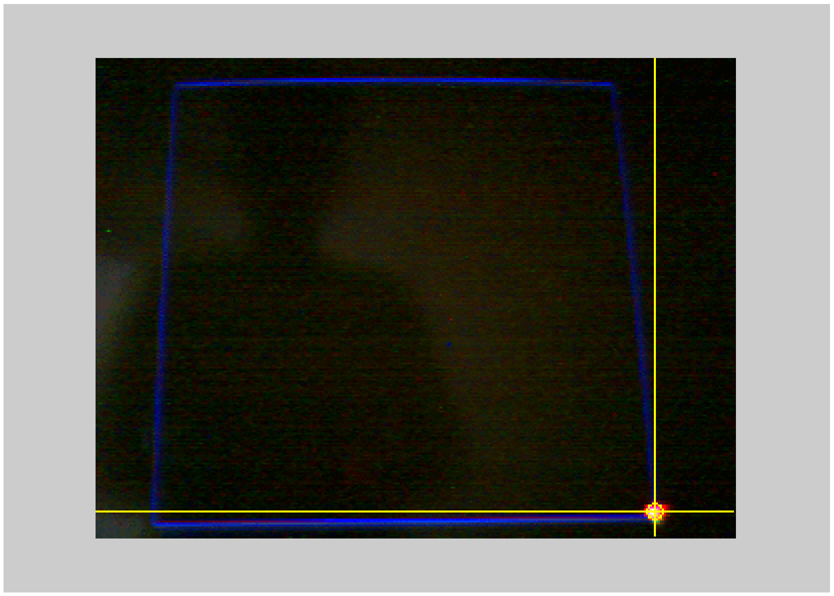
```
% Target 2 results...
util_plotpos(acqResults{2}, xCalib(2), yCalib(2));
```



```
% Target 3 results...
util_plotpos(acqResults{3}, xCalib(3), yCalib(3));
```



```
% Target 4 results...
util_plotpos(acqResults{4}, xCalib(4), yCalib(4));
```



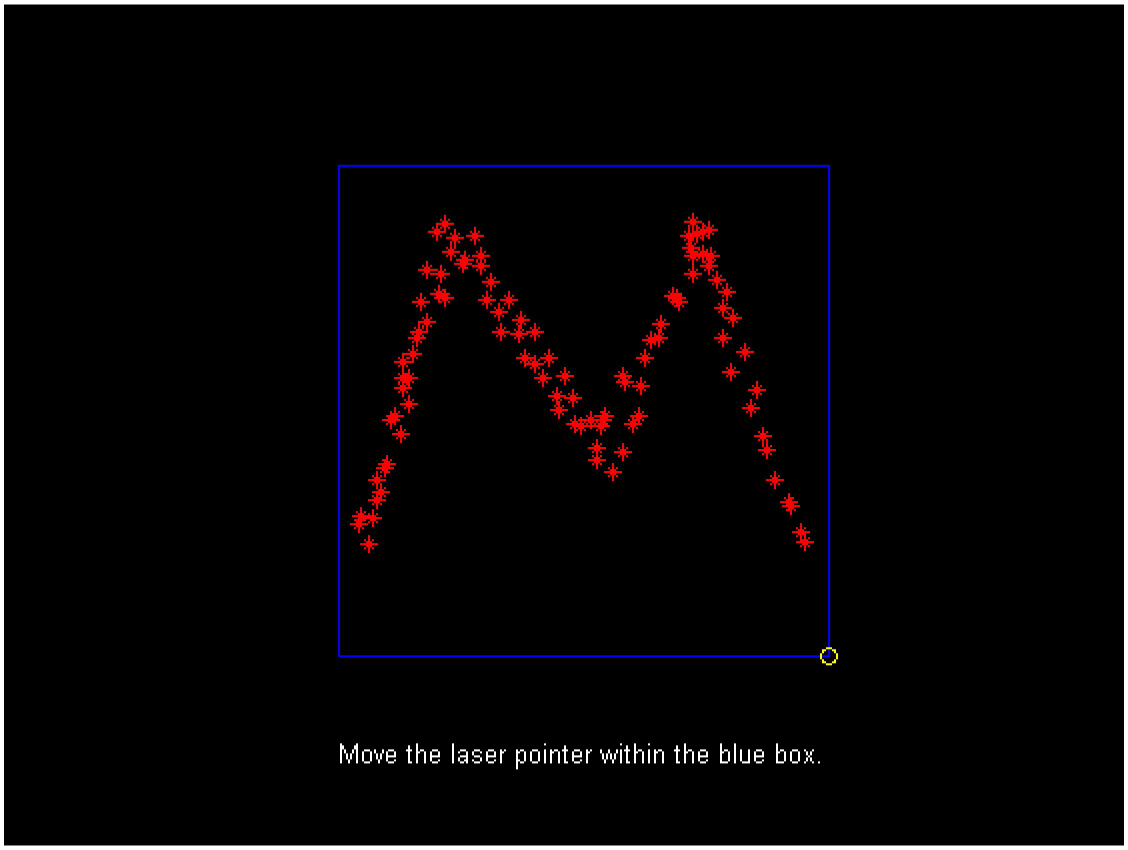
```
% Close the figure illustrating calibration results.
close(calibFig)
```

Laser Tracking

Start the acquisition and process the acquired data a set number of times. The processing consists of locating the laser in the acquired image and determining the laser positions in pixel and MATLAB axis coordinates.

In order to make things interesting, using the laser pointer, attempt to "draw" the letter 'M' (for MATLAB) within the blue box.

```
% Update instructions on laser screen.
figure(laserFig);
infoText.String = 'Move the laser pointer within the blue box.';

% Start the acquisition. For each iteration:
%
% * output a sound to indicate a frame is about to be acquired
% * trigger the device
% * process the acquired image and locate the laser
% * convert pixel coordinates to MATLAB axis coordinates
laser.x = [];
laser.y = [];
start(vid)
for i = 1:100,
    % Acquire an image frame and determine the
    % camera pixel coordinates.
    sound(1), trigger(vid);
    frame = getdata(vid, 1);
    [x, y] = util_findlaser(frame);

    if ~isnan(x) && ~isnan(y),
        % If coordinates were valid, ensure the camera pixel coordinate
        % was in the calibration range.
        x = max([x min(xCalib([1 2]))]);
        x = min([x max(xCalib([3 4]))]);
        y = min([y max(yCalib([1 4]))]);
        y = max([y min(yCalib([2 3]))]);

        % Determine spatial transformation from the unit square calibration points.
        tform = cp2tform([xCalib(:) yCalib(:)], [0 0; 0 1; 1 1; 1 0], 'projective');
        xyScreen = tformfwd([x, y], tform);
        xScreen = xyScreen(1);
        yScreen = xyScreen(2);

        % Ensure the new coordinates remain within the unit square.
        xScreen = min([xScreen 1]);
        xScreen = max([xScreen 0]);
        yScreen = min([yScreen 1]);
        yScreen = max([yScreen 0]);

        % Store the new MATLAB axis coordinates.
        laser.x = [laser.x(:); xScreen];
        laser.y = [laser.y(:); yScreen];
    end
end

% Plot the tracked laser positions.
laserFig.Position = smallFigPos;
plot(laser.x, laser.y, 'r*');
```



```
% Close the laser figure.
close(laserFig);

% Stop the acquisition, remove the object from memory,
% and clear the variable.
stop(vid)
delete(vid)
clear vid
```

Logging Data at Constant Intervals

This example shows how to log data at intervals instead of logging the entire acquisition.

In certain applications, it may not be necessary to log every frame provided by an image acquisition device. In fact, it may be more practical and resourceful to log frames at certain intervals.

To log frames at a constant interval, configure the video input object's FrameGrabInterval property. Configuring the property to an integer value N specifies that every Nth frame should be logged, starting with the first frame.

Note, specifying a FrameGrabInterval value does not modify the rate at which a device is providing frames (device frame rate). It only specifies the interval at which frames are logged.

Step 1: Access and Configure a Device.

Create a video input object and configure the desired logging interval. The logging interval is determined by the value of the FrameGrabInterval property.

```
% Access an image acquisition device.
vidobj = videoinput('winvideo', 1);

% Configure the number of frames to log.
framesToLog = 9;
vidobj.FramesPerTrigger = framesToLog;

% Configure the logging interval. This specifies that
% every 10th frame provided by the device is to be logged.
grabInterval = 10;
vidobj.FrameGrabInterval = grabInterval;

% Access the device's video source and configure the device's frame rate.
% FrameRate is a device specific property, therefore, it may not be supported by
% some devices.
frameRate = 30;
src = getselectedsource(vidobj);
src.FrameRate = num2str(frameRate);
```

Step 2: Log and Retrieve Data.

Initiate the acquisition of images and retrieve the logged frames and their timestamps.

```
% Start the acquisition.
start(vidobj)

% Wait for the acquisition to end.
wait(vidobj, 10)

% Retrieve the data.
[frames, timeStamp] = getdata(vidobj);
```

Step 3: Calculate the Time Difference Between Frames.

Knowing the device's actual frame rate and the grab interval at which frames were logged, the number of frames logged per second can be calculated.

```
% Number of frames logged per second.
loggedPerSec = frameRate/grabInterval


loggedPerSec =

     3
```

Knowing the number of frames logged per second, the expected time interval between each logged frame can be calculated and compared.

```
% Expected number of seconds between each logged frame.
loggingRate = 1/loggedPerSec


loggingRate =

    0.3333

% Actual time difference between each logged frame.
% Note that frames were logged at a constant interval.
diff(timeStamp')


ans =

    0.3332    0.3338    0.3331    0.3332    0.3330    0.3332    0.3331    0.3330

% Determine the average time difference between frames.
avgDiff = mean(diff(timeStamp'))


avgDiff =

    0.3332

percentError = ( abs(loggingRate-avgDiff) ) * 100


percentError =

    0.0125

% Once the video input object is no longer needed, delete
% it and clear it from the workspace.
delete(vidobj)
clear vidobj
```

Video Display with Live Histogram

This example shows how to set up and display a live histogram.

The Image Acquisition Toolbox™ together with the Image Processing Toolbox™ can be used to display a video feed with a live histogram. This can be useful when calibrating camera settings such as aperture using manual controls. This example shows how to use the PREVIEW function, its associated custom update function and the IMHIST function to place a video preview window adjacent to a live histogram. The techniques here can be used to display other live information too. For example, a live video feed can be placed next to a filtered version of the video.

Watch a clip of the video feed and histogram. (8 seconds)

Setup Video Object and Figure

```
% Access an image acquisition device.
vidobj = videoinput('winvideo');

% Convert the input images to grayscale.
vidobj.ReturnedColorSpace = 'grayscale';
```

An image object of the same size as the video is used to store and display incoming frames.

```
% Retrieve the video resolution.
vidRes = vidobj.VideoResolution;

% Create a figure and an image object.
f = figure('Visible', 'off');

% The Video Resolution property returns values as width by height, but
% MATLAB images are height by width, so flip the values.
imageRes = fliplr(vidRes);

subplot(1,2,1);

hImage = imshow(zeros(imageRes));

% Set the axis of the displayed image to maintain the aspect ratio of the
% incoming frame.
axis image;
```

Specify the UpdatePreviewWindowFcn callback function that is called each time a new frame is available. The callback function is responsible for displaying new frames and updating the histogram. It can also be used to apply custom processing to the frames. More details on how to use this callback can be found in the documentation for the PREVIEW function. This callback function itself is defined in the file update_livehistogram_display.m.

```
setappdata(hImage,'UpdatePreviewWindowFcn',@update_livehistogram_display);
```

Define the Callback Function

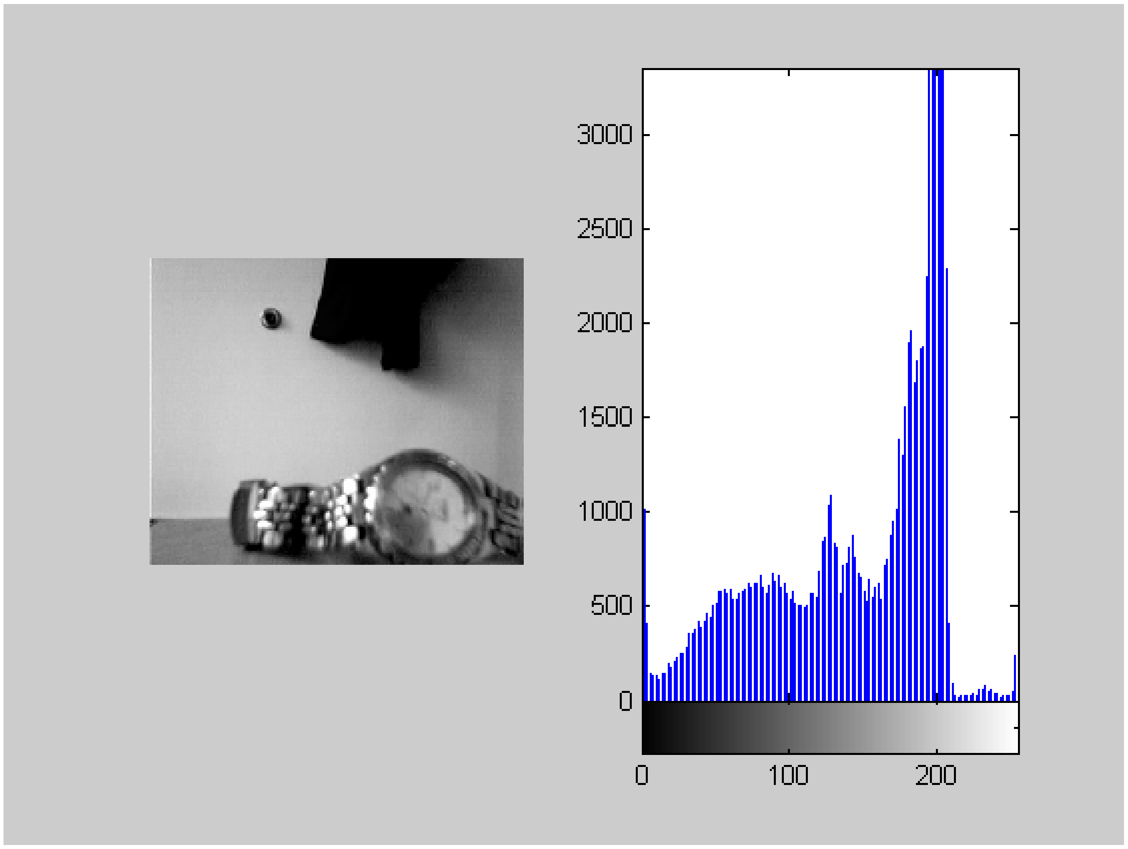
```
% Here are the contents of update_livehistogram_display.m which contains
% the callback function.
dbtype('update_livehistogram_display.m')
```

```
1     function update_livehistogram_display(obj,event,hImage)
```

```
% This callback function updates the displayed frame and the histogram.

% Copyright 2007-2017 The MathWorks, Inc.
%

% Display the current image frame.
set(hImage, 'CData', event.Data);

% Select the second subplot on the figure for the histogram.
subplot(1,2,2);

% Plot the histogram. Choose 128 bins for faster update of the display.
imhist(event.Data, 128);

% Refresh the display.
drawnow
```

Start Previewing

```
% The PREVIEW function starts the camera and display. The image on which to
% display the video feed is also specified.
preview(vidobj, hImage);

% View the histogram for 30 seconds.
pause(30);
```



Above is a sample image of the histogram and video feed.

```
% Stop the preview image and delete the figure.
stoppreview(vidobj);
delete(f);
```

Once the video input object is no longer needed, delete and clear the associated variable.

```
delete(vidobj)
clear vidobj
```

Live Motion Detection Using Optical Flow

This example shows how to create a video algorithm to detect motion using optical flow technique. This example uses the Image Acquisition Toolbox™ System object™ along with Computer Vision Toolbox™ System objects.

Introduction

This example streams images from an image acquisition device to detect motion in the live video. It uses the optical flow estimation technique to estimate the motion vectors in each frame of the live video sequence. Once the motion vectors are determined, we draw it over the moving objects in the video sequence.

Initialization

Create the Video Device System object.

```
vidDevice = imaq.VideoDevice('winvideo', 1, 'YUY2_320x240', ...
                             'ReturnedColorSpace', 'rgb', ...
                             'DeviceProperties.Brightness', 130, ...
                             'DeviceProperties.Sharpness', 50);
```

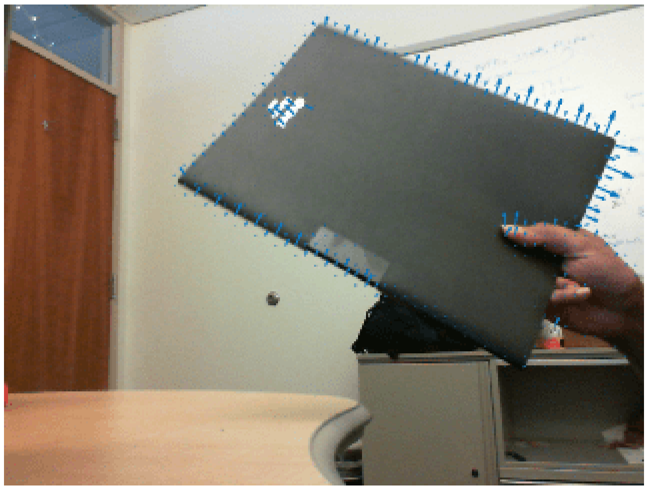
Create a System object to estimate direction and speed of object motion from one video frame to another using optical flow.

```
opticFlow = opticalFlowHS;
```

Stream Acquisition and Processing Loop

Create a processing loop to perform motion detection in the input video. This loop uses the System objects you instantiated above.

```
% Set up for stream
nFrames = 0;
while (nFrames<100)     % Process for the first 100 frames.
    % Acquire single frame from imaging device.
    frameRGB = vidDevice();

    % Compute the optical flow for that particular frame.
    flow = estimateFlow(opticFlow,rgb2gray(frameRGB));

    imshow(frameRGB)
    hold on
    plot(flow,'DecimationFactor',[5 5],'ScaleFactor',25)
    hold off

    % Increment frame count
    nFrames = nFrames + 1;
end
```



Summary

In the figure window, you can see that the example detected the motion of the black file. The moving objects are represented using the vector field lines as seen in the image.

Release

Here you call the release method on the System objects to close any open files and devices.

```
release(vidDevice);
```

Synchronizing Two NI Frame Grabbers

This example shows how to synchronize the start of image capture using Image Acquisition Toolbox™ and two NI® RTSI capable frame grabbers.

It is often necessary to synchronize two or more frame grabbers very closely. For example, you could record synchronized video during an experiment that is costly or impossible to duplicate. Because of the nature of the experiment, it would be beneficial to use RTSI to ensure the most reliable connection between your NI PCI-1409 and PCIe-1430 frame grabbers.

Configure the PCI-1409

Using the Image Acquisition Toolbox, create the video input object to record video and set up the parameters for acquisition.

```
% Create the object.
vid1409 = videoinput('ni', 1);
% Set to acquire approximately one second of frames per trigger.
vid1409.FramesPerTrigger = 30;
```

You can use either card to trigger the other, but this example uses the PCIe-1430 to trigger the PCI-1409. See what triggering settings are available for the PCI-1409.

```
% See the possible settings.
triggerinfo(vid1409)
```

```
   Valid Trigger Configurations:

      TriggerType:   TriggerCondition:   TriggerSource:
      'immediate'    'none'              'none'
      'manual'       'none'              'none'
      'hardware'     'fallingEdge'       'external0'
      'hardware'     'fallingEdge'       'external1'
      'hardware'     'fallingEdge'       'external2'
      'hardware'     'fallingEdge'       'external3'
      'hardware'     'fallingEdge'       'rtsi0'
      'hardware'     'fallingEdge'       'rtsi1'
      'hardware'     'fallingEdge'       'rtsi2'
      'hardware'     'fallingEdge'       'rtsi3'
      'hardware'     'fallingEdge'       'rtsi4'
      'hardware'     'fallingEdge'       'rtsi5'
      'hardware'     'fallingEdge'       'rtsi6'
      'hardware'     'risingEdge'        'external0'
      'hardware'     'risingEdge'        'external1'
      'hardware'     'risingEdge'        'external2'
      'hardware'     'risingEdge'        'external3'
      'hardware'     'risingEdge'        'rtsi0'
      'hardware'     'risingEdge'        'rtsi1'
      'hardware'     'risingEdge'        'rtsi2'
      'hardware'     'risingEdge'        'rtsi3'
      'hardware'     'risingEdge'        'rtsi4'
      'hardware'     'risingEdge'        'rtsi5'
      'hardware'     'risingEdge'        'rtsi6'
```

Set the video input object for hardware triggering off of RTSI line 1 upon a rising edge.

```
% Set the triggering configuration.
triggerconfig(vid1409, 'hardware', 'risingEdge', 'rtsi1');
```

Configure the PCIe-1430

Create the video input object to record video and set up the parameters for acquisition and for driving RTSI1 high when the acquisition starts.

```
% Create the object.
vid1430 = videoinput('ni', 2);
% Set to acquire approximately one second of frames per trigger.
vid1430.FramesPerTrigger = 30;
```

In order to drive the PCI-1409's RTSI line, you need to set the correct line and polarity on the PCIe-1430. In addition, you need to determine what frame grabber event will drive the RTSI line. You can see a list of events that are available by looking at the device-specific source properties that end in “DriveLine” and “DrivePolarity”:

```
% Get the currently selected source.
src = getselectedsource(vid1430);
% Display the properties and their possible settings.
set(src)

  General Settings:
    Tag

  Device Specific Properties:
    AcquisitionDoneDriveLine: [ {none} | external0 | rtsi0 | rtsi1 | rtsi2 | rtsi3 | rtsi4 | rts
    AcquisitionDoneDrivePolarity: [ {activeHigh} | activeLow ]
    AcquisitionInProgressDriveLine: [ {none} | external0 | rtsi0 | rtsi1 | rtsi2 | rtsi3 | rtsi4
    AcquisitionInProgressDrivePolarity: [ {activeHigh} | activeLow ]
    ExternalTriggerLineFilter: [ off | {on} ]
    FrameDoneDriveLine: [ {none} | external0 | rtsi0 | rtsi1 | rtsi2 | rtsi3 | rtsi4 | rtsi5 | r
    FrameDoneDrivePolarity: [ {activeHigh} | activeLow ]
    FrameStartDriveLine: [ {none} | external0 | rtsi0 | rtsi1 | rtsi2 | rtsi3 | rtsi4 | rtsi5 |
    FrameStartDrivePolarity: [ {activeHigh} | activeLow ]
    HSyncDriveLine: [ {none} | external0 | rtsi0 | rtsi1 | rtsi2 | rtsi3 | rtsi4 | rtsi5 | rtsi6
    HSyncDrivePolarity: [ {activeHigh} | activeLow ]
    RTSITriggerLineFilter: [ off | {on} ]
    VSyncDriveLine: [ {none} | external0 | rtsi0 | rtsi1 | rtsi2 | rtsi3 | rtsi4 | rtsi5 | rtsi6
    VSyncDrivePolarity: [ {activeHigh} | activeLow ]
```

In this case, you want to drive RTSI line 1 high when the acquisition is in progress. This ensures that the line is driven high as soon as the acquisition begins. To do this, you need to set the acquisition in progress drive line to 'rtsi1':

```
% Set to drive RTSI1 high when the acquisition begins.
src.AcquisitionInProgressDriveLine = 'rtsi1';
```

Looking at the output above, you can see that the polarity for the acquisition in progress event is already set to 'activeHigh', so you do not need to set it.

Note that the maximum number of lines that you can drive is hardware dependent and will possibly vary between devices.

At this point you are set to acquire approximately one second of frames from each device when the PCIe-1430 is started.

Start the Image Acquisition

You can now start the PCI-1409 video input object and see that it is waiting for a hardware trigger.

```
start(vid1409);
vid1409
```

```
Summary of Video Input Object Using 'PCI/PXI-1409'.

   Acquisition Source(s):  Channel 0, Channel 1, Channel 2, and
                           Channel 3 are available.

  Acquisition Parameters:  'Channel 0' is the current selected source.
                           30 frames per trigger using the selected source.
                           'img1' video data to be logged upon START.
                           Grabbing first of every 1 frame(s).
                           Log data to 'memory' on trigger.

      Trigger Parameters:  1 'hardware' trigger(s).

                  Status:  Waiting for trigger 1 of 1.
                           0 frames acquired since starting.
                           0 frames available for GETDATA.
```

You can now display a summary of the PCIe-1430 video input object and see that it is set up to trigger immediately upon start.

```
vid1430
```

```
Summary of Video Input Object Using 'PCIe-1430'.

   Acquisition Source(s):  Channel 0 is available.

  Acquisition Parameters:  'Channel 0' is the current selected source.
                           30 frames per trigger using the selected source.
                           'img0_Port0' video data to be logged upon START.
                           Grabbing first of every 1 frame(s).
                           Log data to 'memory' on trigger.

      Trigger Parameters:  1 'immediate' trigger(s) on START.

                  Status:  Waiting for START.
                           0 frames acquired since starting.
                           0 frames available for GETDATA.
```

When you start the PCIe-1430 video input object, it will immediately be triggered and begin acquiring. At that moment, the frame grabber will send a signal to the other frame grabber across RTSI line 1, which will cause the PCI-1409 to begin nearly synchronously.

```
start(vid1430)
% Wait on both objects until you are done acquiring.
wait(vid1430), wait(vid1409)
```

Display a Summary of Acquisitions

If you now display a summary you will see that both devices have acquired frames.

```
vid1409
```

```
Summary of Video Input Object Using 'PCI/PXI-1409'.

   Acquisition Source(s):  Channel 0, Channel 1, Channel 2, and
                           Channel 3 are available.

  Acquisition Parameters:  'Channel 0' is the current selected source.
                           30 frames per trigger using the selected source.
                           'img1' video data to be logged upon START.
                           Grabbing first of every 1 frame(s).
                           Log data to 'memory' on trigger.

      Trigger Parameters:  1 'hardware' trigger(s).

                  Status:  Waiting for START.
                           30 frames acquired since starting.
                           30 frames available for GETDATA.
```

and:

```
vid1430
```

```
Summary of Video Input Object Using 'PCIe-1430'.

   Acquisition Source(s):  Channel 0 is available.

  Acquisition Parameters:  'Channel 0' is the current selected source.
                           30 frames per trigger using the selected source.
                           'img0_Port0' video data to be logged upon START.
                           Grabbing first of every 1 frame(s).
                           Log data to 'memory' on trigger.

      Trigger Parameters:  1 'immediate' trigger(s) on START.

                  Status:  Waiting for START.
                           30 frames acquired since starting.
                           30 frames available for GETDATA.
```

Clean Up the Objects

Once the video input objects are no longer needed, delete them and clear them and the reference to the source from the workspace.

```
delete(vid1430)
delete(vid1409)
clear vid1430 vid1409 src
```

Synchronizing an NI Frame Grabber and Data Acquisition Card

This example shows how to synchronize the start of image and data acquisition using Image Acquisition Toolbox™, Data Acquisition Toolbox™, and NI® RTSI capable equipment.

It is often necessary to synchronize two or more acquisition boards very closely. For example, you can record the voltage from an analog sensor, such as a strain gauge, as well as synchronized video during an experiment. For the synchronization/triggering signal, you can use an RTSI cable for a reliable connection between your NI PCI-6229 data acquisition card and PCIe-1430 frame grabber.

Configure the Data Acquisition Board

Using the Data Acquisition Toolbox, create the analog input object to record voltage from the strain gage and set up the parameters for acquisition.

```
% Create the object.
d = daq('ni');
% Add one channel for recording the strain.
ai = addinput(d,'Dev1','ai0','Voltage');
% Set the sample rate to 10,000 Hz.
d.Rate = 10000;
```

Next, configure the DataAcquisition object for hardware-triggered acquisition using the RTSI1 terminal as the external trigger source.

```
addtrigger(d,'Digital','StartTrigger','External','Dev1/RTSI1');
d.DigitalTriggers(1).Condition = 'RisingEdge';
```

Configure the Image Acquisition Board

Using the Image Acquisition Toolbox, create the video input object to record video and set up the parameters for acquisition and for driving RTSI1 high when the acquisition starts.

```
% Create the object.
vid = videoinput('ni', 2);
% Set to acquire approximately one second of frames per trigger.
vid.FramesPerTrigger = 30;
```

In order to drive the data acquisition card's RTSI line, you need to set the correct line and polarity on the frame grabber. In addition, you need to determine what frame grabber event will drive the RTSI line. You can see a list of events that are available by looking at the device-specific source properties that end in “DriveLine” and “DrivePolarity”:

```
% Get the currently selected source.
src = getselectedsource(vid);
% Display the properties and their possible settings.
set(src)
```

```
  General Settings:
    Tag

  Device Specific Properties:
    AcquisitionDoneDriveLine: [ {none} | external0 | rtsi0 | rtsi1 | rtsi2 | rtsi3 | rtsi4 | rts
    AcquisitionDoneDrivePolarity: [ {activeHigh} | activeLow ]
    AcquisitionInProgressDriveLine: [ {none} | external0 | rtsi0 | rtsi1 | rtsi2 | rtsi3 | rtsi4
    AcquisitionInProgressDrivePolarity: [ {activeHigh} | activeLow ]
    ExternalTriggerLineFilter: [ off | {on} ]
    FrameDoneDriveLine: [ {none} | external0 | rtsi0 | rtsi1 | rtsi2 | rtsi3 | rtsi4 | rtsi5 | r
    FrameDoneDrivePolarity: [ {activeHigh} | activeLow ]
    FrameStartDriveLine: [ {none} | external0 | rtsi0 | rtsi1 | rtsi2 | rtsi3 | rtsi4 | rtsi5 | 
    FrameStartDrivePolarity: [ {activeHigh} | activeLow ]
    HSyncDriveLine: [ {none} | external0 | rtsi0 | rtsi1 | rtsi2 | rtsi3 | rtsi4 | rtsi5 | rtsi6
    HSyncDrivePolarity: [ {activeHigh} | activeLow ]
    RTSITriggerLineFilter: [ off | {on} ]
    VSyncDriveLine: [ {none} | external0 | rtsi0 | rtsi1 | rtsi2 | rtsi3 | rtsi4 | rtsi5 | rtsi6
    VSyncDrivePolarity: [ {activeHigh} | activeLow ]
```

In this case, you want to drive RTSI line 1 high when the acquisition is in progress. This ensures that the line is driven high as soon as the acquisition begins. To do this, you need to set the acquisition in progress drive line to 'rtsi1':

```
% Set to drive RTSI1 high when the acquisition begins.
src.AcquisitionInProgressDriveLine = 'rtsi1';
```

Looking at the output above, you can see that the polarity for the acquisition in progress event is already set to 'activeHigh', so you do not need to set it.

Note that the maximum number of lines that you can drive is hardware dependent and will possibly vary between devices.

At this point you are set to acquire approximately one second of data from each device when the image acquisition device is started.

Start the Acquisition

You can now start the analog input object, which acquires one second of data by default. See that it is waiting for a hardware trigger.

```
start(d)
d.WaitingForDigitalTrigger
```

```
ans =

  logical

   1
```

You can now display a summary of the video input object and see that it is set up to trigger immediately upon start.

```
vid
```

```
Summary of Video Input Object Using 'PCIe-1430'.

   Acquisition Source(s):  Channel 0 is available.

  Acquisition Parameters:  'Channel 0' is the current selected source.
                           30 frames per trigger using the selected source.
                           'img0_Port0' video data to be logged upon START.
                           Grabbing first of every 1 frame(s).
                           Log data to 'memory' on trigger.

      Trigger Parameters:  1 'immediate' trigger(s) on START.

                  Status:  Waiting for START.
                           0 frames acquired since starting.
                           0 frames available for GETDATA.
```

When you start the video input object, it will immediately be triggered and begin acquiring. At that moment, the frame grabber will send a signal to the data acquisition card across RTSI line 1, which will cause the data acquisition to begin nearly synchronously.

```
start(vid)
% Wait on both objects until you are done acquiring.
wait(vid), wait(d,2)
```

Display a Summary of Acquisitions

If you now display a summary you will see that both devices have acquired data.

```
d.NumScansAcquired
```

```
ans =

    10000
```

and:

```
vid
```

```
Summary of Video Input Object Using 'PCIe-1430'.

   Acquisition Source(s):  Channel 0 is available.

  Acquisition Parameters:  'Channel 0' is the current selected source.
                           30 frames per trigger using the selected source.
                           'img0_Port0' video data to be logged upon START.
                           Grabbing first of every 1 frame(s).
                           Log data to 'memory' on trigger.

      Trigger Parameters:  1 'immediate' trigger(s) on START.

                  Status:  Waiting for START.
                           30 frames acquired since starting.
                           30 frames available for GETDATA.
```

Clean Up the Objects

Once the video input and analog input objects are no longer needed, delete them and clear them and the reference to the source from the workspace.

```
delete(vid)
clear vid d src
```

Using the Kinect for Windows V1 from Image Acquisition Toolbox

This example shows how to obtain the data available from Kinect® for Windows® V1 sensor using Image Acquisition Toolbox™:

Utility Functions

In order the keep this example as simple as possible, some utility functions for working with the Kinect for Windows metadata have been created. These utility functions include the skeletalViewer function which accepts the skeleton data, color image and number of skeletons as inputs and displays the skeleton overlaid on the color image

See What Kinect for Windows Devices and Formats are Available

The Kinect for Windows has two sensors, an color sensor and a depth sensor. To enable independent acquisition from each of these devices, they are treated as two independent devices in the Image Acquisition Toolbox. This means that separate VIDEOINPUT object needs to be created for each of the color and depth(IR) devices.

```
% The Kinect for Windows Sensor shows up as two separate devices in IMAQHWINFO.
hwInfo = imaqhwinfo('kinect')
```

```
hwInfo = 

       AdaptorDllName: [1x68 char]
    AdaptorDllVersion: '4.5 (R2013a Prerelease)'
          AdaptorName: 'kinect'
            DeviceIDs: {[1]  [2]}
           DeviceInfo: [1x2 struct]
```

```
hwInfo.DeviceInfo(1)
```

```
ans = 

             DefaultFormat: 'RGB_640x480'
       DeviceFileSupported: 0
                DeviceName: 'Kinect Color Sensor'
                  DeviceID: 1
     VideoInputConstructor: 'videoinput('kinect', 1)'
    VideoDeviceConstructor: 'imaq.VideoDevice('kinect', 1)'
          SupportedFormats: {1x7 cell}
```

```
hwInfo.DeviceInfo(2)
```

```
ans = 

             DefaultFormat: 'Depth_640x480'
       DeviceFileSupported: 0
                DeviceName: 'Kinect Depth Sensor'
                  DeviceID: 2
```

```
     VideoInputConstructor: 'videoinput('kinect', 2)'
    VideoDeviceConstructor: 'imaq.VideoDevice('kinect', 2)'
          SupportedFormats: {'Depth_320x240'  'Depth_640x480'  'Depth_80x60'}
```

Acquire Color and Depth Data

In order to acquire synchronized color and depth data, we must use manual triggering instead of immediate triggering. The default immediate triggering suffers from a lag between streams while performing synchronized acquisition. This is due to the overhead in starting of streams sequentially.

```
% Create the VIDEOINPUT objects for the two streams
colorVid = videoinput('kinect',1)
```

```
Summary of Video Input Object Using 'Kinect Color Sensor'.

   Acquisition Source(s):  Color Source is available.

  Acquisition Parameters:  'Color Source' is the current selected source.
                           10 frames per trigger using the selected source.
                           'RGB_640x480' video data to be logged upon START.
                           Grabbing first of every 1 frame(s).
                           Log data to 'memory' on trigger.

      Trigger Parameters:  1 'immediate' trigger(s) on START.

                  Status:  Waiting for START.
                           0 frames acquired since starting.
                           0 frames available for GETDATA.
```

```
depthVid = videoinput('kinect',2)
```

```
Summary of Video Input Object Using 'Kinect Depth Sensor'.

   Acquisition Source(s):  Depth Source is available.

  Acquisition Parameters:  'Depth Source' is the current selected source.
                           10 frames per trigger using the selected source.
                           'Depth_640x480' video data to be logged upon START.
                           Grabbing first of every 1 frame(s).
                           Log data to 'memory' on trigger.

      Trigger Parameters:  1 'immediate' trigger(s) on START.

                  Status:  Waiting for START.
                           0 frames acquired since starting.
                           0 frames available for GETDATA.
```

```
% Set the triggering mode to 'manual'
triggerconfig([colorVid depthVid],'manual');
```

Set the FramesPerTrigger property of the VIDEOINPUT objects to '100' to acquire 100 frames per trigger. In this example 100 frames are acquired to give the Kinect for Windows sensor sufficient time to start tracking a skeleton.

```
colorVid.FramesPerTrigger = 100;
depthVid.FramesPerTrigger = 100;

% Start the color and depth device. This begins acquisition, but does not
% start logging of acquired data.
start([colorVid depthVid]);

% Trigger the devices to start logging of data.
trigger([colorVid depthVid]);

% Retrieve the acquired data
[colorFrameData,colorTimeData,colorMetaData] = getdata(colorVid);
[depthFrameData,depthTimeData,depthMetaData] = getdata(depthVid);

% Stop the devices
stop([colorVid depthVid]);
```

Configure Skeletal Tracking

The Kinect for Windows sensor provides different modes to track skeletons. These modes can be accessed and configured from the VIDEOSOURCE object of the depth device. Let's see how to enable skeleton tracking.

```
% Get the VIDEOSOURCE object from the depth device's VIDEOINPUT object.
depthSrc = getselectedsource(depthVid)
```

```
   Display Summary for Video Source Object:

      General Settings:
        Parent = [1x1 videoinput]
        Selected = on
        SourceName = Depth Source
        Tag = 
        Type = videosource

      Device Specific Properties:
        Accelerometer = [-0.008547 -0.98046 -0.11966]
        BodyPosture = Standing
        CameraElevationAngle = 9
        DepthMode = Default
        FrameRate = 30
        IREmitter = on
        SkeletonsToTrack = [1x0 double]
        TrackingMode = Off
```

The properties on the depth source object that control the skeletal tracking features are TrackingMode, SkeletonToTrack and BodyPosture properties on the VIDEOSOURCE.

TrackingMode controls whether or not skeletal tracking is enabled and, if it is enabled, whether all joints are tracked, ‘Skeleton’, or if just the hip position is tracked, ‘Position’. Setting TrackingMode to ‘off’ (default) disables all tracking and reduces the CPU load.

The ‘BodyPosture’ property determines how many joints are tracked. If ‘BodyPosture’ is set to ‘Standing’, twenty joints are tracked. If it is set to ‘Seated’, then ten joints are tracked.

The SkeletonToTrack property can be used to selectively track one or two skeletons using the 'SkeletonTrackingID'. The currently valid values for 'SkeletonTrackingID' are returned as a part of the metadata of the depth device.

```
% Turn on skeletal tracking.
depthSrc.TrackingMode = 'Skeleton';
```

Access Skeletal Data

The skeleton data that the Kinect for Windows produces is accessible from the depth device as a part of the metadata returned by GETDATA. The Kinect for Windows can track the position of up to six people in view and can actively track the joint locations of two of the six skeletons. It also supports two modes of tracking people based on whether they are standing or seated. In standing mode, the full 20 joint locations are tracked and returned; in seated mode the 10 upper body joints are returned. For more details on skeletal data, see the MATLAB documentation on Kinect for Windows adaptor.

```
% Acquire 100 frames with tracking turned on.
% Remember to have a person in person in front of the
% Kinect for Windows to see valid tracking data.
colorVid.FramesPerTrigger = 100;
depthVid.FramesPerTrigger = 100;

start([colorVid depthVid]);
trigger([colorVid depthVid]);

% Retrieve the frames and check if any Skeletons are tracked
[frameDataColor] = getdata(colorVid);
[frameDataDepth, timeDataDepth, metaDataDepth] = getdata(depthVid);

% View skeletal data from depth metadata
metaDataDepth
```

```
metaDataDepth =

100x1 struct array with fields:

    AbsTime
    FrameNumber
    IsPositionTracked
    IsSkeletonTracked
    JointDepthIndices
    JointImageIndices
    JointTrackingState
    JointWorldCoordinates
    PositionDepthIndices
    PositionImageIndices
    PositionWorldCoordinates
    RelativeFrame
    SegmentationData
    SkeletonTrackingID
    TriggerIndex
```

We randomly choose the 95th frame to visualize the image and skeleton data.

```
% Check for tracked skeletons from depth metadata
anyPositionsTracked = any(metaDataDepth(95).IsPositionTracked ~= 0)
anySkeletonsTracked = any(metaDataDepth(95).IsSkeletonTracked ~= 0)
```

```
anyPositionsTracked =

     1


anySkeletonsTracked =

     1
```

The results above show that at least one skeleton is being tracked. If tracking is enabled but no IDs are specified with the TrackingID property, the Kinect for Windows software automatically chooses up to two skeletons to track. Use the IsSkeletonTracked metadata to determine which skeletons are being tracked.

```
% See which skeletons were tracked.
trackedSkeletons = find(metaDataDepth(95).IsSkeletonTracked)
```

```
trackedSkeletons =

     1
```

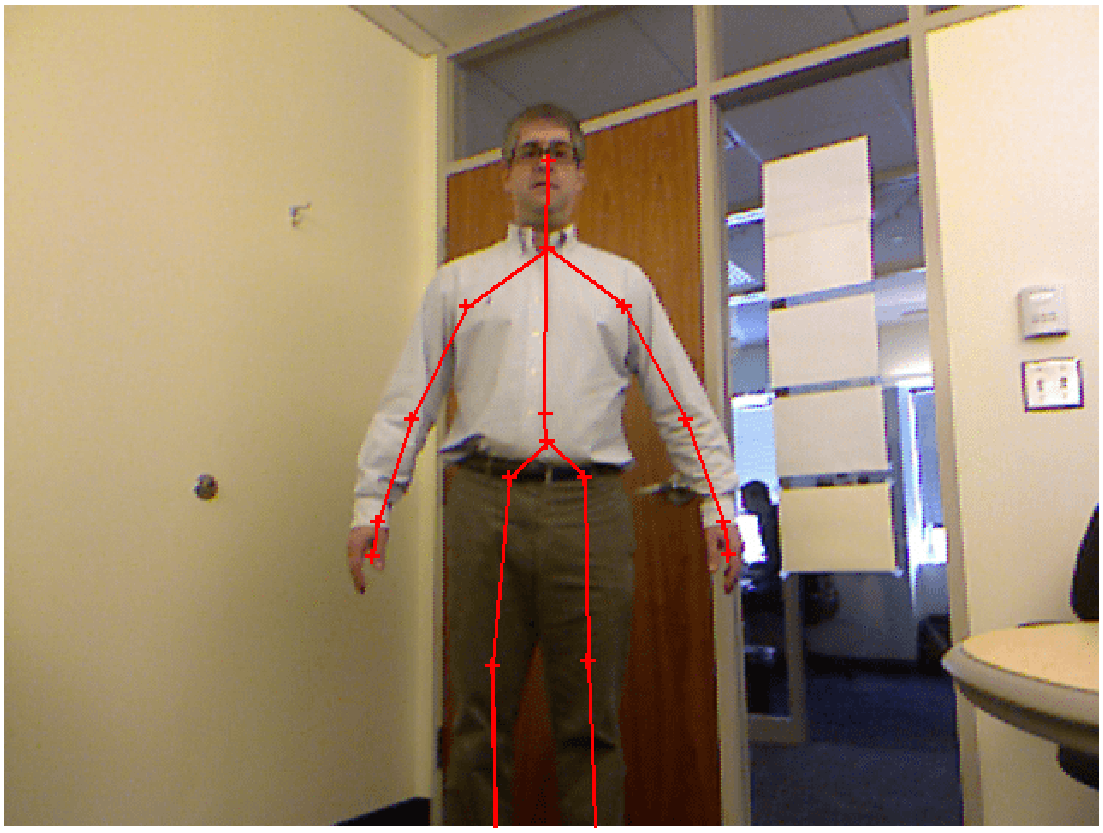
Display skeleton's joint coordinates. Note that if the 'BodyPosture' property is set to 'Seated', the 'JointCoordinates' and 'JointIndices' will still have a length of 20, but indices 2-11(upper-body joints) alone will be populated.

```
jointCoordinates = metaDataDepth(95).JointWorldCoordinates(:, :, trackedSkeletons)
% Skeleton's joint indices with respect to the color image
jointIndices = metaDataDepth(95).JointImageIndices(:, :, trackedSkeletons)
```

```
jointCoordinates =

   -0.0119   -0.0072    1.9716
   -0.0107    0.0545    2.0376
   -0.0051    0.4413    2.0680
    0.0033    0.6430    2.0740
   -0.1886    0.3048    2.0469
   -0.3130    0.0472    2.0188
   -0.3816   -0.1768    1.9277
   -0.3855   -0.2448    1.8972
    0.1724    0.3022    2.0449
    0.3102    0.0382    2.0304
    0.3740   -0.1929    1.9591
    0.3786   -0.2625    1.9356
   -0.0942   -0.0850    1.9540
   -0.1367   -0.4957    1.9361
   -0.1356   -0.8765    1.9339
   -0.1359   -0.9284    1.8341
    0.0683   -0.0871    1.9504
    0.0706   -0.4822    1.9293
    0.0858   -0.8804    1.9264
    0.0885   -0.9321    1.8266


jointIndices =

   318   256
   317   240
   318   143
   319    92
   271   177
   239   243
   219   303
   216   323
   363   177
   399   243
   421   303
   424   322
   296   277
   286   387
   288   492
   286   520
   340   277
   342   384
   347   493
   350   522
```

Draw the Skeleton Over the Corresponding Color Image

```
% Pull out the 95th color frame
image = frameDataColor(:, :, :, 95);

% Find number of Skeletons tracked
nSkeleton = length(trackedSkeletons);

% Plot the skeleton
util_skeletonViewer(jointIndices, image, nSkeleton);
```



Creating Time-Lapse Video Using a Noncontiguous Acquisition

This example shows how to create a time-lapse video without using all the frames of the acquisition.

The Image Acquisition Toolbox™ makes it easy to produce time-lapse video. The most efficient way to do time-lapse acquisition is to use the Image Acquisition Toolbox's built-in ability to log frames directly to an AVI file, and its ability to perform time decimation by retaining only a fraction of all the frames acquired by the camera.

Watch a day long time-lapse sequence. (21 seconds)

Create a Video Input Object

Before acquiring images using the Image Acquisition Toolbox, create a video input object.

```
% When executing the following code, you may need to
% modify it to match your acquisition hardware.
vid = videoinput('winvideo',1,'RGB24_352x288');
```

Determine the Frame Rate

Most devices do not allow you to control their frame rate. It is best to determine the frame rate experimentally by acquiring frames and analyzing the time stamps of the frames.

```
vid.FramesPerTrigger = 100;
start(vid);
wait(vid,Inf);

% Retrieve the frames and timestamps for each frame.
numframes = vid.FramesAvailable;
[frames,time] = getdata(vid,numframes);

% Calculate the frame rate by taking the average difference
% between the timestamps for each frame.
framerate = mean(1./diff(time))
```

```
framerate =

   17.5296
```

Specify the Noncontiguous Acquisition

The `FrameGrabInterval` property specifies how often frames are stored from the video stream. For instance, if we set it to 5, then only 1 in 5 frames is kept -- the other 4 frames will be discarded.

```
% We want to compress 30 seconds into 3 seconds, so
% only acquire every tenth frame.
vid.FrameGrabInterval = 10;
```

Determine the Number of Frames to Acquire

To determine how many frames to acquire in total, calculate the total number of frames that would be acquired at the device's frame rate, and then divide by the `FrameGrabInterval`.

```
capturetime = 30;
interval = vid.FrameGrabInterval;
numframes = floor(capturetime * framerate / interval)
```

```
numframes =

    52
```

```
vid.FramesPerTrigger = numframes;
```

Configure AVI Disk Logging

Due to the large number of frames that will be acquired, it is more practical to log the images to disk rather than to memory. Using the Image Acquisition Toolbox, you can log images directly to an AVI file. We configure this using the `LoggingMode` property.

```
vid.LoggingMode = 'disk';
```

Create a VideoWriter object to log to, using the `VideoWriter` command. We must specify the filename to use, and then set frame rate that the AVI file should be played back at. Then, set the `DiskLogger` property of the video input object to the VideoWriter object.

```
vwObj = VideoWriter('timelapsevideo','Uncompressed AVI');
vwObj.FrameRate = framerate;
vid.DiskLogger = vwObj;
vid
```

```
Summary of Video Input Object Using 'Logitech QuickCam Fusion'.

   Acquisition Source(s):  input1 is available.

  Acquisition Parameters:  'input1' is the current selected source.
                           91 frames per trigger using the selected source.
                           'RGB24_352x288' video data to be logged upon START.
                           Grabbing first of every 10 frame(s).
                           Log data to 'disk' on trigger.

      Trigger Parameters:  1 'immediate' trigger(s) on START.

                  Status:  Waiting for START.
                           100 frames acquired since starting.
                           0 frames available for GETDATA.
```

Perform the Time-Lapse Acquisition

Start the time-lapse acquisition, and wait for the acquisition to complete. Note that the Image Acquisition Toolbox does not tie up MATLAB® while it is acquiring. You can start the acquisition and keep working in MATLAB.

```
start(vid);

% Wait for the capture to complete before continuing.
wait(vid,Inf);
```

Close the AVI File

Once the capture is completed, retrieve the AVI file object, and use the `close` function to release the resources associated with it.

```
vwObj = vid.DiskLogger;
close(vwObj);
```

Play Back the Time-Lapse AVI Sequence

To play back the time-lapse AVI sequence, right-click on the filename in the MATLAB Current Folder browser, and choose Open Outside MATLAB from the context menu.

Clean Up

When you are done with the video input object, you should use the `delete` function to free the hardware resources associated with it, and remove it from the workspace using the `clear` function.

```
delete(vid);
clear vid;
```

Creating Time-Lapse Video Using Timer Events

This example shows how to create a time-lapse video using timer events to prequalify frames.

The Image Acquisition Toolbox™ makes it easy to produce time-lapse video. In this example, we will use timer events to acquire frames to an AVI file. This technique of time decimation has the advantage that it allows you to make a decision about each frame before storing it. An application of this would be to only store frames that meet certain illumination levels, have motion relative to the previous frame, etc.

Watch a day long time-lapse sequence. (21 seconds)

Create a Video Input Object

Before acquiring images using the Image Acquisition Toolbox, create a video input object.

```
% When executing the following code, you may need to
% modify it to match your acquisition hardware.
vid = videoinput('winvideo',1,'RGB24_352x288');
```

Configure the Timer

To generate timer events, we specify two things: what happens when it occurs, and how often it should occur. The `TimerFcn` property specifies the callback function to execute when a timer event occurs. A timer event occurs when the time period specified by the `TimerPeriod` property expires.

The callback function is responsible for triggering the acquisition and storing the frame into the AVI file. More details on how to use this callback can be found in the documentation. This callback function itself is defined in the file timelapse_timer.m.

The configuration we will use is

- timelapse_timer will execute each time the timer elapses
- The timer function will execute every one second

```
set(vid,'TimerPeriod',1);
vid.TimerFcn = @timelapse_timer;
```

Store the VideoWriter Object

Store the VideoWriter object in the `UserData` property of the video input object so that it will be available inside the callback.

```
vwObj = VideoWriter('timelapsevideo', 'Uncompressed AVI');
vwObj.FrameRate = 15;
open(vwObj);
```

Configure the Video Input Object to Use Manual Triggering

Each time the timer event occurs

- Manually trigger the acquisition using the `triggerconfig` command
- Acquire one frame
- Acquire 9 additional triggers worth of data, for a total of 10 frames

```
triggerconfig(vid, 'manual');
vid.FramesPerTrigger = 1;
vid.TriggerRepeat = 9;
vid
```

```
Summary of Video Input Object Using 'Logitech QuickCam Fusion'.

   Acquisition Source(s):  input1 is available.

  Acquisition Parameters:  'input1' is the current selected source.
                           1 frames per trigger using the selected source.
                           'RGB24_352x288' video data to be logged upon START.
                           Grabbing first of every 1 frame(s).
                           Log data to 'memory' on trigger.

      Trigger Parameters:  10 'manual' trigger(s) upon TRIGGER.

                  Status:  Waiting for START.
                           0 frames acquired since starting.
                           0 frames available for GETDATA.
```

Perform the Time-Lapse Acquisition

Now, start the time lapse acquisition, and wait for up to 20 seconds for the acquisition to complete.

```
start(vid);
wait(vid,20);
```

Close the AVI File

Once the capture is completed, retrieve the VideoWriter object stored in the UserData property, and use the `close` function to release the resources associated with it.

```
avi = vid.UserData;
avi = close(avi);
```

Play Back the Time-Lapse AVI Sequence

To play back the time-lapse AVI sequence, right-click on the filename in the MATLAB® Current Folder browser, and choose Open Outside MATLAB from the context menu.

Clean Up

When you are done with the video input object, you should use the `delete` function to free the hardware resources associated with it, and remove it from the workspace using the `clear` function.

```
delete(vid);
clear vid;
```

The Timer Callback Function

The following is a description of the callback function executed for each timer event

- Trigger the toolbox to acquire a single frame
- Retrieve the frame

- Determine whether to keep the frame
- Use the `writeVideo` function to add the frame to the AVI file

The VideoWriter object is stored in the `UserData` property of the object.

```
type timelapse_timer
```

```
function timelapse_timer(vid,~)
% This callback function triggers the acquisition and saves frames to an AVI file.

% trigger the acquisition and get the frame
trigger(vid);
frame = getdata(vid,1);

% Retrieve the total number of frames acquired
numframes_acquired = vid.FramesAcquired;

% Drop every other frame:  If the frame is odd,
% keep it.  If it is an even frame, do not keep it.
keepframe = (mod(numframes_acquired,2) == 1);

% Insert your processing code here

if(~keepframe)
    return;
end

% Retrieve the VideoWriter object stored in the UserData
% property.
vwObj = vid.UserData;

% Add the frame to the AVI
writeVideo(vwObj, frame);

end
```

Creating Time-Lapse Video Using Postprocessed Data

This example shows how to create a time-lapse video by deleting unnecessary frames during postprocessing.

The Image Acquisition Toolbox™ makes it easy to produce time-lapse photography. In this example, we will use postprocessing to delete frames from an acquired sequence of images. This is ideal for situations where you are not sure which frames are relevant during capture, or where your processing would take too long to occur during the acquisition. A possible application would be to delete frames that have no motion relative to the previous frame. The primary disadvantage of this method of time decimation is that it requires large amounts of memory to store all the frames. This example acquires to memory, but you would likely acquire to an AVI file, and then use the `VideoReader` command to postprocess the frames.

Watch a day long time-lapse sequence.

Create a Video Input Object

Before acquiring images using the Image Acquisition Toolbox, create a video input object.

```
% When executing the following code, you may need to
% modify it to match your acquisition hardware.
vid = videoinput('winvideo',1,'RGB24_352x288');
```

Configure the Video Input Object

The configuration we will use is

- Acquire 100 frames

```
vid.FramesPerTrigger = 100;
vid
```

```
Summary of Video Input Object Using 'Logitech QuickCam Fusion'.

   Acquisition Source(s):  input1 is available.

  Acquisition Parameters:  'input1' is the current selected source.
                           100 frames per trigger using the selected source.
                           'RGB24_352x288' video data to be logged upon START.
                           Grabbing first of every 1 frame(s).
                           Log data to 'memory' on trigger.

      Trigger Parameters:  1 'immediate' trigger(s) on START.

                  Status:  Waiting for START.
                           0 frames acquired since starting.
                           0 frames available for GETDATA.
```

Acquire and Retrieve the Frames

```
start(vid);
wait(vid);
framesavailable = vid.FramesAvailable;
framesavailable =

   100

```

```
frames = getdata(vid,framesavailable);
```

Postprocess the Acquired Frames

Here, simply remove every other frame. However, you could do processing that is much more complex.

```
toberemoved_index = [2:2:framesavailable];
frames(:,:,:,toberemoved_index) = [];
numframes = size(frames,4)
```

```
numframes =

    50

```

Render the Frames to an AVI File

Render the frames to an AVI file. To do this, create a `VideoWriter` object, call the `writeVideo` function to add all the frames to the AVI file, and then use the `close` function to release the resources associated with the AVI file.

```
vwObj = VideoWriter('timelapsevideo', 'Uncompressed AVI');
vwObj.FrameRate = 15;
open(vwObj);
writeVideo(vwObj, frames);
close(vwObj);
```

Play Back the Time-Lapse AVI Sequence

To play back the time-lapse AVI sequence, right-click on the filename in the MATLAB® Current Folder browser, and choose Open Outside MATLAB from the context menu.

Clean Up

When you are done with the video input object, you should use the `delete` function to free the hardware resources associated with it, and remove it from the workspace using the `clear` function. Also delete and clear the VideoWriter object.

```
delete(vid);
delete(vwObj);
clear vid vwObj;
```

Barcode Recognition Using Live Video Acquisition

This example shows how to use the From Video Device block to recognize a barcode.

Image Acquisition Toolbox™ provides a Simulink® block to acquire live image data from image acquisition devices into Simulink models.

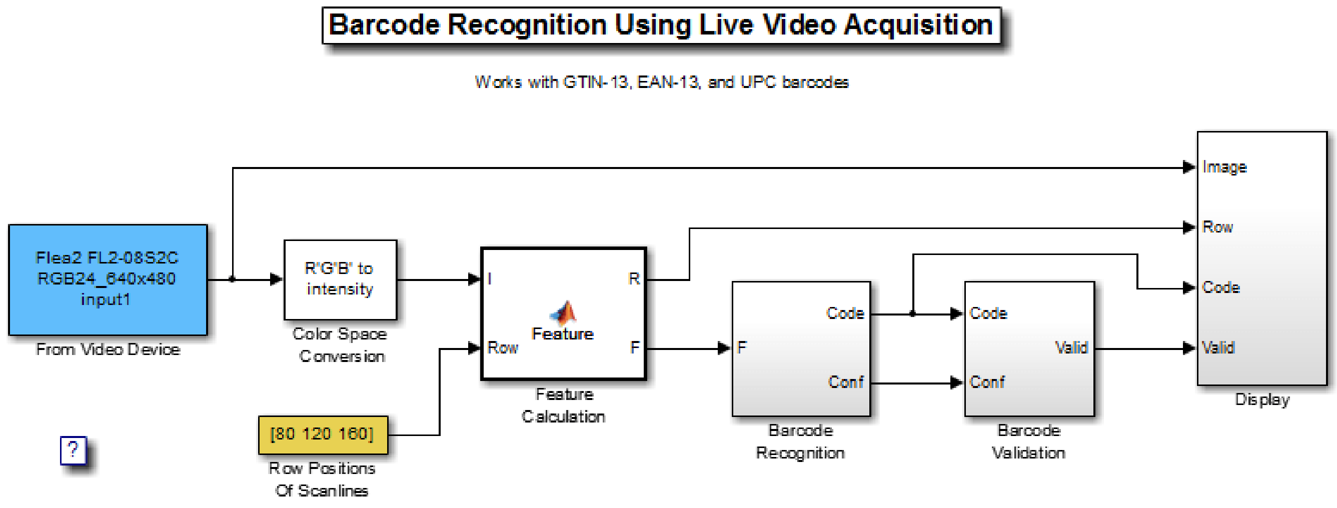
This example uses the From Video Device block to acquire live image data from the Point Grey Flea® 2 camera into Simulink. The example uses the Computer Vision Toolbox™ to create an image processing system which can recognize and interpret a GTIN-13 barcode. The GTIN-13 barcode, formally known as EAN-13, is an international barcode standard. It is a superset of the widely used UPC standard.

This example requires Simulink, Computer Vision Toolbox and the Point Grey Flea® 2 camera to open and run the model.

Watch barcode recognition on live video stream. (11 seconds)

Example Model

The following figure shows the example model using the From Video Device block.



Live Video Input

The input video is acquired live from a DCAM image acquisition device (Point Grey Flea® 2). In this example, the block acquires RGB frames from the camera and outputs them into the Simulink model at every simulation time step.

Algorithm

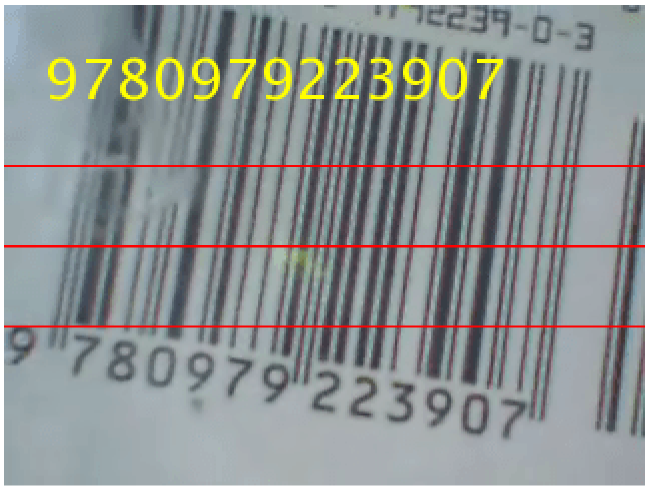
The barcode recognition example performs a search on some selected rows of the input image, called scan lines. The scan lines are analyzed per pixel and marked by feature. Once all pixels are marked with a feature value, the sequences of patterns are analyzed. The example identifies the guard patterns and symbols by sequence and location. The symbols are up sampled and compared with the codebook to determine a corresponding code.

To compensate for various barcode orientations, the example analyzes from left to right and from right to left and chooses the better match. If the check sum is correct and a matching score against the codebook is higher than a set threshold, the code is considered valid and is displayed.

You can change the number and location of the scan lines by changing the value of the "Row Positions Of Scan Lines" parameter.

Results

The scan lines that have been used to detect barcodes are displayed in red. When a GTIN-13 is correctly recognized and verified, the code is displayed in yellow.



Even though a Flea® 2 camera was used for this example, this model can be easily updated to connect your models to other supported image acquisition devices. This provides you the flexibility to use the same Simulink model with different image acquisition hardware.

Live Image Acquisition and Histogram Display

This example shows how to use Simulink® blocks to display live video stream and a histogram of its RGB values side by side.

Image Acquisition Toolbox™ provides a Simulink block to acquire live image data from image acquisition devices into Simulink models. This example uses the From Video Device block to acquire live image data from the Logitech® Quickcam® Zoom webcam into Simulink. This example uses the Histogram block in DSP System Toolbox™ to calculate the histograms of R, G, and B values in each video frame.

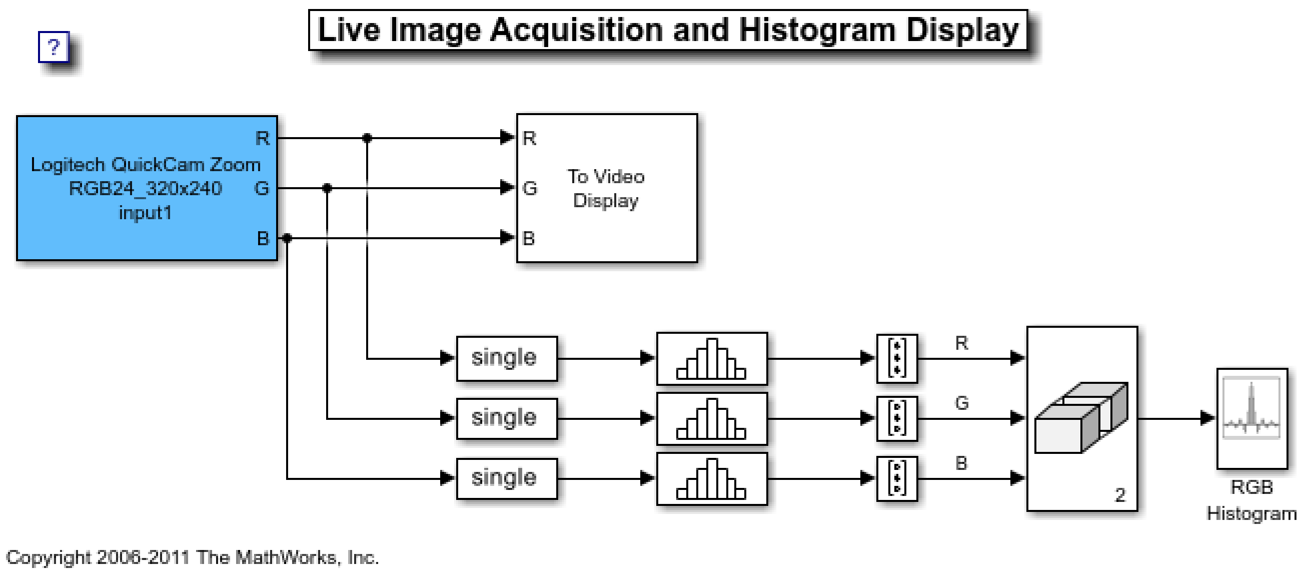
This example requires Simulink, DSP System Toolbox and Computer Vision Toolbox™ to open and run the model.

Watch histogram display on live video stream. (10 seconds)

Example Model

The following figure shows the example model using the From Video Device block.

```
open_system('demoimaqsl_rgbhistogram_win');
```



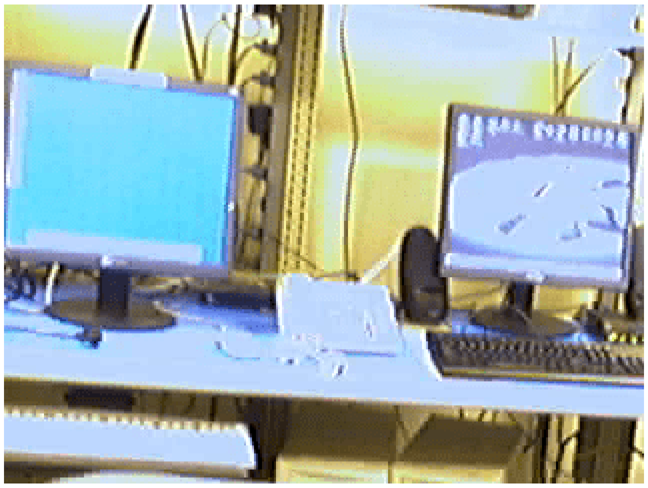
```
close_system('demoimaqsl_rgbhistogram_win');
```

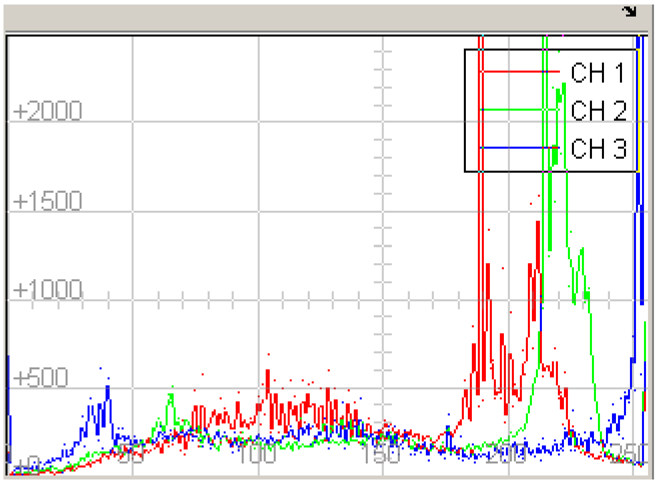
Live Video Input

The input video is acquired live from a winvideo image acquisition device (Logitech Quickcam Zoom). In this example, the block acquires RGB frames from the Logitech camera and outputs them into the Simulink model at every simulation time step. The data type output from the block is uint8.

Histogram Display Results

The example displays the histograms of R, G, and B values in the RGB Histogram window and displays the original RGB video in the viewer window.





Even though a winvideo Logitech webcam was used for this example, this model can be easily updated to connect your models to other supported image acquisition devices. This provides you the flexibility to use the same Simulink model with different image acquisition hardware.

Available Example Versions

Windows® only: demoimaqsl_rgbhistogram_win.slx

Platform independent: demoimaqsl_rgbhistogram_all.slx

Windows-only example model contains the To Video Display block (supported only on Windows) from Computer Vision Toolbox and supports code generation. The platform independent version consists of Video Viewer block and does not support code generation.

Edge Detection on Live Video Stream

This example shows how to use the From Video Device block to detect the edges of objects in a live video stream.

Image Acquisition Toolbox™ provides a Simulink® block to acquire live image data from image acquisition devices into Simulink models.

This example uses the From Video Device block to acquire live image data from a Hamamatsu C8484 camera into Simulink. The Prewitt method is applied to find the edges of objects in the input video stream.

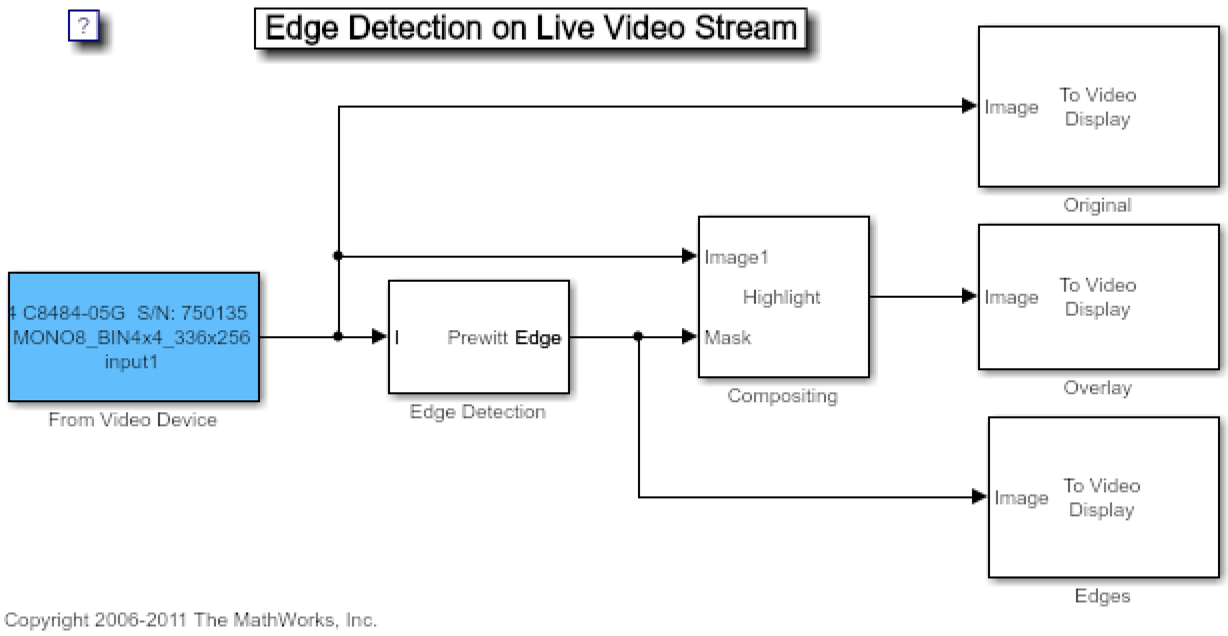
This example requires Simulink, and Computer Vision Toolbox™ to open and run the model.

Watch edge detection on live video. (9 seconds)

Example Model

The following figure shows the example model using the From Video Device block.

```
open_system('demoimaqsl_edgedetection_win');
```



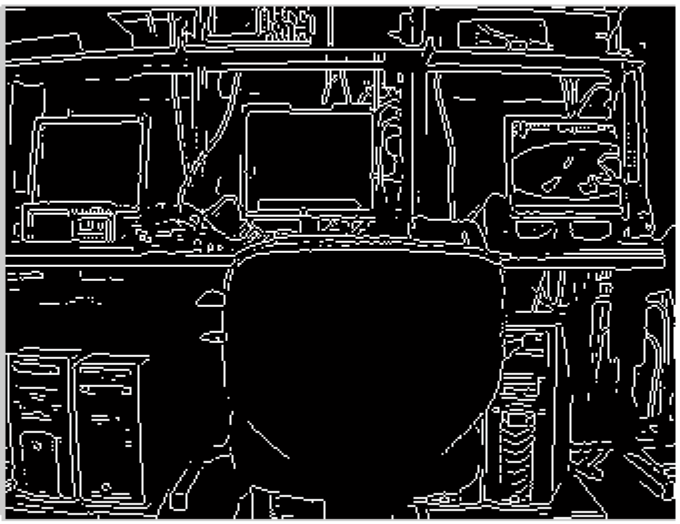
```
close_system('demoimaqsl_edgedetection_win');
```

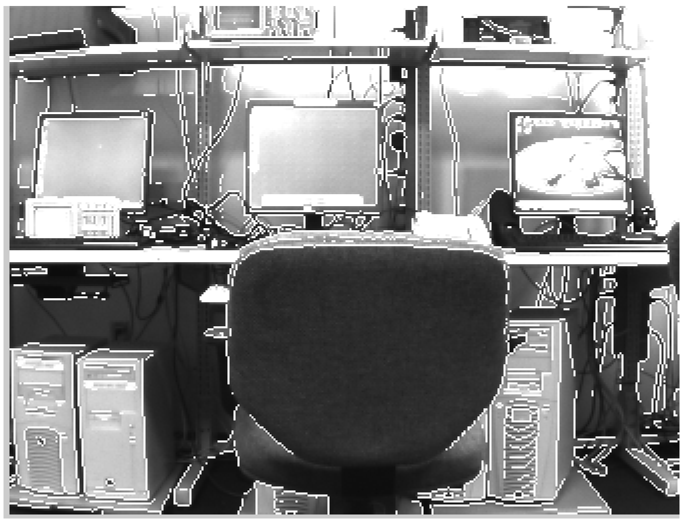
Live Video Input

The input video is acquired live from a Hamamatsu image acquisition device (C8484). In this example, the block acquires intensity data from the camera and outputs it into the Simulink model at every simulation time step. The data type output from the block is single.

Edge Detection Analysis

This example uses Computer Vision Toolbox to find the edges of objects in the video input. When you run the model, you can double-click the Edge Detection block and adjust the threshold parameter while the simulation is running. The higher you make the threshold, the smaller the amount of edges the example finds in the video stream.





Even though a Hamamatsu camera was used for this example, this model can be easily updated to connect your models to other supported image acquisition devices. This provides you the flexibility to use the same Simulink model with different image acquisition hardware.

Available Example Versions

Windows® only: demoimaqsl_edgedetection_win.slx

Platform independent: demoimaqsl_edgedetection_all.slx

Windows-only example model contains the To Video Display block (supported only on Windows) from Computer Vision Toolbox and supports code generation. The platform independent version consists of Video Viewer block and does not support code generation.

Acquire Images Using Parallel Workers

This example shows how to use the Parallel Computing Toolbox™ together with the Image Acquisiton Toolbox™ to acquire and save images in a separate MATLAB® worker.

Doing so allows you to perform other operations in the main MATLAB worker with minimal impact on the image acquisition. As a result, your image acquisition is more consistent and you can run more CPU-intensive operations in parallel on multicore CPUs.

Set Up Image Acquistion

This example uses the `parfeval` function from the Parallel Computing Toolbox to asynchronously execute a specified function. You can run the specified function in the background of your main MATLAB worker without waiting for it to complete. For more information about this function, see `parfeval` (Parallel Computing Toolbox).

Create a function called `captureVideo` to execute on the parallel MATLAB worker. This function creates a new `videoinput` object and sets the `FramesAcquiredFcn` property. It then configures the frames to acquire and starts the acquisition. See the `captureVideo` function at the end of this example. You can modify `captureVideo` to fit your image acquisition needs and setup.

The `captureVideo` function sets the `FramesAcquiredFcn` property to a handle to the `saveImages` callback function. The `saveImages` function saves acquired images to a folder in your current working directory. See the `saveImages` function at the end of this example.

Before you start the image acquisition, create a folder called `data`, where the parallel worker saves your acquired images.

```
mkdir("data\")
```

Start Image Acquisition on Parallel Worker

Create a parallel pool with one worker on your local machine using `parpool`.

```
parpool('local',1);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 1).
```

Run the `captureVideo` function in your parallel worker using `parfeval`. Since `captureVideo` does not have any output, specify the number of output arguments as `0`.

```
f = parfeval(@captureVideo,0)
```

```
f = 
 FevalFuture with properties:

                   ID: 4
             Function: @captureVideo
       CreateDateTime: 13-Jan-2021 11:58:35
        StartDateTime: 13-Jan-2021 11:58:35
     Running Duration: 0 days 0h 0m 0s
                State: running
                Error: none
```

The output shows that the status of the `parfeval` future is `running`.

If you want to block MATLAB until `parfeval` completes, you can use the `wait` function on the future `f`. Using the wait function is useful when subsequent code depends on the completion of `parfeval`.

```
wait(f);
```

Display the `parfeval` future to confirm that its status is `finished`.

```
disp(f);
```

```
 FevalFuture with properties:

                   ID: 4
             Function: @captureVideo
       CreateDateTime: 13-Jan-2021 11:58:35
        StartDateTime: 13-Jan-2021 11:58:35
     Running Duration: 0 days 0h 0m 15s
                State: finished (unread)
                Error: none
```

Your images are acquired and saved to the `data` folder in your current working directory.

Shut Down Parallel Pool

When you finish working with the parallel pool, clear the future and shut down the parallel pool.

```
clear f
delete(gcp("nocreate"))
```

Helper Functions

Function to Capture Video

The parallel worker executes the `captureVideo` function. This function creates a `videoinput` object and sets the appropriate properties. You can modify it to fit your image acquisition needs.

```
function captureVideo()
    % Create videoinput object.
    v = videoinput('winvideo');

    % Specify a custom callback to save images.
    v.FramesAcquiredFcn = @saveImages;

    % Specify the number of frames to acquire before calling the callback.
    v.FramesAcquiredFcnCount = 60;

    % Specify the total number of frames to acquire.
    v.FramesPerTrigger = 120;

    % Start recording.
    start(v);

    % Wait for the acquision to finish.
    wait(v);
end
```

Callback Function to Save Images

The `captureVideo` function sets the `saveImages` callback as the `FramesAcquiredFcn` for the `videoinput` object. This function reads the number of frames specified by the `FramesAcquiredFcnCount` from the buffer and saves them to the `data` folder. You can modify this callback to fit your needs.

```
function saveImages(src,obj)
    % Calculate the total frame number for each frame,
    % in order to save the files in order.
    currframes = src.FramesAcquired - src.FramesAcquiredFcnCount;

    % Read images from the videoinput buffer.
    imgs = getdata(src,src.FramesAvailable);

    % Save each image to a file in order.
    for i = 1:src.FramesAcquiredFcnCount
        imname = "data\img_" + (currframes + i) + ".TIFF";
        imwrite(imgs(:,:,:,i),imname);
    end
end
```

Detect Anomalies in Pills During Live Image Acquisition

This example demonstrates how to detect anomalies in pills during live image acquisition. This example uses the pretrained neural network from the “Detect Image Anomalies Using Explainable FCDD Network” (Computer Vision Toolbox) example for anomaly detection.

You must first calibrate and test the existing neural network. To obtain the data set for calibration and testing, capture images of both normal pills and pills with anomalies. This example shows how to capture these images with image acquisition hardware and use the captured images to calibrate and test the pretrained neural network.

Next, capture live images of pills and use the trained anomaly detector to classify the pill as normal or anomalous. The live image preview also displays a text label that indicates the probability that each pixel is normal or anomalous.

Lastly, you can deploy the trained anomaly detection model and live image acquisition using MATLAB® Compiler™. Once the script is deployed as a standalone executable, you can launch it on any machine without a MATLAB® license.

This example uses a webcam to capture the pill images for calibration and testing, as well as for live pill image classification. However, you can run this example using any image acquisition hardware that is supported by the Image Acquisition Toolbox™.

In addition to the required MATLAB toolboxes, you must also have the following Add-Ons installed to run this example.

- Image Acquisition Toolbox™ Support Package for OS Generic Video Interface
- Computer Vision Toolbox™ Automated Visual Inspection Library

Download Pretrained Network

This example uses a pretrained network that is trained to detect pill anomalies. Download the pretrained version of the pill anomaly detector by running the following commands.

```
trainedPillAnomalyDetectorNet_url = ...
    "https://ssd.mathworks.com/supportfiles/vision/data/trainedFCDDPillAnomalyDetectorSpkg.zip";
downloadTrainedNetwork(trainedPillAnomalyDetectorNet_url,pwd);
net = load(fullfile(pwd,"folderForSupportFilesInceptionModel","trainedPillFCDDNet.mat"));
detector = net.detector;
```

Prepare Data Set for Calibration and Testing

This example uses the downloaded pill anomaly detector since the images used to train it are similar to the images captured and classified in this example. Rather than retraining the existing model from scratch, this example shows how to capture images for calibration and testing. If your workflow requires a visual inspection model for a different application, refer to “Detect Image Anomalies Using Explainable FCDD Network” (Computer Vision Toolbox) for how to train the model from scratch.

Set Up Image Acquisition Hardware

Connect to your camera using the `videoinput` function and specify its properties. This example uses the OS generic video interface to acquire images from a Logitech® webcam with the `YUV2_352x288` video format.

```
vid = videoinput("winvideo","1","YUY2_352x288");
vid.ReturnedColorSpace = 'rgb';
vid.FramesPerTrigger = 70;
```

Get the source properties of the `videoinput` object. Specify device-specific properties to adjust for optimal image data collection. The values you specify for these properties depend on the camera hardware and the surrounding environment that images are being captured in.

```
src = getselectedsource(vid);
```

Acquire Data for Calibration and Testing

After setting up and configuring your image acquisition hardware, you can collect images to calibrate and test the anomaly detection model. Collect data in immediate acquisition mode. For more information about trigger types, see “Specifying the Trigger Type” on page 6-8.

```
start(vid);
wait(vid);
```

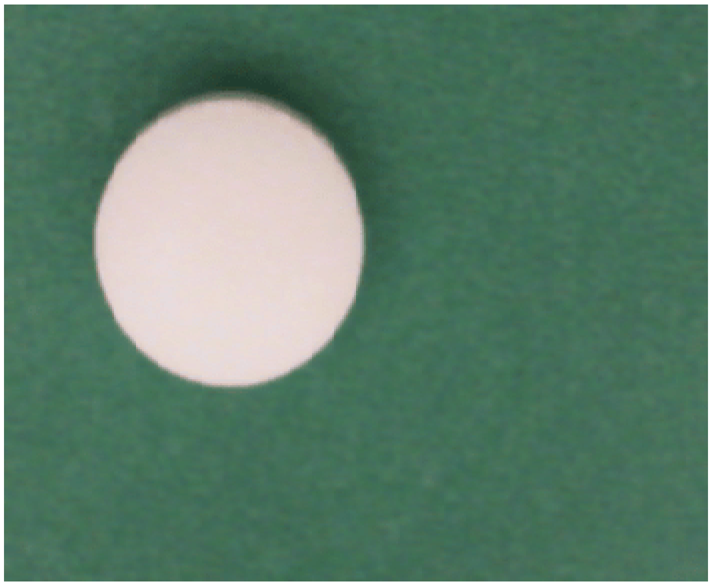
After you are finished capturing images, save the acquired image frames to MATLAB workspace and save the frames as TIF files using `imwrite`.

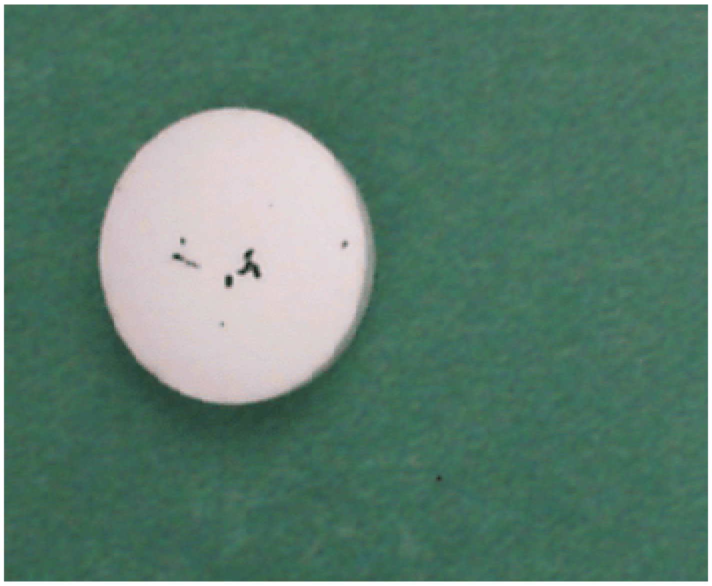
```
img = getdata(vid,70);

for f=1:vid.FramesAcquired
     imwrite(img(:,:,:,f), "image"+f+".tif");
end

delete(vid);
```

In the current working directory create a folder titled `ImageData`. Within this folder create two new folders titled `normal` and `dirt`. Move the normal and dirt image files into their respective folders.

The data set used in this example contains 40 normal images and 30 dirt images.

These two images show an example image from each class. A normal pill with no defects is on the left and a pill contaminated with dirt is on the right. The properties of these images align closely with the properties of the training data set from the “Detect Image Anomalies Using Explainable FCDD Network” (Computer Vision Toolbox) example. This example demonstrates how pre-trained models can be used after re-calibration and testing with a slightly modified data set.





Load and Preprocess Data

The `imageDataStore` object manages the collection of image files from both normal and dirt image classes.

```
imageDir = fullfile(pwd,"ImageData");
imds = imageDatastore(imageDir,IncludeSubfolders=true,LabelSource="foldernames");
```

The images for calibration are used to determine a threshold value for the classifier. The classifier labels images with anomaly scores above the threshold as anomalous. The remaining images are for testing and are used to evaluate the classifier.

```
normalTrainRatio  = 0;
anomalyTrainRatio = 0;
normalCalRatio  = 0.70;
anomalyCalRatio = 0.60;
```

```
normalTestRatio  = 1 - (normalTrainRatio + normalCalRatio);
anomalyTestRatio = 1 - (anomalyTrainRatio + anomalyCalRatio);

rng(0);
[imdsTrain,imdsCal,imdsTest] = splitAnomalyData(imds,["dirt"],...
    NormalLabelsRatio=[normalTrainRatio normalCalRatio normalTestRatio],...
    AnomalyLabelsRatio=[anomalyTrainRatio anomalyCalRatio anomalyTestRatio]);
```

```
Splitting anomaly dataset
-------------------------
* Finalizing... Done.
* Number of files and proportions per class in all the datasets:

                         Input                         Train                    Validation
              NumFiles          Ratio          NumFiles     Ratio     NumFiles           Ratio
              _____________________________    _________________    _____________________________

    dirt         30       0.428571428571429        0          0          18       0.39130434782608
    normal       40       0.571428571428571        0          0          28       0.60869565217391
```

Add binary labels to `normal` and `dirt` images in both calibration and test data sets by using the `transform` function with the operations specified by the `addLabelData` helper function.

```
dsCal = transform(imdsCal,@addLabelData,IncludeInfo=true);
dsTest = transform(imdsTest,@addLabelData,IncludeInfo=true);
```

Visualize a sample of calibration data that contains both normal and anomaly images.

```
numObs = length(imdsCal.Labels);
idx = randperm(numObs,9);
montage(imdsCal,Indices=idx)
```

Calibrate and Test Classification Model

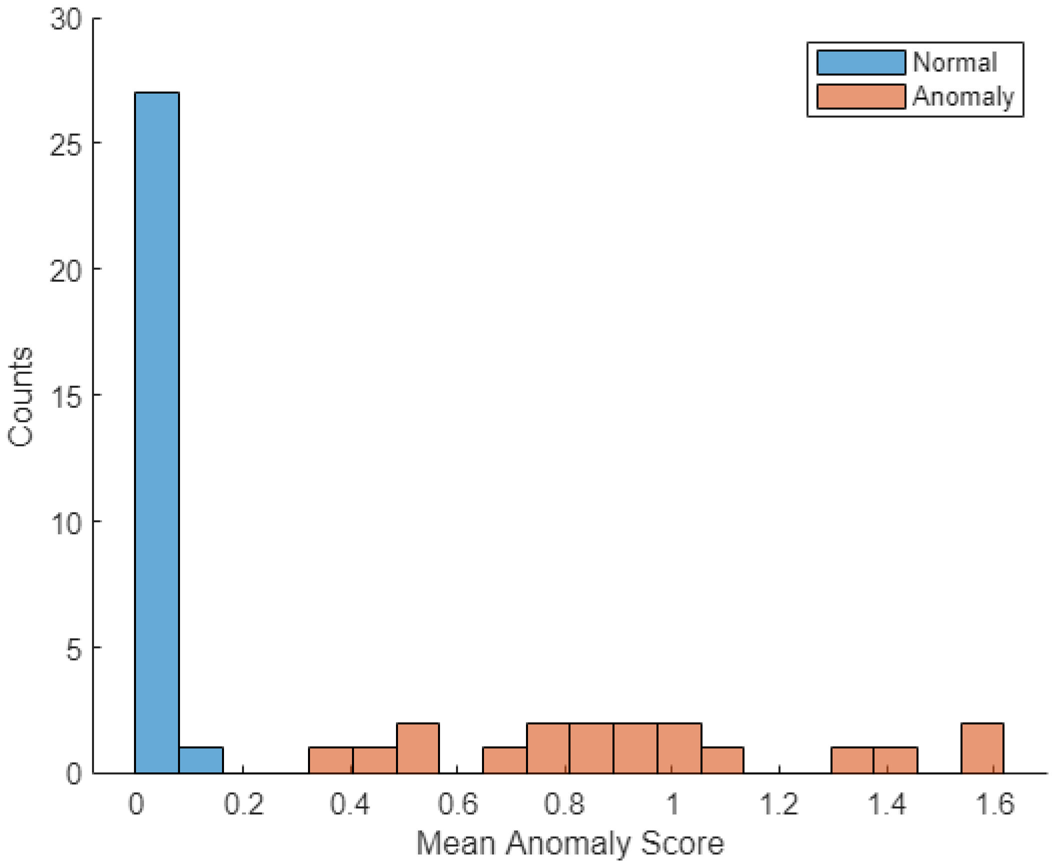
Select Anomaly Threshold

Use the calibration data set to select an anomaly score threshold for the anomaly detector. The detector classifies images based on whether their scores are above or below the threshold value. The calibration data set uses both normal and anomalous images to determine the threshold.

Obtain the mean anomaly score and ground truth label for each image in the calibration set and plot a histogram scores.

```
scores = predict(detector,dsCal);
labels = imdsCal.Labels ~= "normal";

numBins = 20;
[~,edges] = histcounts(scores,numBins);
figure
hold on
hNormal = histogram(scores(labels==0),edges);
```

```
hAnomaly = histogram(scores(labels==1),edges);
hold off
legend([hNormal,hAnomaly],"Normal","Anomaly")
xlabel("Mean Anomaly Score")
ylabel("Counts")
```



Calculate the optimal anomaly threshold using the `anomalyThreshold` (Computer Vision Toolbox) function. Specify the first two input arguments as the ground truth labels, `labels`, and predicted anomaly scores, `scores`, for the calibration data set. Specify the third input argument as `true` because true positive anomaly images have a `labels` value of `true`.

```
thresh = anomalyThreshold(labels,scores,true);
```

Set the `Threshold` property of the anomaly detector to the optimal value.

```
detector.Threshold = thresh;
```

Evaluate Classification Model

At this point, we are interested in evaluating the accuracy of the detector with the test images.

Classify each image in the test set as either normal or anomalous using the anomaly detector, `detector`.

```
testSetOutputLabels = classify(detector,dsTest);
```

Get the ground truth labels of each test image.

```
testSetTargetLabels = dsTest.UnderlyingDatastores{1}.Labels;
```

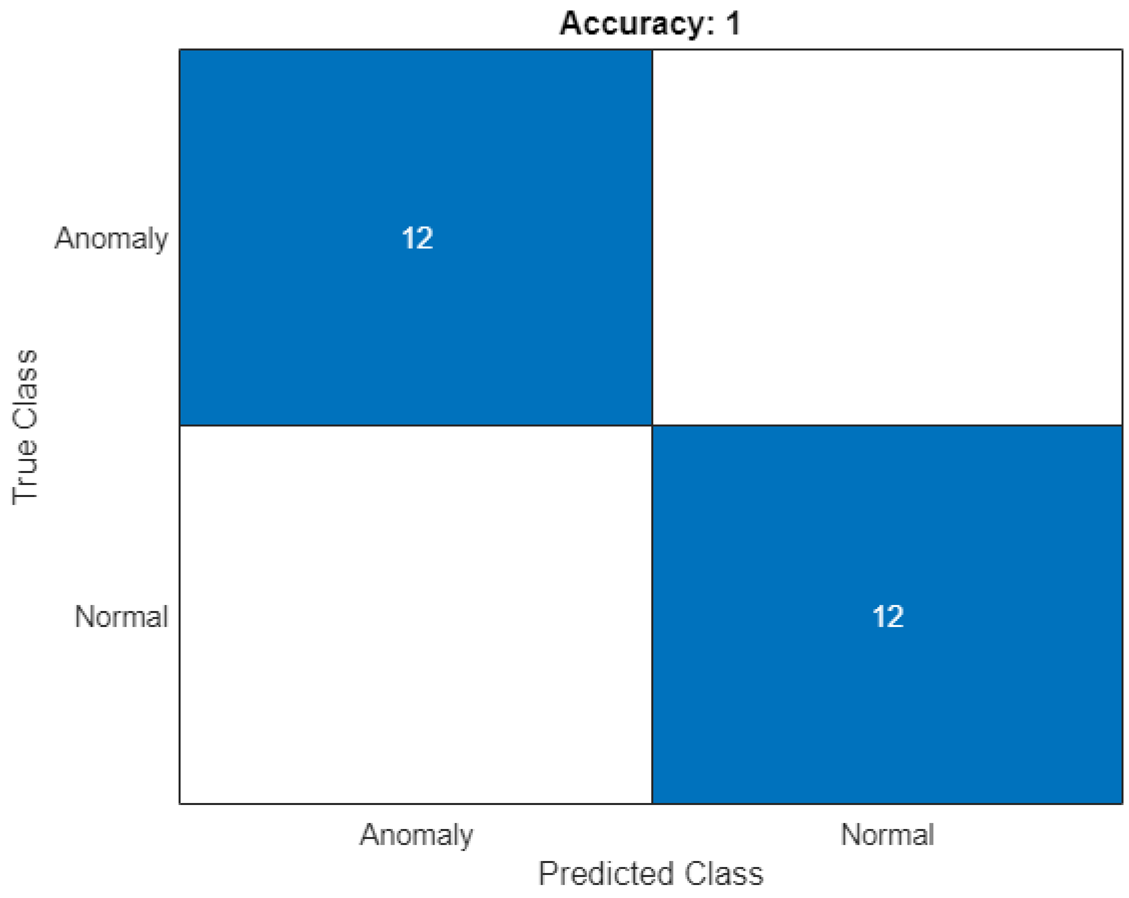
Evaluate the anomaly detector by calculating performance metrics by using the `evaluateAnomalyDetection` (Computer Vision Toolbox) function. The function calculates several metrics that evaluate the accuracy, precision, sensitivity, and specificity of the detector for the test data set.

```
metrics = evaluateAnomalyDetection(testSetOutputLabels,testSetTargetLabels,"dirt");
```

```
Evaluating anomaly detection results
------------------------------------
* Finalizing... Done.
* Data set metrics:

    GlobalAccuracy    MeanAccuracy    Precision    Recall    Specificity    F1Score    FalsePosi
    ______________    ____________    _________    ______    ___________    _______    _________

          1                1              1          1            1            1               0
```

The `ConfusionMatrix` property of `metrics` contains the confusion matrix for the test set. Extract the confusion matrix and display a confusion plot. The recalibrated classification model with the anomaly threshold selected is very accurate and predicts a small percentage of false positives and false negatives.

```
M = metrics.ConfusionMatrix{:,:};
confusionchart(M,["Normal","Anomaly"])
acc = sum(diag(M)) / sum(M,"all");
title("Accuracy: "+acc)
```



For more information about explainable classification decisions, see the “Detect Image Anomalies Using Explainable FCDD Network” (Computer Vision Toolbox) example.

Save the detector to a MAT file for later use in MATLAB or in deployment workflows.

```
save('livePillDetector.mat','detector')
```

Classify Live Images of Pills

You can use a custom video preview from a camera to display the results of the live classification of pills as normal or anomalous. This section demonstrates how to create a custom preview update callback function to process live images, identify individual pills, classify each pill, and show classification results overlayed with the camera preview video. You can find the code from this section as well as the update preview callback function attached to this example as supporting files.

Acquire images using a webcam with the `videoinput` interface at a resolution of 1280x720. Then, use a custom preview update callback function to identify each pill using the `imfindcircles` function, since the pills are circular. Crop the section of the camera image around each pill and pass it to the detector for classification. Based on the classification decision, display a label on each pill indicating whether it is a normal or anomalous pill. Normal pills are labeled as `Good` and anomalous pills are labeled as `Bad` in the preview window.

Load the anomaly detector saved previously.

```
load('livePillDetector.mat');
```

Create a connection to the webcam and specify appropriate values for the device-specific properties. The following device-specific property values are optimized for this hardware setup. Your setup might require a different configuration.

```
vid = videoinput('winvideo',1,'YUY2_1280x720');
vid.ReturnedColorSpace = "rgb";

nBands = vid.NumberOfBands;
vidRes = vid.ROIPosition;
imWidth = vidRes(3);
imHeight = vidRes(4);

src = getselectedsource(vid);
src.ExposureMode = "manual";
src.Exposure = -11;
src.FocusMode = "manual";
src.Focus = 11;
src.WhiteBalanceMode = "manual";
src.WhiteBalance = 2800;
src.FrameRate = "20.0000";
```

Create a figure window and axes in the figure. Then create the image object in which you want to display the video preview data. The size of the image object must match the dimensions of the video frames. Add a scroll panel to the image.

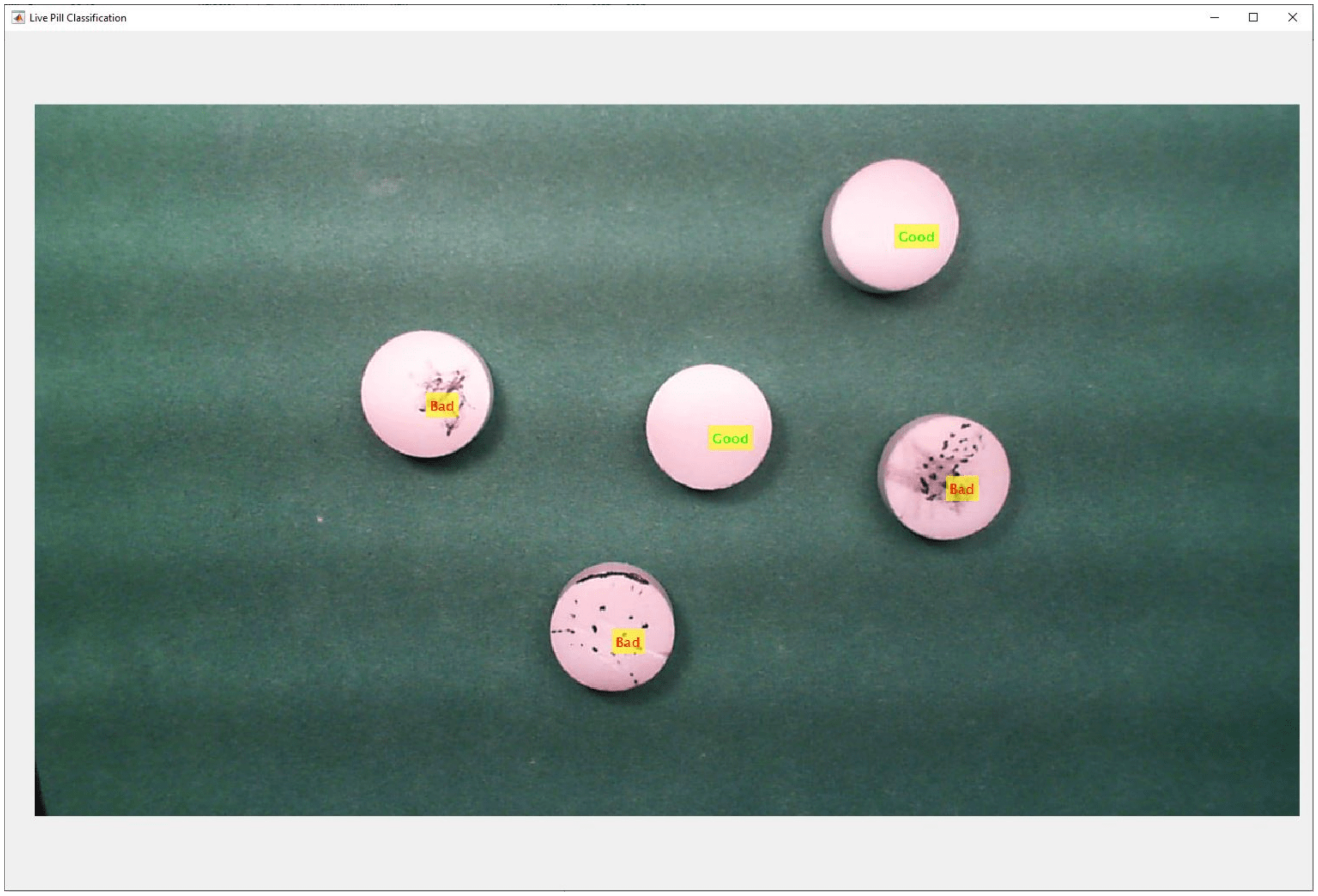
```
hFig = uifigure('Toolbar','none',...
       'Menubar', 'none',...
       'NumberTitle','Off',...
       'AutoResizeChildren','Off',...
       'Name','Live Pill Classification');
hAxis = uiaxes(hFig);
disableDefaultInteractivity(hAxis);
hImage = image(zeros(imHeight, imWidth, nBands),'Parent',hAxis);
imscrollpanel(hFig,hImage);
```

Configure the update preview window callback function and the detector as application-defined data on the image object. To define the `mypreview_fcn` callback function, refer to the Perform Custom Processing of Preview Data section.

```
setappdata(hImage,'UpdatePreviewWindowFcn',@mypreview_fcn);
setappdata(hImage,'Detector',detector);
```

Preview the video data in the custom figure window.

```
preview(vid,hImage);
```



Perform Custom Processing of Preview Data

Create a callback function `mypreview_fcn` that processes each frame and identifies whether pills are normal or dirt. This callback function is called for each acquired frame when you call the `preview` function.

The `mypreview_fcn` callback function performs the following:

1. Processes each frame that is captured and determines the number of circles in it using `imfindcircles`. Since the pills are circular, the `imfindcircles` function identifies the number of pills based on the specified radius range. Determine the radius range by measuring the radius of a pill in pixels using the Distance tool on a captured image. For more information on how to use the Distance tool, see “Measure Distance Between Pixels in Image Viewer App”.
2. Crops the image around each detected circle and passes the cropped image to the classifier for classification.
3. Adds a text label to the center of each circle that identifies the pill as `Good` or `Bad` based on the classification decision.
4. Repeats the cropping and labeling steps for every circle detected.
5. Updates the image in the custom preview window.

Refer to the comments in the code for additional context.

```
function mypreview_fcn(obj,event,hImage)

    persistent detector;
    persistent ImgSize;

    if isempty(detector) && isempty(ImgSize)
        % Anomaly detector
        detector = getappdata(hImage,'Detector');

        % Width and height of images used during calibration
        ImgSize = [352 288];
    end

    % Find circles (pills) in the current image frame
    img = event.Data;
    radiusRange = [30 80];
    circles = imfindcircles(img,radiusRange);

    % Use classifier if circles are found
    if ~isempty(circles)

        % Loop through all circles
        for i = 1:size(circles,1)

            % Crop image around the circle (pill)
            % [pos1, pos2] - top-left corner pixel position
            % ImgSize - width and height from [pos1, pos2]

            pos1 = max(circles(i,1)-ImgSize(1)/2,0);
            pos2 = max(circles(i,2)-ImgSize(2)/2,0);

            croppedImage = imcrop(img,[pos1, pos2,ImgSize(1),ImgSize(2)]);

            if ~isempty(croppedImage)
                % Classify cropped image and assign a text label in the
                % center of the circle
                decision = classify(detector, croppedImage);

                if decision
                    img = insertText(img,circles(i,:),"Bad", TextColor="Red",FontSize=14);
                else
                    img = insertText(img,circles(i,:),"Good", TextColor="Green",FontSize=14);
                end
            end
        end
    end
    hImage.CData = img;
end
```

This code is expected to classify images correctly if there is one pill in the camera's field of view. If there are multiple pills in the field of view, the distance between the pills must be large enough so that the cropped images do not contain more than one pill.

Deploy Live Image Classification as Standalone Application

You can deploy the live image classification of pills as a standalone application using the Application Compiler (MATLAB Compiler) app. You must have a license for MATLAB Compiler. Once you have created the standalone application, you can deploy it to multiple machines without additional MATLAB licenses.

You must do the following in the Application Compiler app to create the standalone application.

- Add customGUIForLiveClassification.m as the main file for the project.
- Add the Image Acquisition Toolbox™ Support Package for OS Generic Video Interface and Computer Vision Toolbox™ Automated Visual Inspection Library from the suggested list of support packages.

See Also
`videoinput` | `getdata` | `imwrite` | `imageDatastore` | `imfindcircles`

Acquire and Analyze Images from FLIR Ax5 Thermal Infrared Camera

This example shows how to acquire, preview, and analyze images from a FLIR Ax5 series thermal infrared camera in multiple ways.

This example uses a FLIR A35 camera, but is expected to work with other cameras in the Ax5 series with minor modifications.

Requirements:

- MATLAB® R2022a
- Image Acquisition Toolbox™
- Image Processing Toolbox™
- Image Acquisition Toolbox Support Package for GenICam™ Interface
- FLIR AX5 GigE Vision® Thermal Infrared Camera
- FLIR GenTL Producer included with the Spinnaker SDK
- Gigabit Ethernet network adapter with jumbo frame support

Connect to Camera and Configure Acquisition

Create a connection to the FLIR AX5 using the `gentl` adapter with the `videoinput` function. Then, get the source properties of the `videoinput` object.

```
vid = videoinput("gentl",1,"Mono14")
```

```
Summary of Video Input Object Using 'FLIR Systems AB FLIR AX5(00:11:1c:02:7a:ef)'.

   Acquisition Source(s):  FLIR AX5(00:11:1c:02:7a:ef)_Stream_0
                           is available.

  Acquisition Parameters:  'FLIR AX5(00:11:1c:02:7a:ef)_Stream_0' is the current selected source
                           10 frames per trigger using the selected source.
                           'Mono14' video data to be logged upon START.
                           Grabbing first of every 1 frame(s).
                           Log data to 'memory' on trigger.

      Trigger Parameters:  1 'immediate' trigger(s) on START.

                  Status:  Waiting for START.
                           0 frames acquired since starting.
                           0 frames available for GETDATA.
```

```
src = getselectedsource(vid);
```

Since the default bit depth is 8-bit, set the preview data to full bit depth. Otherwise, MATLAB shows only half of the data recieved from the camera.

```
vid.PreviewFullBitDepth = "on";
```

Configure Camera Properties

The FLIR A35 camera has a 14-bit depth. Configure the camera to provide image data in temperature linear mode with high temperature resolution.

```
src.CMOSBitDepth = "bit14bit";
src.TemperatureLinearMode = "On";
src.TemperatureLinearResolution = "High";
```

Specify the atmospheric and object parameters. For measurement accuracy, these parameters must be configured correctly before acquiring images. For more information about these parameters, refer to the camera manual.

```
src.AtmosphericTemperature = 298;
src.ObjectEmissivity = 1.0;
src.ReflectedTemperature = 298;
```

The temperature resolution constant is used to convert image data to °C. Set its value to 0.04 or 0.4, depending on the value of the `TemperatureLinearResolution` property. Refer to the camera manual for the appropriate values of the temperature resolution constant.

```
switch src.TemperatureLinearResolution
    case "High"
        k = 0.04;
    case "Low"
        k = 0.4;
end
```

Acquire and Display Image from Camera Then Save them as an RGB Image

Acquire an image from the camera using the `getsnapshot` function.

```
img = getsnapshot(vid);
```

Convert the captured image data to °C using the `ax5degC` helper function with the temperature resolution value `k`.

The helper functions used in this example are attached as supporting files in the same directory as this example file.

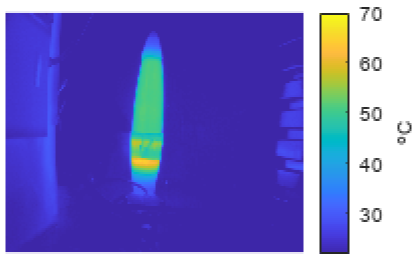
```
imgC = ax5degC(img,k);
```

Specify the temperature range of the display color limits in °C.

```
tempRange = [22 70];
```

Display the thermal image using the `thermalImageShow` helper function with the specified temperature range and units. You can click anywhere on the image to show the temperature information of that pixel.

```
thermalImageShow(imgC,tempRange,"\circC");
```



Convert the temperature intensity image to an RGB image using the `thermal2rgb` helper function with the specified temperature range and colormap. Save the converted image as an image file.

```
RGB = thermal2rgb(imgC,tempRange,parula);
imwrite(RGB,"thermalImageRGB.png")
```

Acquire and Display Video from Camera

Configure the camera's `FramesPerTrigger` acquisition property to capture 20 images.

```
vid.FramesPerTrigger = 20;
```

Acquire image data using the `getdata` function.

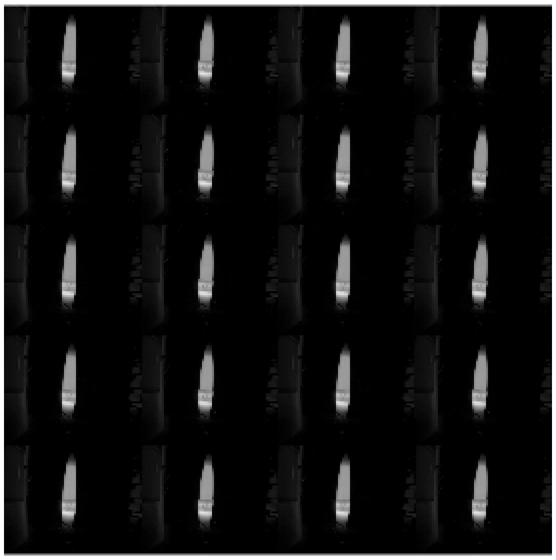
```
start(vid)
frames = getdata(vid,vid.FramesPerTrigger);
```

Convert the captured image data to °C using the `ax5degC` helper function with the temperature resolution value k.

```
framesC = ax5degC(frames,k);
```

Display the sequence of acquired image frames using the `montage` function with the previously specified temperature range.

```
montage(framesC,DisplayRange=tempRange)
```

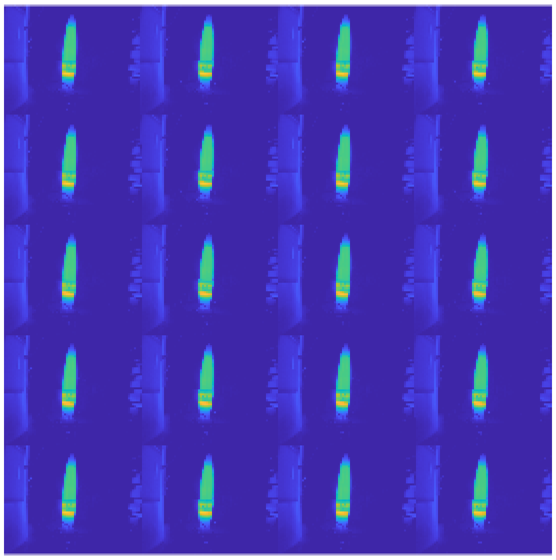


Convert the frames to RGB and apply the parula colormap.

```
frameSize = vid.ROIPosition;
numFrames = size(framesC,4);
framesRGB = zeros(frameSize(4)-frameSize(2),frameSize(3)-frameSize(1),3,numFrames);

for frame = 1:numFrames
    framesRGB(:,:,:,frame) = thermal2rgb(framesC(:,:,:,frame),tempRange,parula);
end
```

Display the sequence of acquired image frames in color with the colormap applied.

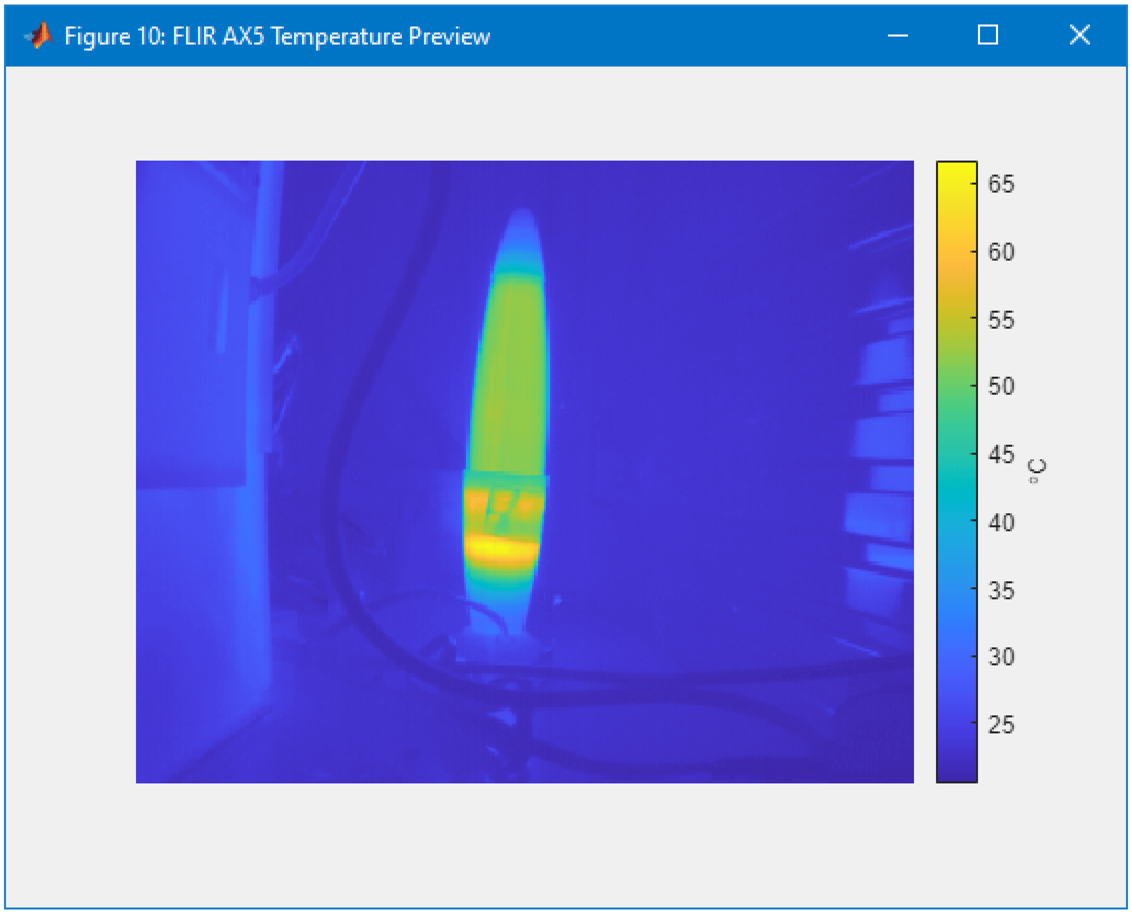
```
montage(framesRGB)
```



Show Live Preview from Camera

Show an interactive live preview from the camera using the `ax5Preview` helper function. The live images from the camera are converted to °C and shown in the preview window. You can click anywhere in the live preview image to view temperature information for the selected pixel.

```
ax5Preview(vid);
```



See Also

`videoinput` | `getdata` | `montage` | `imshow`

Functions

Image Acquisition Explorer

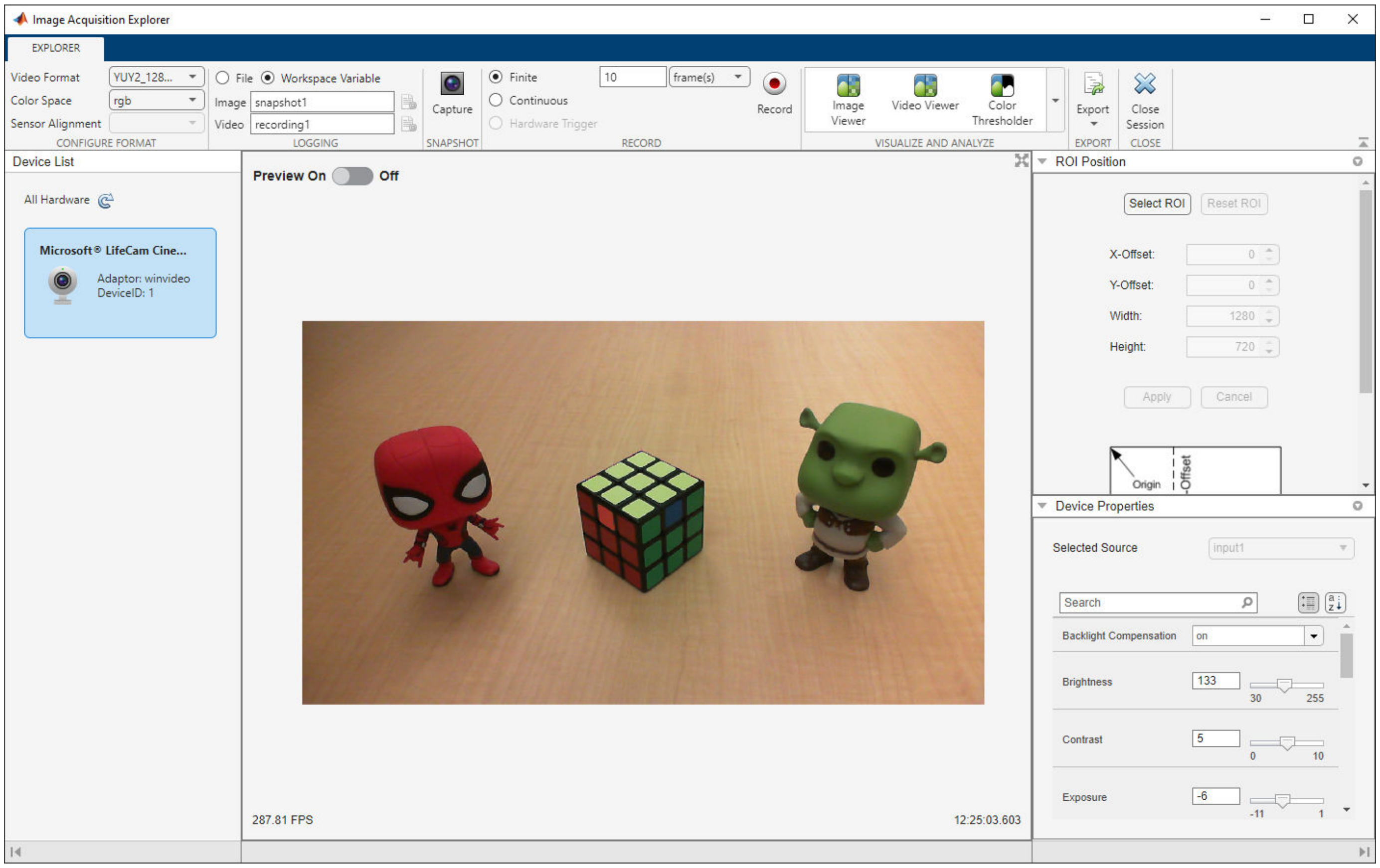
Acquire images and video from hardware

Description

The **Image Acquisition Explorer** app provides a user interface to acquire images and video from cameras and frame grabbers.

Using this app, you can:

- Preview live video data from your image acquisition hardware.
- Configure device-specific properties and acquisition settings such as video format, region of interest, and hardware trigger.
- Save image snapshot and video recording data to a file or to the MATLAB workspace.
- Visualize and analyze saved data by launching Image Processing Toolbox apps.
- Generate a MATLAB live script for app interactions that uses the `videoinput` interface.



Video Format: YUY2_128...
Color Space: rgb
Sensor Alignment:
Image: snapshot1
Video: recording1
10 frame(s)
X-Offset: 0
Y-Offset: 0
Width: 1280
Height: 720
Selected Source: input1
Backlight Compensation: on
Brightness: 133 (30–255)
Contrast: 5 (0–10)
Exposure: -6 (-11–1)
287.81 FPS
12:25:03.603

Open the Image Acquisition Explorer App

- MATLAB Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click the app icon.
- MATLAB command prompt: Enter `imageAcquisitionExplorer`.

Examples

- "Get Started with Image Acquisition Explorer" on page 3-5
- "Select Your Device and Configure Format in Image Acquisition Explorer" on page 3-10
- "Set Acquisition Parameters in Image Acquisition Explorer" on page 3-13
- "Log Data in Image Acquisition Explorer" on page 3-18
- "Preview and Acquire Data in Image Acquisition Explorer" on page 3-22
- "Export Code from Image Acquisition Explorer" on page 3-27

Parameters

Configure Format

`Select Camera File` — Specify device configuration file
button

Click this button to select a device configuration file, also known as a camera file or digitizer configuration format file, from your computer. Some image acquisition devices use these files to store device configuration information. The app uses this file to determine the video format and other configuration information.

When you open the app, the rest of the app toolstrip is disabled until you select a camera file. You can change the camera file at any time.

Note The camera file is provided by the device manufacturer. See your device documentation for more information.

Dependencies

This button is available only for devices that support camera files.

`Camera File` — Name of camera file
selected camera file

This parameter is read-only.

This is the name of the camera file that you selected using the **Select Camera File** button. You can view the full file path by hovering your cursor over the file name.

Dependencies

This parameter is available only for devices that support camera files.

`Video Format` — Video format
supported video formats

Select the video format used by the device to capture images and video. The list of values for this parameter depends on the video formats supported by your device. The format selected when you open the app is the device's default format.

Dependencies

This parameter is available only for devices that support video format.

`Color Space` — Color space used in MATLAB
`grayscale` | `rgb` | `YCbCr` | `bayer`

Select the color space that the app uses when it returns image data. The list of values for this parameter depends on the **Video Format** selected.

If you select `grayscale`, you can set the **Colormap** and **Color Limits** parameters.

- **Color Limit** — Toggle this switch to **Manual** to set the minimum and maximum values on the specified colormap. The default values are 0 and 255. All values in the preview that are less than or equal to the minimum value map to the lowest value of the colormap. All values in the preview that are greater than or equal to the maximum value map to the highest value of the colormap.
- **Colormap** — Colormap applied to the preview. For a full list of options, see `map`.

If you select `bayer`, you can set the **Sensor Alignment** parameter.

`Sensor Alignment` — Sensor alignment for Bayer demosaicing
`grbg` (default) | `gbrg` | `rggb` | `bggr`

Select the 2-by-2 pixel Bayer color filter array pattern of the Bayer color filter array. The specified pattern is used to convert the Bayer pattern image to RGB. There are four possible sensor alignments. For more information about which one to select, refer to your device documentation.

Value	Description

`gbrg`	The 2-by-2 sensor alignment is
	
	`green blue`
	`red   green`

`grbg`	The 2-by-2 sensor alignment is
	
	`green red`
	`blue  green`

`bggr`	The 2-by-2 sensor alignment is
	
	`blue  green`
	`green red`

Value	Description

`rggb`	The 2-by-2 sensor alignment is
	`red   green`
	`green blue`

Dependencies

This parameter is enabled only if your device supports Bayer sensor alignment and **Color Space** is set to `bayer`.

Logging

`Image` — Specify image file or workspace variable name to log snapshot data
valid file name | valid variable name

Edit the name of the image file or name of the workspace variable to save snapshot image data as.

- If you select the **File** option, this parameter defines the image file name. You can click the configuration icon next to this parameter for additional settings, including the file location to save to and file format to save as.
- If you select the **Workspace Variable** option, this parameter defines the workspace variable name.

When you click the **Capture** button, the snapshot image data is saved as the specified file or workspace variable.

For more information, see “Log Data in Image Acquisition Explorer” on page 3-18.

`Video` — Specify video file or workspace variable name to log recorded data
valid file name | valid variable name

Edit the name of the video file or name of the workspace variable to save recorded data as.

- If you select the **File** option, this parameter defines the video file name. You can click the configuration icon next to this parameter for additional settings, including the file location to save to and file format to save as.
- If you select the **Workspace Variable** option, this parameter defines the workspace variable name.

When you click the **Record** button, the recorded data is saved as the specified file or workspace variable.

For more information, see “Log Data in Image Acquisition Explorer” on page 3-18.

Snapshot

`Capture` — Take image snapshot
button

Click this button to immediately capture a single image frame and save it as an image file or as a workspace variable, depending on your selection of **File** or **Workspace Variable** in the **Logging** section. For more information, see “Capture Image Snapshot” on page 3-23.

Record

`Finite` — Select finite recording option
button

Set the recording mode as finite and specify the number of frames or seconds to save when you click the **Record** button. When you select this option, you have two options for finite recording.

- Specify the number of `frame(s)` to record. For more information, see “Record Finite Number of Frames” on page 3-24.
- Specify the number of `second(s)` to record. For more information, see “Record for Finite Duration” on page 3-24.

`Continuous` — Select continuous recording option
button

Set the recording mode as continuous to start saving frames when you click the **Record** button. For more information, see “Record Continuously” on page 3-25.

`Hardware Trigger` — Select hardware triggered recording option
button

Set the recording mode as hardware trigger. Hardware triggered acquisition is supported for GigE Vision and GenICam GenTL devices. When you select this option, the app opens a **Hardware Trigger** tab. You can define the following hardware trigger parameters.

- **Number of Triggers**
- **Frames per Trigger**
- **Trigger Source**
- **Trigger Condition**

If you are using this recording mode, make sure you also enable hardware triggered acquisition in the **Device Properties** by setting **Trigger Mode** to `On` and specifying other **Trigger Selector** parameters for your setup.

For more information, see “Set Up Hardware Triggering” on page 3-16 and “Record with Hardware Trigger” on page 3-25.

Dependencies

This parameter is enabled only if your device supports hardware triggered acquisition.

`Record` — Record video
button

Click this button Record to acquire multiple frames and save them as a video file or as a workspace variable, depending on your selection of **File** or **Workspace Variable** in the **Logging** section. This button becomes a **Stop** button after you click it. End recording at any time and save the recorded data by clicking **Stop**.

While you are recording, the app toolstrip and all property tabs are disabled. You can not change the value of any parameters during recording.

For more information, see “Record Video” on page 3-24.

Visualize and Analyze

`Image Viewer` — View captured snapshot in Image Viewer app
button

Click this button Image Viewer to launch the **Image Viewer** app and send it the most recent image data captured in this app session.

You must have Image Processing Toolbox installed to use the **Image Viewer** app.

`Video Viewer` — View recorded video in Video Viewer app
button

Click this button Video Viewer to launch the **Video Viewer** app and send it the most recent video data recorded in this app session.

You must have Image Processing Toolbox installed to use the **Video Viewer** app.

`Color Thresholder` — View captured snapshot in Color Thresholder app
button

Click this button Color Thresholder to launch the **Color Thresholder** app and send it the most recent image data captured in this app session.

You must have Image Processing Toolbox installed to use the **Color Thresholder** app.

Export

`Export` — Export MATLAB code
`Generate Snapshot Script` | `Generate Record Script`

Click this button Export to select an option to generate a MATLAB live script for capturing a snapshot or recording a video and open it in the Live Editor. The live script contains code for the current device configuration, as specified in the **Configure Format** section, and code for saving data as a file or workspace variable, as specified in the **Logging** section.

For more information, see “Export Code from Image Acquisition Explorer” on page 3-27.

Version History

Introduced in R2022a

See Also

Functions
`videoinput`

Topics
"Get Started with Image Acquisition Explorer" on page 3-5
"Select Your Device and Configure Format in Image Acquisition Explorer" on page 3-10
"Set Acquisition Parameters in Image Acquisition Explorer" on page 3-13
"Log Data in Image Acquisition Explorer" on page 3-18
"Preview and Acquire Data in Image Acquisition Explorer" on page 3-22
"Export Code from Image Acquisition Explorer" on page 3-27

Image Acquisition Tool

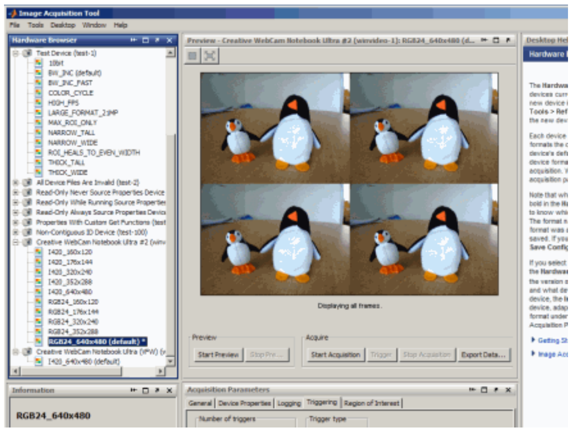
(Removed) Acquire images and video from hardware

Note The **Image Acquisition Tool** app has been removed. Use the **Image Acquisition Explorer** app instead.

Description

The **Image Acquisition Tool** enables you to explore, configure, and acquire data from your installed and supported image acquisition devices. You connect directly to your hardware and can preview and acquire image data.

Using this app, you can log the acquired image data to MATLAB in several formats, and also generate a VideoWriter file. The Image Acquisition Tool provides a desktop environment that integrates a preview and acquisition area with acquisition parameters so that you can change settings and see the changes dynamically applied to your image data.



Open the Image Acquisition Tool App

- MATLAB Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click the app icon.
- MATLAB command prompt: Enter `imaqtool`.

Programmatic Use

`imaqtool` opens the Image Acquisition Tool, which enables you to explore, configure, and acquire data from image acquisition devices.

`imaqtool(file)` starts the tool and then immediately reads an Image Acquisition Tool configuration file, where `file` is the name of an IAT-file that you previously saved.

Version History

Introduced in R2007b

R2022b: Removed

Errors starting in R2022b

The **Image Acquisition Tool** app (`imaqtool`) has been removed. Use **Image Acquisition Explorer** (`imageAcquisitionExplorer`) instead. The new **Image Acquisition Explorer** app has expanded workflows.

R2022a: To be removed

Warns starting in R2022a

The **Image Acquisition Tool** app (`imaqtool`) will be removed in a future release.

See Also

Image Acquisition Explorer

clear

Clear image acquisition object from MATLAB workspace

Syntax

```
clear obj
```

Description

`clear obj` removes the image acquisition object `obj` from the MATLAB workspace. `obj` can be either a video input object or a video source object.

It is important to note that if you clear a video input object that is running (the `Running` property is set to `'on'`), the object continues executing.

You can restore cleared objects to the MATLAB workspace with the `imaqfind` function.

To remove an image acquisition object from memory, use the `delete` function.

Version History

Introduced before R2006a

See Also

`delete` | `imaqfind` | `isvalid`

closepreview

Close Video Preview window

Syntax

```
closepreview(obj)
closepreview
```

Description

`closepreview(obj)` stops the image acquisition object `obj` from previewing and, if the default Video Preview window was used, closes the window.

`closepreview` stops all image acquisition objects from previewing and, for all image acquisition objects that used the default Video Preview window, closes the windows.

Note that if the preview window was created with a user-specified image object handle as the target, `closepreview` does not close the figure window.

Version History

Introduced before R2006a

See Also

`preview` | `stoppreview`

commands

List of commands available for GigE Vision camera

Syntax

```
commands(g)
```

Description

`commands(g)` lists the available commands for the GigE Vision camera `g`, where `g` is the object created using the `gigecam` function. The output depends on the commands that are supported by your specific hardware.

Examples

List Commands Available for GigE Vision

The `commands` function tells you what commands are available for your camera to use.

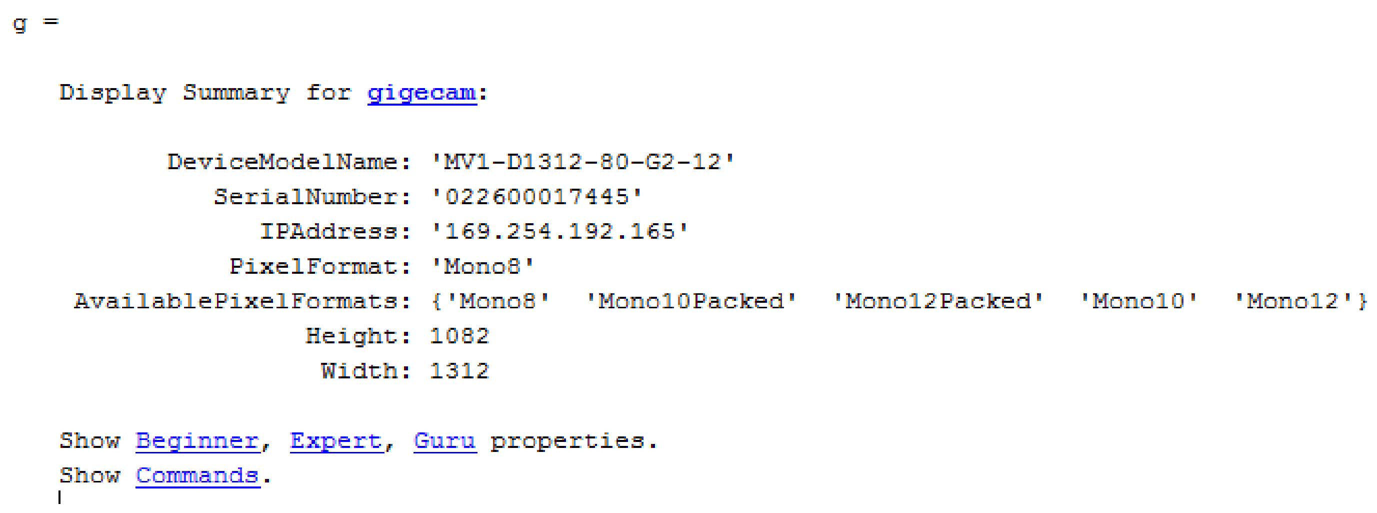
Use the `gigecamlist` function to ensure that MATLAB is discovering your cameras.

```
gigecamlist
```

```
ans =

    Model                 Manufacturer             IPAddress        SerialNumber
  ____________________    ___________________    _______________    ______________

  'MV1-D1312-80-G2-12'    'Photonofocus AG'      '169.254.192.165'  '022600017445'
```

Use the `gigecam` function to create the object and connect it to the camera.

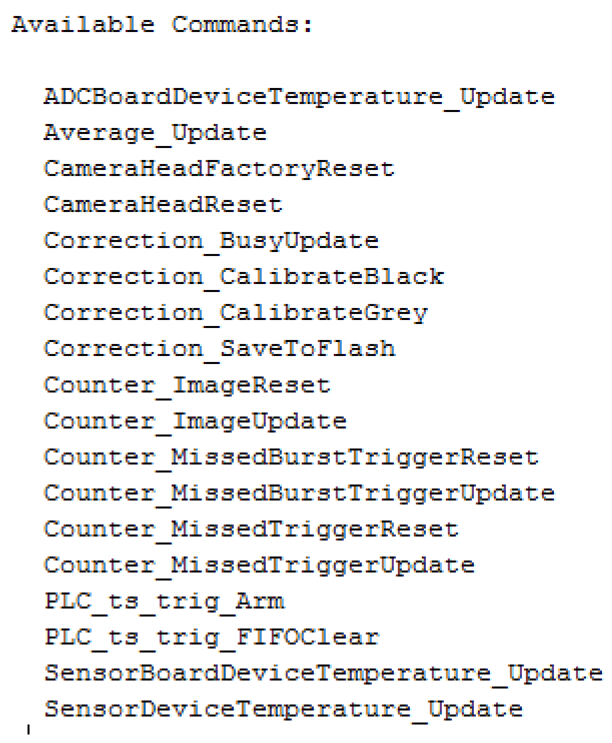
```
g = gigecam
```



```
g =

   Display Summary for gigecam:

          DeviceModelName: 'MV1-D1312-80-G2-12'
             SerialNumber: '022600017445'
                IPAddress: '169.254.192.165'
              PixelFormat: 'Mono8'
    AvailablePixelFormats: {'Mono8'  'Mono10Packed'  'Mono12Packed'  'Mono10'  'Mono12'}
                   Height: 1082
                    Width: 1312

   Show Beginner, Expert, Guru properties.
   Show Commands.
```

Get the list of supported commands from the camera. You can click **Show Commands** in the property list that is displayed when you create the object, or you can use the function:

```
commands(g)
```



```
Available Commands:

  ADCBoardDeviceTemperature_Update
  Average_Update
  CameraHeadFactoryReset
  CameraHeadReset
  Correction_BusyUpdate
  Correction_CalibrateBlack
  Correction_CalibrateGrey
  Correction_SaveToFlash
  Counter_ImageReset
  Counter_ImageUpdate
  Counter_MissedBurstTriggerReset
  Counter_MissedBurstTriggerUpdate
  Counter_MissedTriggerReset
  Counter_MissedTriggerUpdate
  PLC_ts_trig_Arm
  PLC_ts_trig_FIFOClear
  SensorBoardDeviceTemperature_Update
  SensorDeviceTemperature_Update
```

The list shows the commands that the camera supports. You can then use the `executeCommand` function to execute any of these commands.

Version History

Introduced in R2014b

See Also

`gigecamlist` | `gigecam` | `snapshot` | `executeCommand`

delete

Remove image acquisition object from memory

Syntax

`delete(obj)`

Description

`delete(obj)` removes `obj`, an image acquisition object or array of image acquisition objects, from memory. Use `delete` to free memory at the end of an image acquisition session.

If `obj` is an array of image acquisition objects and one of the objects cannot be deleted, the `delete` function deletes the objects that can be deleted and returns a warning.

When `obj` is deleted, it becomes invalid and cannot be reused. Use the `clear` command to remove invalid image acquisition objects from the MATLAB workspace.

If multiple references to an image acquisition object exist in the workspace, deleting the image acquisition object invalidates the remaining references. Use the `clear` command to delete the remaining references to the object from the workspace.

If the image acquisition object `obj` is running or being previewed, the `delete` function stops the object and closes the preview window before deleting it.

Examples

Create a video object, preview the object, then delete the object:

```
vid = videoinput('winvideo', 1);
preview(vid);
delete(vid);
```

Version History

Introduced before R2006a

See Also

`imaqfind` | `isvalid` | `videoinput`

disp

Display method for image acquisition objects

Syntax

```
obj
disp(obj)
```

Description

`obj` displays summary information for image acquisition object `obj`.

`disp(obj)` displays summary information for image acquisition object `obj`.

If `obj` is an array of image acquisition objects, `disp` outputs a table of summary information about the image acquisition objects in the array.

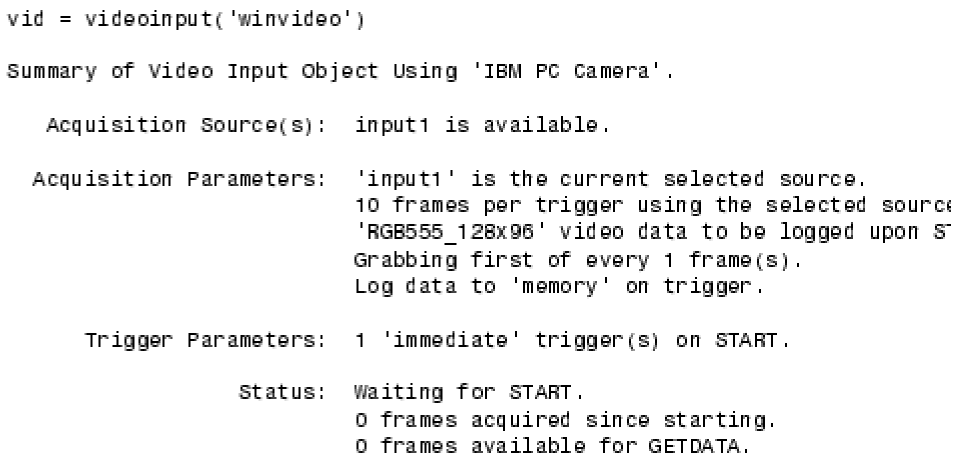
In addition to the syntax shown above, you can display summary information for `obj` by excluding the semicolon when:

- Creating an image acquisition object, using the `videoinput` function
- Configuring property values using the dot notation

Examples

This example illustrates the summary display of a video input object.

```
vid = videoinput('winvideo')
```



```
vid = videoinput('winvideo')

Summary of Video Input Object Using 'IBM PC Camera'.

   Acquisition Source(s):  input1 is available.

  Acquisition Parameters:  'input1' is the current selected source.
                           10 frames per trigger using the selected sourc
                           'RGB555_128x96' video data to be logged upon S
                           Grabbing first of every 1 frame(s).
                           Log data to 'memory' on trigger.

      Trigger Parameters:  1 'immediate' trigger(s) on START.

                  Status:  Waiting for START.
                           0 frames acquired since starting.
                           0 frames available for GETDATA.
```

This example shows the summary information displayed for an array of video input objects.

```
vid2 = videoinput('winvideo');

[vid vid2]
```

```
   Video Input Object Array:

   Index:   Type:          Name:
   1        videoinput     RGB555_128x96-winvideo-1
   2        videoinput     RGB555_128x96-winvideo-1
```

Version History

Introduced before R2006a

See Also

`videoinput`

executeCommand

Execute command on GigE Vision camera

Syntax

```
executeCommand(g, 'commandname')
```

Description

`executeCommand(g, 'commandname')` executes the specified command for the GigE Vision camera `g`, where `g` is the object created using the `gigecam` function, and `'commandname'` is the name of the command to execute.

Use the `commands` function to get the list of available commands for your camera.

Examples

Execute a Command to Set the Calibration on GigE Vision Camera

Use `executeCommand` to execute any of the commands found by the `commands` function, which tells you what commands are available for your camera to use.

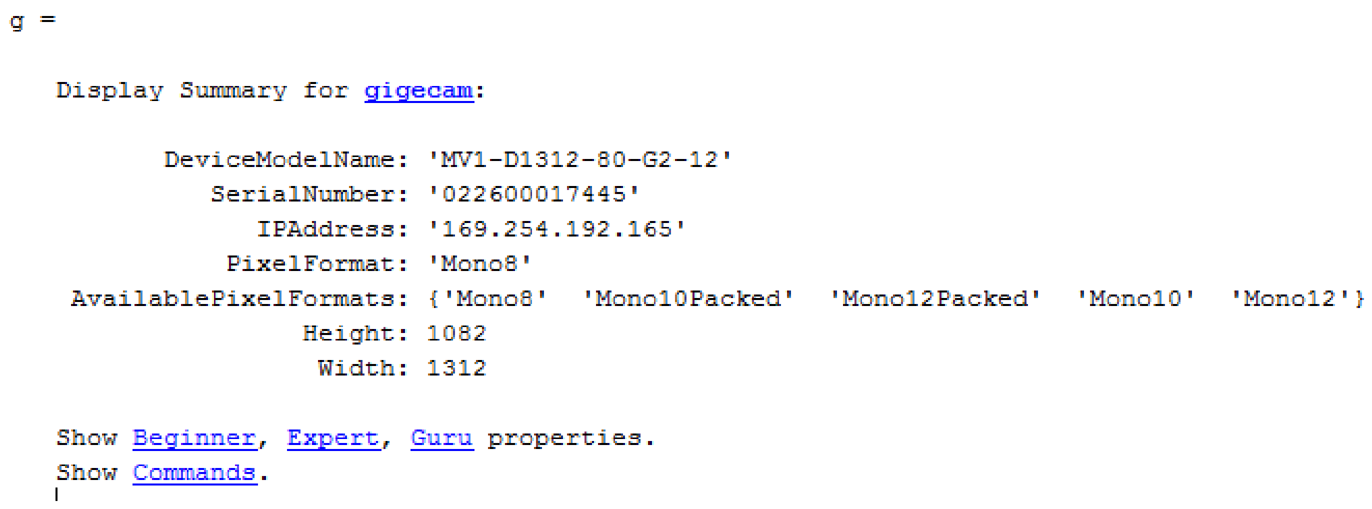
Use the `gigecamlist` function to ensure that MATLAB is discovering your camera.

```
gigecamlist
```

```
ans = 

    Model              Manufacturer          IPAddress       SerialNumber
  ____________________    ___________________    _______________    ______________

  'MV1-D1312-80-G2-12'   'Photonofocus AG'     '169.254.192.165'  '022600017445'
```

Use the `gigecam` function to create the object and connect it to the camera.

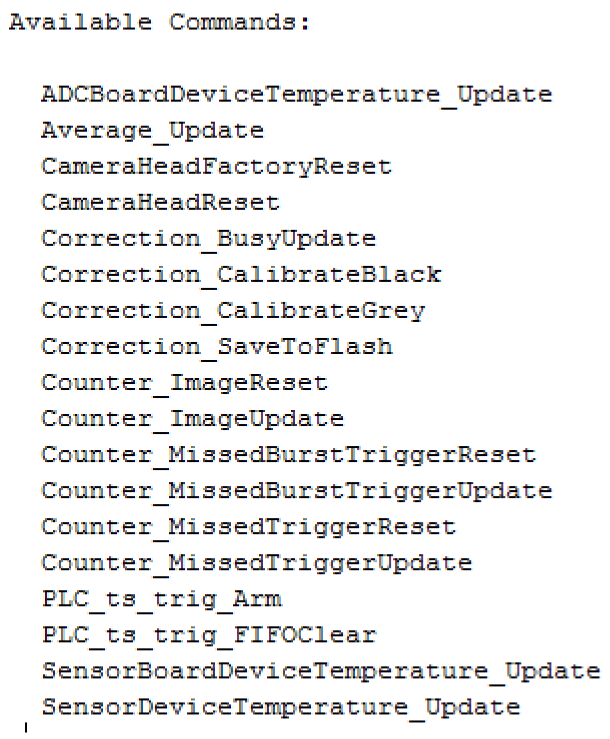
```
g = gigecam
```



```
g =

   Display Summary for gigecam:

          DeviceModelName: 'MV1-D1312-80-G2-12'
             SerialNumber: '022600017445'
                IPAddress: '169.254.192.165'
              PixelFormat: 'Mono8'
    AvailablePixelFormats: {'Mono8'  'Mono10Packed'  'Mono12Packed'  'Mono10'  'Mono12'}
                   Height: 1082
                    Width: 1312

   Show Beginner, Expert, Guru properties.
   Show Commands.
```

Get the list of supported commands from the camera. You can click **Show Commands** in the property list that is displayed when you create the object, or you can use the function:

```
commands(g)
```



```
 Available Commands:

   ADCBoardDeviceTemperature_Update
   Average_Update
   CameraHeadFactoryReset
   CameraHeadReset
   Correction_BusyUpdate
   Correction_CalibrateBlack
   Correction_CalibrateGrey
   Correction_SaveToFlash
   Counter_ImageReset
   Counter_ImageUpdate
   Counter_MissedBurstTriggerReset
   Counter_MissedBurstTriggerUpdate
   Counter_MissedTriggerReset
   Counter_MissedTriggerUpdate
   PLC_ts_trig_Arm
   PLC_ts_trig_FIFOClear
   SensorBoardDeviceTemperature_Update
   SensorDeviceTemperature_Update
```

Execute a command, such as setting a calibration correction.

```
executeCommand(g, 'Correction_CalibrateGrey');
```

Input Arguments

commandname — Name of GigE Vision camera command to execute
character vector

Name of command you want to execute on your GigE Vision camera, specified as a character vector. Use the `commands` function to get the list of available commands for your camera. Then use `executeCommand` to execute any of the available commands.

Example: `executeCommand(g, 'AutoFocus')`

Data Types: `char` | `string`

Version History

Introduced in R2014b

See Also

`gigecamlist` | `gigecam` | `snapshot` | `commands`

flushdata

Remove data from memory buffer used to store acquired image frames

Syntax

```
flushdata(obj)
flushdata(obj,mode)
```

Description

`flushdata(obj)` removes all the data from the memory buffer used to store acquired image frames. `obj` can be a single video input object or an array of video input objects.

`flushdata(obj,mode)` removes all the data from the memory buffer used to store acquired image frames, where *`mode`* can have either of the following values:

Mode	Description
`'all'`	Removes all the data from the memory buffer and sets the `FramesAvailable` property to 0 for the video input object `obj`. This is the default mode when none is specified, `flushdata(obj)`.
`'triggers'`	Removes data from the memory buffer that was acquired during the oldest trigger executed. `TriggerRepeat` must be greater than 0 and `FramesPerTrigger` must not be set to `inf`.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Version History

Introduced before R2006a

See Also

`getdata` | `imaqhelp` | `peekdata` | `propinfo` | `videoinput`

get

Return image acquisition object properties

Syntax

```
get(obj)
V = get(obj)
V = get(obj,PropertyName)
```

Description

`get(obj)` displays all property names and their current values for image acquisition object `obj`.

`V = get(obj)` returns a structure, `V`, in which each field name is the name of a property of `obj` and each field contains the value of that property.

`V = get(obj,`*`PropertyName`*`)` returns the value of the property specified by *`PropertyName`* for image acquisition object `obj`. Use the `get(obj)` syntax to view a list of all the properties supported by a particular image acquisition object.

If *`PropertyName`* is a 1-by-N or N-by-1 cell array of character vectors containing property names, `V` is a 1-by-N cell array of values. If `obj` is a vector of image acquisition objects, `V` is an M-by-N cell array of property values where M is equal to the length of `obj` and N is equal to the number of properties specified.

Examples

Create video object, then get values of two frame-related properties, then display all properties of the object:

```
vid = videoinput('matrox', 1);
get(vid, {'FramesPerTrigger','FramesAcquired'})
out = get(vid, 'LoggingMode')
get(vid);
```

Instead of using `get` to query individual property values, you should use dot notation. So for example, instead of this:

```
get(vid, 'FramesPerTrigger')
```

You should use this syntax:

```
vid.FramesPerTrigger
```

Version History

Introduced before R2006a

See Also

`set` | `videoinput`

getdata

Acquired image frames to MATLAB workspace

Syntax

```
data = getdata(obj)
data = getdata(obj,n)
data = getdata(obj,n,type)
data = getdata(obj,n,type,format)
[data,time] = getdata(...)
[data, time, metadata] = getdata(...)
```

Description

`data = getdata(obj)` returns `data`, which contains the number of frames specified in the `FramesPerTrigger` property of the video input object `obj`. `obj` must be a 1-by-1 video input object.

`data` is returned as an H-by-W-by-B-by-F matrix where

`H`	Image height, as specified in the object's `ROIPosition` property
`W`	Image width, as specified in the object's `ROIPosition` property
`B`	Number of color bands, as specified in the `NumberOfBands` property
`F`	The number of frames returned

`data` is returned to the MATLAB workspace in its native data type using the color space specified by the `ReturnedColorSpace` property.

You can use the MATLAB `image` or `imagesc` functions to view the returned data. Use `imaqmontage` to view multiple frames at once.

`data = getdata(obj,n)` returns `n` frames of data associated with the video input object `obj`.

`data = getdata(obj,n,`*`type`*`)` returns `n` frames of data associated with the video input object `obj`, where *`type`* is one of the character vectors in the following table that specify the data type used to store the returned data.

Type Character Vector	Data Type
`'uint8'`	Unsigned 8-bit integer
`'uint16'`	Unsigned 16-bit integer
`'uint32'`	Unsigned 32-bit integer
`'single'`	Single precision
`'double'`	Double precision
`'native'`	Uses native data type. This is the default.

If *`type`* is not specified, `'native'` is used as the default. If there is no MATLAB data type that matches the object's native data type, `getdata` chooses a MATLAB data type that preserves numerical accuracy. For example, the components of 12-bit RGB color data would each be returned as `uint8` data.

`data = getdata(obj,n,`*`type`*`,`*`format`*`)` returns `n` frames of data associated with the video input object `obj`, where *`format`* is one of the character vectors in the following table that specify the MATLAB format of `data`.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Format Character Vector	Description
`'numeric'`	Returns `data` as an H-by-W-by-B-by-F array. This is the default format if none is specified.
`'cell'`	Returns data as an F-by-1 cell array of H-by-W-by-B matrices

`[data,time] = getdata(...)` returns `time`, an F-by-1 matrix, where F is the number of frames returned in `data`. Each element of `time` indicates the relative time, in seconds, of the corresponding frame in `data`, relative to the first trigger.

`time = 0` is defined as the point at which data logging begins. When data logging begins, the object's `Logging` property is set to `'On'`. `time` is measured continuously with respect to 0 until the acquisition stops. When the acquisition stops, the object's `Running` property is set to `'Off'`.

`[data, time, metadata] = getdata(...)` returns `metadata`, an F-by-1 array of structures, where F is the number of frames returned in `data`. Each structure contains information about the corresponding frame in `data`. The `metadata` structure contains these fields:

Metadata Field	Description
`'AbsTime'`	Absolute time the frame was acquired, expressed as a time vector
`'FrameNumber'`	Number identifying the *n*th frame since the `start` command was issued
`'RelativeFrame'`	Number identifying the *n*th frame relative to the start of a trigger
`'TriggerIndex'`	Number of the trigger in which this frame was acquired

In addition to the fields in the above table, some adaptors may choose to add other adaptor-specific metadata as well.

`getdata` is a blocking function that returns execution control to the MATLAB workspace after the requested number of frames becomes available within the time period specified by the object's `Timeout` property. The object's `FramesAvailable` property is automatically reduced by the number of frames returned by `getdata`. If the requested number of frames is greater than the frames to be acquired, `getdata` returns an error.

It is possible to issue a **Ctrl+C** while `getdata` is blocking. This does not stop the acquisition but does return control to MATLAB.

Examples

Construct a video input object associated with a Matrox device at ID 1.

```
obj = videoinput('matrox', 1);
```

Initiate an acquisition and access the logged data.

```
start(obj);
data = getdata(obj);
```

Display each image frame acquired.

```
imaqmontage(data);
```

Remove the video input object from memory.

```
delete(obj);
```

Version History

Introduced before R2006a

See Also

`getsnapshot` | `imaqhelp` | `imaqmontage` | `peekdata` | `propinfo`

getselectedsource

Return currently selected video source object

Syntax

```
src = getselectedsource(obj)
```

Description

`src = getselectedsource(obj)` searches all the video source objects associated with the video input object `obj` and returns the video source object, `src`, that has the `Selected` property value set to `'on'`.

To select a source for acquisition, use the `SelectedSourceName` property of the video input object.

`obj` must be a 1-by-1 video input object.

Version History

Introduced before R2006a

See Also

`imaqhelp` | `get` | `videoinput`

getsnapshot

Immediately return single image frame

Syntax

```
frame = getsnapshot(obj)
[frame, metadata] = getsnapshot(obj)
```

Description

`frame = getsnapshot(obj)` immediately returns one single image frame, `frame`, from the video input object `obj`. The frame of data returned is independent of the video input object `FramesPerTrigger` property and has no effect on the value of the `FramesAvailable` or `FramesAcquired` property.

The object `obj` must be a 1-by-1 video input object.

`frame` is returned as an H-by-W-by-B matrix where

H	Image height, as specified in the `ROIPosition` property
W	Image width, as specified in the `ROIPosition` property
B	Number of bands associated with `obj`, as specified in the `NumberOfBands` property

`frame` is returned to the MATLAB workspace in its native data type using the color space specified by the `ReturnedColorSpace` property.

You can use the MATLAB `image` or `imagesc` function to view the returned data.

`[frame, metadata] = getsnapshot(obj)` returns metadata, a 1-by-1 array of structures. This structure contains information about the corresponding frame. The metadata structure contains the field `AbsTime`, which is the absolute time the frame was acquired, expressed as a time vector. In addition to that field, some adaptors may choose to add other adaptor-specific metadata as well.

Note If `obj` is running but not logging, and has been configured with a hardware trigger, a timeout error will occur.

To interrupt the `getsnapshot` function and return control to the MATLAB command line, issue the `^C` (**Ctrl+C**) command.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Examples

Create a video input object.

```
obj = videoinput('matrox', 1);
```

Acquire and display a single frame of data.

```
frame = getsnapshot(obj);
image(frame);
```

Remove the video input object from memory.

```
delete(obj);
```

For an example of using `getsnapshot`, see the Image Acquisition Toolbox example **Acquiring a Single Image in a Loop** in the **Examples** list at the top of the Image Acquisition Toolbox main Documentation Center page, or open the file demoimaq_GetSnapshot.m in the MATLAB Editor.

Version History

Introduced before R2006a

See Also

`getdata` | `imaqhelp` | `peekdata`

gigecam

Create `gigecam` object to acquire images from GigE Vision cameras

Syntax

```
g = gigecam
g = gigecam('IPAddress')
g = gigecam(devicenumber)
g = gigecam('serialnumber')
```

Description

`g = gigecam` creates the `gigecam` object `g` and connects to the single GigE Vision camera on your system. If you have multiple cameras and you use the `gigecam` function with no input argument, then it creates the object and connects it to the first camera it finds listed in the output of the `gigecamlist` function.

When the `gigecam` object is created, it connects to the camera and establishes exclusive access. You can then preview the data and acquire images using the `snapshot` function.

`g = gigecam('IPAddress')` creates a `gigecam` object `g` where `IPAddress` is a character vector value that identifies a particular camera by its IP address and connects it to the camera with that address.

`g = gigecam(devicenumber)` creates a `gigecam` object `g`, where `devicenumber` is a numeric scalar value that identifies a particular camera by its index number, and connects it to that camera.

`g = gigecam('serialnumber')` creates a `gigecam` object `g` where `serialnumber` is a character vector value that identifies a particular camera by its serial number.

Examples

Create a gigecam Object Using No Arguments

Use the `gigecam` function with no input arguments to connect to the single GigE Vision camera on your system. If you have multiple cameras and you use the `gigecam` function with no input argument, it creates the object and connects it to the first camera it finds listed in the output of the `gigecamlist` function.

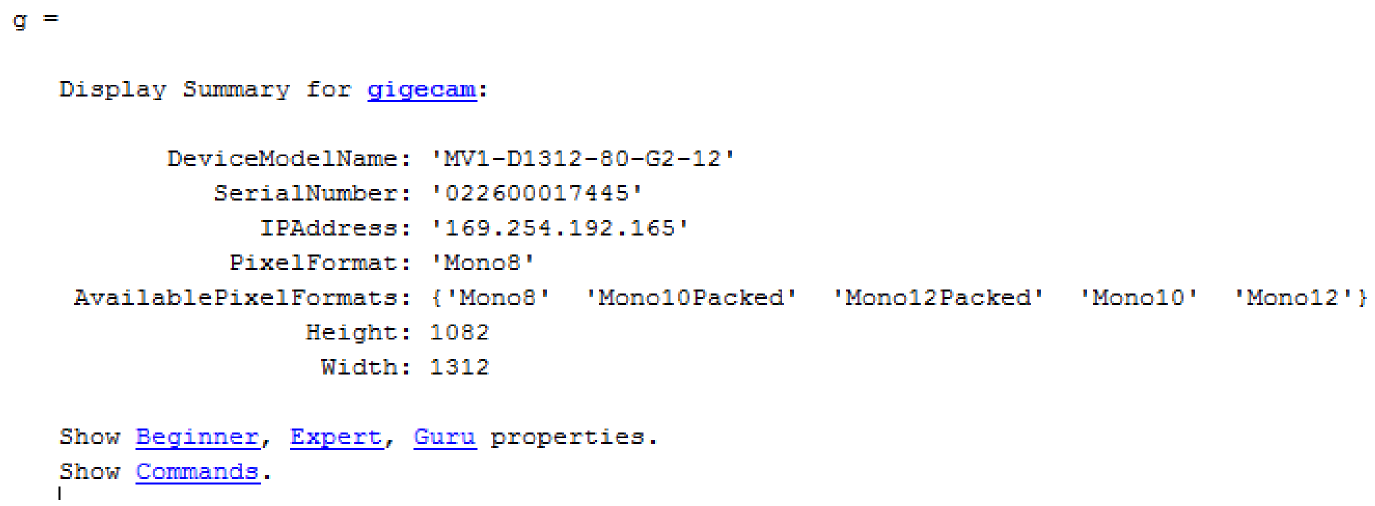
Use the `gigecamlist` function to ensure that MATLAB is discovering your camera.

```
gigecamlist
```

```
ans =

    Model                 Manufacturer           IPAddress        SerialNumber
  ____________________    ___________________    _______________    ______________

  'MV1-D1312-80-G2-12'    'Photonofocus AG'     '169.254.192.165'  '022600017445'
```

Create an object, g.

```
g = gigecam
```



```
g =

   Display Summary for gigecam:

          DeviceModelName: 'MV1-D1312-80-G2-12'
             SerialNumber: '022600017445'
                IPAddress: '169.254.192.165'
              PixelFormat: 'Mono8'
    AvailablePixelFormats: {'Mono8'  'Mono10Packed'  'Mono12Packed'  'Mono10'  'Mono12'}
                   Height: 1082
                    Width: 1312

   Show Beginner, Expert, Guru properties.
   Show Commands.
```

It creates the object and connects it to the Photonofocus AG camera.

Create a gigecam Object Using IP Address or Serial Number

Use the `gigecam` function with the IP address or serial number of the camera as the input argument to create the object and connect it to the camera with that address or number.

Use the `gigecamlist` function to ensure that MATLAB is discovering your cameras.

```
gigecamlist
```

```
ans =

    Model                   Manufacturer            IPAddress          SerialNumber
  ______________________   ____________________   _________________   ______________

  'MV1-D1312-80-G2-12'    'Photonofocus AG'      '169.254.192.165'   '022600017445'
  'mvBlueCOUGER-X120aG'   'MATRIX VISION GmbH'   '169.254.242.122'   'GX000818'
```

Create an object, g, using the IP address of the camera. You can also create the object in this same way using the serial number. You use the same syntax, but use a serial number instead of the IP address, also as a character vector.

```
g = gigecam('169.254.242.122')
```

```
g =

Display Summary for gigecam:

         DeviceModelName: 'mvBlueCOUGER-X120aG'
            SerialNumber: 'GX000818'
               IPAddress: '169.254.242.122'
             PixelFormat: 'Mono8'
```

```
   AvailablePixelFormats: {'Mono8' 'Mono12' 'Mono14' 'Mono16' 'Mono12Packed'
                           'BayerGR8' 'BayerGR10' 'BayerGR12' 'BayerGR16' 'BayerGR12Packed'
                           'YUV422Packed' 'YUV422_YUYVPacked' 'YUV444Packed'}
                  Height: 1082
                   Width: 1312

Show Beginner, Expert, Guru properties.
Show Commands.
```

It creates the object and connects it to the Matrix Vision camera with that IP address.

Create a gigecam Object Using Device Number as an Index

Use the `gigecam` function with an index as the input argument to create the object corresponding to that index and connect it to that camera. The index corresponds to the order of cameras in the table returned by `gigecamlist` when you have multiple cameras connected.

Use the `gigecamlist` function to ensure that MATLAB is discovering your cameras.

```
gigecamlist
```

```
ans = 

    Model                  Manufacturer           IPAddress         SerialNumber
  ____________________   ___________________   _______________   ______________

  'MV1-D1312-80-G2-12'   'Photonofocus AG'     '169.254.192.165'  '022600017445'
  'mvBlueCOUGER-X120aG'  'MATRIX VISION GmbH'  '169.254.242.122'  'GX000818'
```

Create an object, `g`, using the index number.

```
g = gigecam(2)
```

```
g = 

Display Summary for gigecam:

         DeviceModelName: 'mvBlueCOUGER-X120aG'
            SerialNumber: 'GX000818'
               IPAddress: '169.254.242.122'
             PixelFormat: 'Mono8'
   AvailablePixelFormats: {'Mono8' 'Mono12' 'Mono14' 'Mono16' 'Mono12Packed'
                           'BayerGR8' 'BayerGR10' 'BayerGR12' 'BayerGR16' 'BayerGR12Packed'
                           'YUV422Packed' 'YUV422_YUYVPacked' 'YUV444Packed'}
                  Height: 1082
                   Width: 1312

Show Beginner, Expert, Guru properties.
Show Commands.
```

It creates the object and connects it to the Matrix Vision camera with that index number, in this case, the second one displayed by `gigecamlist`. If you only have one camera, you do not need to use the index.

Input Arguments

`IPAddress` — IP address of your camera
character vector

IP address of your camera, specified as a character vector. This argument creates a `gigecam` object `g` where `IPAddress` is a character vector value that identifies a particular camera by its IP address. When you use the `gigecam` function with the IP address of the camera as the input argument, it creates the object and connects it to the camera with that address. You can see the IP address for your camera in the list returned by the `gigecamlist` function.

Example: `g = gigecam('169.254.192.165')`

Data Types: `char` | `string`

`devicenumber` — Device number of your camera
numeric scalar

Device number of your camera, specified as a numeric scalar. This number identifies a particular camera by its index order. It creates the object corresponding to that index and connects it to that camera. The index corresponds to the order of cameras in the table returned by `gigecamlist` when you have multiple cameras connected.

Example: `g = gigecam(2)`

Data Types: `double`

`serialnumber` — Serial number of your camera
character vector

Serial number of your camera, specified as a character vector. This argument creates a `gigecam` object `g` where `serialnumber` is a character vector value that identifies a particular camera by its serial number. When you use the `gigecam` function with the serial number of the camera as the input argument, it creates the object and connects it to the camera with that number. You can see the serial number for your camera in the list returned by the `gigecamlist` function.

Example: `g = gigecam('022600017445')`

Data Types: `char` | `string`

Tips

- When the `gigecam` object is created, it connects to the camera and establishes exclusive access. You can then preview the data and acquire images using the `snapshot` function.
- You cannot create more than one object connected to the same device, and trying to do that generates an error.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Version History

Introduced in R2014b

See Also

`gigecamlist` | `snapshot` | `commands` | `executeCommand`

gigecamlist

List of GigE Vision cameras connected to your system

Syntax

`gigecamlist`

Description

`gigecamlist` returns a list of available GigE Vision Compliant cameras connected to your system, with model, manufacturer, IP address, and serial number. If the camera has a user-defined name, that name is displayed. If you plug in different cameras during the same MATLAB session, then the `gigecamlist` function returns an updated list of cameras.

Examples

Display List of GigE Vision Compliant Cameras

The output of `gigecamlist` shows any GigE Vision cameras connected to your system.

```
gigecamlist
```

```
ans = 

    Model                Manufacturer           IPAddress        SerialNumber 
  ____________________   ___________________   ______________   ______________

  'MV1-D1312-80-G2-12'   'Photonofocus AG'     '169.254.192.165'  '022600017445'
  'mvBlueCOUGER-X120aG'  'MATRIX VISION GmbH'  '169.254.242.122'  'GX000818'
```

Version History

Introduced in R2014b

See Also

`gigecam` | `snapshot` | `commands` | `executeCommand`

imaqfind

Find image acquisition objects

Syntax

```
imaqfind
out = imaqfind
out = imaqfind(PropertyName, Value, PropertyName2, Value2,...)
out = imaqfind(S)
out = imaqfind(obj, PropertyName, Value, PropertyName2, Value2,...)
```

Description

`imaqfind` returns an array containing all the video input objects that exist in memory. If only a single video input object exists in memory, `imaqfind` displays a detailed summary of that object.

`out = imaqfind` returns an array, `out`, of all the video input objects that exist in memory.

`out = imaqfind(`*`PropertyName`*`, Value, `*`PropertyName2`*`, Value2,...)` returns a cell array, `out`, of image acquisition objects whose property names and property values match those passed as arguments. You can specify the property name/property value pairs in a cell array. You can use a mixture of character vectors, structures, and cell arrays. Use the `get` function to determine the list of properties supported by an image acquisition object.

`out = imaqfind(S)` returns a cell array, `out`, of image acquisition objects whose property values match those defined in the structure `S`. The field names of `S` are image acquisition object property names and the field values are the requested property values.

`out = imaqfind(obj, `*`PropertyName`*`, Value, `*`PropertyName2`*`, Value2,...)` restricts the search for matching parameter/value pairs to the image acquisition objects listed in `obj`. `obj` can be an array of image acquisition objects.

Note When searching for properties with specific values, `imaqfind` performs case-sensitive searches. For example, if the value of an object's `Name` property is `'MyObject'`, `imaqfind` does not find a match if you specify `'myobject'`. Note, however, that searches for properties that have an enumerated list of possible values are not case sensitive. For example, `imaqfind` will find an object with a `Running` property value of `'Off'` or `'off'`. Use the `get` function to determine the exact spelling of a property value.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Examples

To illustrate various `imaqfind` syntaxes, first create two video input objects.

```
obj1 = videoinput('matrox',1,'M_RS170','Tag','FrameGrabber');
obj2 = videoinput('winvideo',1,'RGB24_320x240','Tag','Webcam');
```

Now use `imaqfind` to find these objects by `type` and `tag`.

```
out1 = imaqfind('Type', 'videoinput')
out2 = imaqfind('Tag', 'FrameGrabber')
out3 = imaqfind({'Type', 'Tag'}, {'videoinput', 'Webcam'})
```

Version History

Introduced before R2006a

See Also

`get` | `videoinput`

imaqhelp

Image acquisition object function and property help

Syntax

```
imaqhelp
imaqhelp(Name)
imaqhelp(obj)
imaqhelp(obj,Name)
out = imaqhelp(...)
```

Description

`imaqhelp` provides a complete listing of image acquisition object functions.

`imaqhelp(`*`Name`*`)` provides online help for the function or property specified by the character vector *`Name`*.

`imaqhelp(obj)` displays a listing of functions and properties for the image acquisition object `obj` along with the online help for the object's constructor. `obj` must be a 1-by-1 image acquisition object.

`imaqhelp(obj,`*`Name`*`)` displays the help for the function or property specified by the character vector *`Name`* for the image acquisition object `obj`.

If *`Name`* is a device-specific property name, `obj` must be provided.

`out = imaqhelp(...)` returns the help text in character vector `out`.

When property help is displayed, the names in the “See Also” section that contain all uppercase letters are function names. The names that contain a mixture of upper- and lowercase letters are property names.

When function help is displayed, the “See Also” section contains only function names.

Examples

Getting general function and property help.

```
imaqhelp('videoinput')
out = imaqhelp('videoinput');
imaqhelp getsnapshot
imaqhelp LoggingMode
```

Getting property help with device-specific information.

```
vid = videoinput('dt', 1);
src = getselectedsource(vid);
imaqhelp(vid, 'TriggerType')
imaqhelp(src, 'FrameRate')
```

Version History

Introduced before R2006a

See Also

`propinfo`

imaqhwinfo

Information about available image acquisition hardware

Syntax

```
out = imaqhwinfo
out = imaqhwinfo(adaptorname)
out = imaqhwinfo(adaptorname,field)
out = imaqhwinfo(adaptorname, deviceID)
out = imaqhwinfo(obj)
out = imaqhwinfo(obj,field)
```

Description

`out = imaqhwinfo` returns `out`, a structure that contains information about the image acquisition adaptors available on the system. An adaptor is the interface between MATLAB and the image acquisition devices connected to the system. The adaptor's main purpose is to pass information between MATLAB and an image acquisition device via its driver.

`out = imaqhwinfo(`*`adaptorname`*`)` returns `out`, a structure that contains information about the adaptor specified by the character vector *`adaptorname`*. The information returned includes adaptor version and available hardware for the specified adaptor. To get a list of valid adaptor names, use the `imaqhwinfo` syntax.

`out = imaqhwinfo(`*`adaptorname`*`,`*`field`*`)` returns the value of the field specified by the character vector *`field`* for the adaptor specified by the character vector *`adaptorname`*. The argument can be a single character vector or a cell array of character vectors. If `field` is a cell array, `out` is a 1-by-`n` cell array where `n` is the length of *`field`*. To get a list of valid field names, use the `imaqhwinfo('adaptorname')` syntax.

`out = imaqhwinfo(`*`adaptorname`*`, deviceID)` returns `out`, a structure containing information about the device specified by the numeric device ID `deviceID`. The `deviceID` can be a scalar or a vector. If `deviceID` is a vector, `out` is a 1-by-`n` structure array where `n` is the length of `deviceID`.

`out = imaqhwinfo(obj)` returns `out`, a structure that contains information about the specified image acquisition object `obj`. The information returned includes the adaptor name, device name, video resolution, native data type, and device driver name and version. If `obj` is an array of device objects, then `out` is a 1-by-`n` cell array of structures where `n` is the length of `obj`.

`out = imaqhwinfo(obj,`*`field`*`)` returns the information in the field specified by `field` for the device object `obj`. `field` can be a single field name or a cell array of field names. `out` is an `m`-by-`n` cell array where `m` is the length of `obj` and `n` is the length of *`field`*. You can return a list of valid field names with the `imaqhwinfo(obj)` syntax.

Note After you call `imaqhwinfo` once, hardware information is cached by the toolbox. To force the toolbox to search for new hardware that might have been installed while MATLAB was running, use `imaqreset`.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Examples

This example returns information about all the adaptors available on the system.

```
imaqhwinfo

ans =

InstalledAdaptors: {'matrox'  'winvideo'}
        MATLABVersion: '7.4 (R2007a)'
          ToolboxName: 'Image Acquisition Toolbox'
       ToolboxVersion: '2.1 (R2007a)'
```

This example returns information about all the devices accessible through a particular adaptor.

```
info = imaqhwinfo('winvideo')
info =

       AdaptorDllName: [1x73 char]
    AdaptorDllVersion: '2.1 (R2007a)'
          AdaptorName: 'winvideo'
            DeviceIDs: {[1]}
           DeviceInfo: [1x1 struct]
```

This example returns information about a specific device accessible through a particular adaptor. You identify the device by its device ID.

```
dev_info = imaqhwinfo('winvideo', 1)

dev_info =

          DefaultFormat: 'RGB555_128x96'
    DeviceFileSupported: 0
             DeviceName: 'IBM PC Camera'
               DeviceID: 1
  VideoInputConstructor: 'videoinput('winvideo', 1)'
 VideoDeviceConstructor: 'imaq.VideoDevice('winvideo', 1)'
       SupportedFormats: {1x34 cell}
```

This example gets information about the device associated with a particular video input object.

```
obj = videoinput('winvideo', 1);

obj_info = imaqhwinfo(obj)

obj_info =

                AdaptorName: 'winvideo'
                 DeviceName: 'IBM PC Camera'
                  MaxHeight: 96
                   MaxWidth: 128
             NativeDataType: 'uint8'
```

```
               TotalSources: 1
    VendorDriverDescription: 'Windows WDM Compatible Driver'
        VendorDriverVersion: 'DirectX 9.0'
```

This example returns the value of a particular field in the device information associated with a particular video input object.

```
field_info = imaqhwinfo(vid,'adaptorname')
field_info =

winvideo
```

Version History

Introduced before R2006a

See Also

`imaqhelp` | `imaqreset`

imaqmontage

Sequence of image frames as montage

Syntax

```
imaqmontage(frames)
imaqmontage(obj)
imaqmontage(...,CLIM)
imaqmontage(..., 'CLim', CLIM, 'Parent', PARENT)
h = imaqmontage(...)
```

Description

`imaqmontage(frames)` displays a montage of image frames in a MATLAB figure window using the `imagesc` function.

`frames` can be any data set returned by `getdata`, `peekdata`, or `getsnapshot`.

`imaqmontage(obj)` calls the `getsnapshot` function on video input object `obj` and displays a single image frame in a MATLAB figure window using the `imagesc` function. `obj` must be a 1-by-1 video input object.

`imaqmontage(...,CLIM)` displays a montage of image frames, where `CLIM` is a two-element vector, `[CLOW CHIGH]`, specifying the image scaling. Use `CLIM` to specify a scaling value when overscaling the image data is a risk, for example, when you are working with devices that provide data in a 12-bit format.

`imaqmontage(..., 'CLim', CLIM, 'Parent', PARENT)` where `CLIM` is as noted previously, and `PARENT` is a valid AXES object that allows you to specify where the montage is displayed. One or both property/value pairs can be specified. See the example below.

`h = imaqmontage(...)` returns a handle to an image object.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Examples

Construct a video input object associated with a Matrox device at ID 1.

```
obj = videoinput('matrox', 1);
```

Initiate an acquisition and access the logged data.

```
start(obj);
data = getdata(obj);
```

Create an axes object.

```
a = axes;
```

Display each image frame acquired on axes `a`.

```
imaqmontage(data, 'Parent', a);
```

Remove the video input object from memory.

```
delete(obj);
```

Version History

Introduced before R2006a

See Also

`getdata` | `getsnapshot` | `imaqhelp` | `peekdata`

imaqregister

Register third-party custom adaptor

Syntax

```
adaptors = imaqregister
adaptors = imaqregister(adaptorpath)
adaptors = imaqregister(adaptorpath,action)
```

Description

`adaptors = imaqregister` returns a list of registered third-party adaptors with their full paths. If there are no registered adaptors, `imaqregister` returns an empty cell array.

Note The `imaqhwinfo` and `videoinput` functions use the adaptor base name, not the full path. For example, if the adaptor full path is `"c:\adaptor.dll"` (Windows), `"/local/adaptors/adaptor.so"` (Linux), or `"/local/adaptors/adaptor.dylib"` (macOS), the adaptor base name is `"adaptor"`.

`adaptors = imaqregister(adaptorpath)` registers the specified third-party adaptor library and returns a list of all registered adaptors.

Registering an adaptor informs Image Acquisition Toolbox of the location of a third-party adaptor library. If you query the system for available image acquisition hardware by using `imaqhwinfo`, the toolbox makes available any previously registered adaptor libraries. The `imaqregister` function saves the name of the registered adaptor in the MATLAB preferences directory so that the location persists across MATLAB sessions. Because `imaqhwinfo` caches the list of available adaptors, you might need to call `imaqreset` after calling `imaqregister` to make the newly registered adaptor available.

Note The adaptor shared library (a DLL on Windows) is not the same as the driver shared library supplied by a hardware vendor (also a DLL on Windows). The adaptor shared library is specific to Image Acquisition Toolbox and is specified as such by the hardware vendor.

`adaptors = imaqregister(adaptorpath,action)` adds or removes the third-party adaptor from the list of available adaptors, as specified by `action`.

Examples

Register a Third-Party Adaptor

Register a third-party adaptor in Image Acquisition Toolbox on a Windows system and preview its data. This example uses the demo adaptor included with Image Acquisition Toolbox, but you can follow these steps to register any custom third-party adaptor. To view the demo adaptor source files, navigate to the directory on your system. The exact file path might differ on your computer.

```
cd('C:\Program Files\MATLAB\R2019b\toolbox\imaq\imaqadaptors\kit\demo\')
```

Use `imaqregister` to inform Image Acquisition Toolbox of your third-party adaptor. For this example, consider a prebuilt version of the demo adaptor for Windows in the `\toolbox\imaq\imaqadaptors\kit\demo\win64` folder.

```
imaqregister('C:\Program Files\MATLAB\R2019b\toolbox\imaq\imaqadaptors\kit\demo\win64\mwdemoimaq
```

```
ans = 1×1 cell array
    {'C:\Program Files\MATLAB\R2019b\toolbox\imaq\imaqadaptors\kit\demo\win64\mwdemoimaq.dll'}
```

The toolbox caches adaptor information, so you must reload the adaptor libraries registered with the toolbox for your third-party adaptor to appear.

```
imaqreset
```

View a list of installed adaptors in the `InstalledAdaptors` field. The newly registered third-party adaptor appears as `mwdemoimaq`.

```
imaqhwinfo
```

```
ans = struct with fields:
    InstalledAdaptors: {'demo'  'gentl'  'gige'  'kinect'  'matrox'  'mwdemoimaq'  'spinnaker'
        MATLABVersion: '9.7 (R2019b)'
          ToolboxName: 'Image Acquisition Toolbox'
       ToolboxVersion: '6.1 (R2019b)'
```

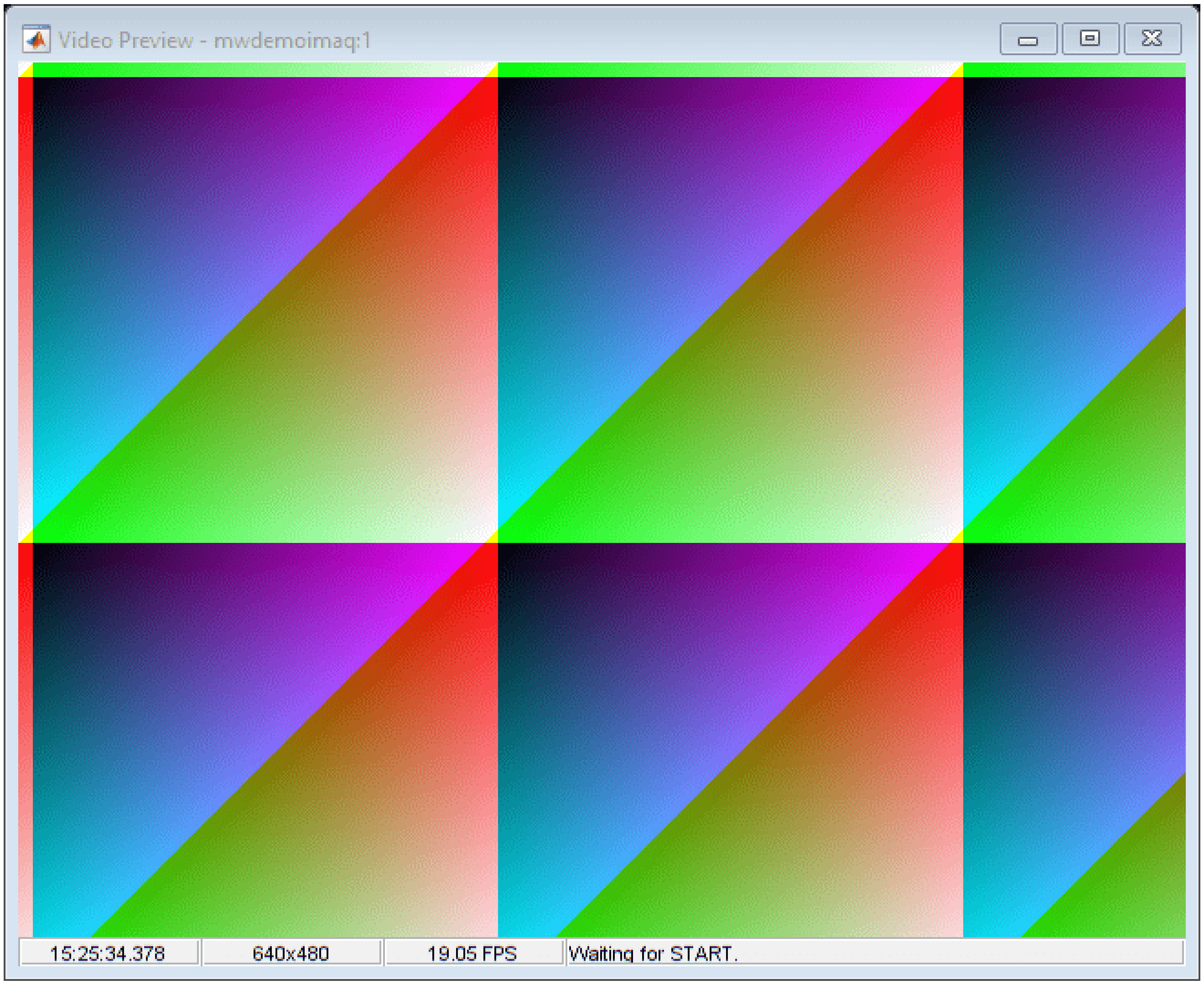
Create a video input object with this adaptor.

```
vid = videoinput('mwdemoimaq')
```

```
Summary of Video Input Object Using 'Color Device'.

   Acquisition Source(s):  input1 is available.

  Acquisition Parameters:  'input1' is the current selected source.
                           10 frames per trigger using the selected source.
                           'RGB_NTSC' video data to be logged upon START.
                           Grabbing first of every 1 frame(s).
                           Log data to 'memory' on trigger.

      Trigger Parameters:  1 'immediate' trigger(s) on START.

                  Status:  Waiting for START.
                           0 frames acquired since starting.
                           0 frames available for GETDATA.
```

Get a preview of the data returned by the adaptor.

```
preview(vid)
```



After you finish working with the preview, close the window.

```
closepreview(vid)
```

Input Arguments

adaptorpath — Adaptor path
character vector | string array

Adaptor path, specified as a character vector or string array. You must specify the full absolute path of the adaptor library file.

Example: `imaqregister('c:\temp\thirdpartyadaptor.dll')` registers the adaptor `thirdpartyadaptor`.

Data Types: `char` | `string`

action — Adaptor registration
`'register'` (default) | `'unregister'`

Adaptor registration, specified as `'register'` or `'unregister'`. Using `'register'` adds the third-party adaptor to the list of available adaptors. Using `'unregister'` removes the third-party adaptor from the list.

Example: `imaqregister('c:\temp\thirdpartyadaptor.dll','unregister')` removes the adaptor `thirdpartyadaptor`.

Data Types: `char` | `string`

Tips

- Follow these suggestions when you deploy a custom adaptor to a standalone application using MATLAB Compiler.
  - Call `imaqregister` in the MATLAB code that you are deploying. This ensures that the deployed application registers the custom adaptor library for the user running the application. For more information, see "Creating Custom Adaptors".
  - Package the custom adaptor library with the standalone application. To do this, add the adaptor DLL file in the **Files installed for your end user** section of the **Application Compiler** app. For more information about creating a standalone application, see "Create Standalone Application from MATLAB Function" (MATLAB Compiler).
  - The first time you run a deployed application that calls `imaqregister`, you might need to execute the application in **Run as administrator** mode.

Version History

Introduced before R2006a

See Also

`imaqhwinfo`

Topics

"Looking at the Demo Adaptor"

imaqreset

Disconnect and delete all image acquisition objects

Syntax

```
imaqreset
```

Description

`imaqreset` deletes any image acquisition objects that exist in memory and unloads all adaptors loaded by the toolbox. As a result, the image acquisition hardware is reset.

`imaqreset` is the image acquisition command that returns MATLAB to the known state of having no image acquisition objects and no loaded image acquisition adaptors.

You can use `imaqreset` to force the toolbox to search for new hardware that might have been installed while MATLAB was running.

Note that `imaqreset` should not be called from any of the callbacks of a videoinput object, such as the `StartFcn` or `FramesAcquiredFcn`.

Version History

Introduced before R2006a

See Also

`delete` | `videoinput`

imaq.VideoDevice

Acquire one frame at a time from video device

Syntax

```
obj = imaq.VideoDevice
obj = imaq.VideoDevice(adaptorname)
obj = imaq.VideoDevice(adaptorname, deviceid)
obj = imaq.VideoDevice(adaptorname, deviceid, format)
obj = imaq.VideoDevice(adaptorname, deviceid, format, P1, V1, ...)
frame = step(obj)
[frame metadata] = step(obj)
```

Description

The VideoDevice System object allows single-frame image acquisition and code generation from MATLAB. You use the `imaq.VideoDevice` function to create the System object. It supports the same adaptors and hardware that the `videoinput` object supports; however, it has different functions and properties associated with it. For example, the System object uses the `step` function to acquire single frames.

`obj = imaq.VideoDevice` creates a VideoDevice System object, `obj`, that acquires images from a specified image acquisition device. When you specify no parameters, by default, it selects the first available device for the first adaptor returned by `imaqhwinfo`.

`obj = imaq.VideoDevice(adaptorname)` creates a VideoDevice System object, `obj`, using the first device of the specified `adaptorname`. `adaptorname` is a character vector that specifies the name of the adaptor used to communicate with the device. Use the `imaqhwinfo` function to determine the adaptors available on your system.

`obj = imaq.VideoDevice(adaptorname, deviceid)` creates a VideoDevice System object, `obj`, with the default format for specified `adaptorname` and `deviceid`. `deviceid` is a numeric scalar value that identifies a particular device available through the specified `adaptorname`. Use the `imaqhwinfo(adaptorname)` syntax to determine the devices available and corresponding values for `deviceid`.

`obj = imaq.VideoDevice(adaptorname, deviceid, format)` creates a VideoDevice System object, `obj`, where `format` is a character vector that specifies a particular video format supported by the device or a device configuration file (also known as a camera file).

`obj = imaq.VideoDevice(adaptorname, deviceid, format, P1, V1, ...)` Creates a VideoDevice System object, `obj`, with the specified property values. If an invalid property name or property value is specified, the object is not created.

Specifying properties at the time of object creation is optional. They can also be specified after the object is created. See the table below for a list of applicable properties.

`frame = step(obj)` acquires a single frame from the VideoDevice System object, `obj`.

`[frame metadata] = step(obj)` acquires a single image frame from the VideoDevice System object, `obj`, plus metadata from the Kinect for Windows Depth sensor. You can return Kinect for

Windows skeleton data using the VideoDevice System object on the Kinect Depth sensor. For information on how to do this, see "Kinect for Windows Metadata" on page 14-5.

Properties

You can specify properties at the time of object creation, or they can be specified and changed after the object is created. Properties that can be used with the VideoDevice System object include:

Property	Description
`Device`	Device from which to acquire images. Specify the image acquisition device to use to acquire a frame. It consists of the device name, adaptor, and device ID. The default device is the first device returned by `imaqhwinfo`.
`VideoFormat`	Video format to be used by the image acquisition device. Specify the video format to use while acquiring the frame. The default value of `VideoFormat` is the default format returned by `imaqhwinfo` for the selected device. To specify a Video Format using a device file, set the `VideoFormat` property to `'From device file'` This option exists only if your device supports device configuration files.
`DeviceFile`	Name of file specifying video format. This property is only visible when `VideoFormat` is set to `'From device file'`.
`DeviceProperties`	Object containing properties specific to the image acquisition device.
`ROI`	Region-of-interest for acquisition. This is set to the default ROI value for the specified device, which is the maximum resolution possible for the specified format. You can change the value to change the size of the captured image. The format is 1-based, that is, it is specified in pixels in a 1-by-4 element vector `[x y width height]`. Note that this differs from the `videoinput` object and the From Video Device block, which are 0-based.
`HardwareTriggering`	Turn hardware triggering on/off. Set this property to `'on'` to enable hardware triggering to acquire images. The property is visible only when the device supports hardware triggering.
`TriggerConfiguration`	Specifies the trigger source and trigger condition before acquisition. The triggering condition must be met via the trigger source before a frame is acquired. This property is visible only when `HardwareTriggering` is set to `'on'`.
`ReturnedColorSpace`	Specify the color space of the returned image. The default value of the property depends on the device and the video format selected. Possible values are {`rgb`|`grayscale`|`YCbCr`} when the default returned color space for the device is not `grayscale`. Possible values are {`rgb`|`grayscale`|`YCbCr`|`bayer`} when the default returned color space for the device is `grayscale`

Property	Description
`BayerSensorAlignment`	Character vector indicating the 2x2 sensor alignment. Specifies Bayer patterns returned by hardware. Specify the sensor alignment for Bayer demosaicing. The default value of this property is `'grbg'`. Possible values are {`grbg`|`gbrg`|`rggb`|`bggr`}. Visible only if `ReturnedColorSpace` is set to `'bayer'`.
`ReturnedDataType`	The returned data type of the acquired frame. The default `ReturnedDataType` is `single`.
`ReadAllFrames`	Specify whether to read one image frame or all available frames. Set to `'on'` to capture all available image frames. When set to the default of `'off'`, the system object takes a snapshot of one frame, which is the equivalent of the `getsnapshot` function in the toolbox. When the option is on, all available image frames are captured, which is the equivalent of the `getdata` function in the toolbox.

The setting of properties for the System object supports tab completion for enumerated properties while coding in MATLAB. Using the tab completion is an easy way to see available property values. After you type the property name, type a comma, then a space, then the first quote mark for the value, then hit tab to see the possible values.

You can also use the `set` function with the object name and property name to get a list of available values for that property. For example:

```
set(obj, 'ReturnedColorSpace')
```

gets the list of available color space settings for the VideoDevice System object, `obj`.

Note that once you have done a step, in order to change a property or set a new one, you need to release the object using the `release` function, before setting the new property.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Functions

You can use these functions with the VideoDevice System object.

Function	Purpose

Function	Purpose
`step`	Acquire a single frame from the image acquisition device. `frame = step(obj);` acquires a single frame from the VideoDevice System object, `obj`. Note that the first time you call step, it acquires exclusive use of the hardware and will start streaming data.
`release`	Release VideoDevice resources and allow property value changes. `release(obj)` releases system resources (such as memory, file handles, or hardware connections) of System object, `obj`, and allows all its properties and input characteristics to be changed.
`isLocked`	Returns a value that indicates if the VideoDevice resource is locked. (Use `release` to unlock.) `L = isLocked(obj)` returns a logical value, `L`, which indicates whether properties are locked for the System object, obj. The object performs an internal initialization the first time the `step` function is executed. This initialization locks properties and input specifications. Once this occurs, the `isLocked` function returns a value of `true`.
`preview`	Activate a live image preview window. `preview(obj)` creates a Video Preview window that displays live video data for the VideoDevice System object, `obj`. The Video Preview window displays the video data at 100% magnification. The size of the preview image is determined by the value of the VideoDevice System object `ROI` property. If not specified, it uses the default resolution for the device.
`closepreview`	Close live image preview window. `closepreview(obj)` closes the live preview window for VideoDevice System object, `obj`.
`imaqhwinfo`	Returns information about the object. `imaqhwinfo(obj)` displays information about the VideoDevice System object, `obj`.

Examples

Construct a VideoDevice System object associated with the Winvideo adaptor with device ID of 1.

```
vidobj = imaq.VideoDevice('winvideo', 1);
```

Set an object-level property, such as `ReturnedColorSpace`. The syntax for an object-level property uses the object name, property name, and property value.

```
vidobj.ReturnedColorSpace = 'grayscale';
```

Set a device-specific property, such as `Brightness`. The syntax for a device-specific property uses the `DeviceProperties` object, the property name, and property value.

```
vidobj.DeviceProperties.Brightness = 150;
```

Preview the image.

```
preview(vidobj)
```

Acquire a single frame.

```
frame = step(vidobj);
```

Display the acquired frame.

```
imshow(frame)
```

Release the hardware resource.

```
release(vidobj);
```

Clear the VideoDevice System object.

```
clear vidobj;
```

Version History

Introduced in R2012a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “Code Generation with VideoDevice System Object” on page 14-11.

islogging

Determine whether video input object is logging

Syntax

```
bool = islogging(obj)
```

Description

`bool = islogging(obj)` returns `true` if the video input object `obj` is logging data, otherwise `false`. A video input object is logging if the value of its `Logging` property is set to `'on'`.

If `obj` is an array of video input objects, `bool` is a logical array where each element in `bool` represents the corresponding element in `obj`. If an object in `obj` is logging data, `islogging` sets the corresponding element in `bool` to `true`, otherwise `false`. If any of the video input objects in `obj` is invalid, `islogging` returns an error.

Examples

Create a video input object.

```
vid = videoinput('winvideo');
```

To put the video input object in a logging state, start acquiring data. The example acquires 50 frames to increase the amount of time that the object remains in logging state.

```
vid.FramesPerTrigger = 50
start(vid)
```

When the call to the `start` function returns, and the object is still acquiring data, use `islogging` to check the state of the object.

```
bool = islogging(vid)
bool =

   1
```

Create a second video input object.

```
vid2 = videoinput('winvideo');
```

Start one of the video input objects again, such as `vid`, and use `islogging` to determine which of the two objects is logging.

```
start(vid)
bool = islogging([vid vid2])

bool =

     1     0
```

Version History

Introduced before R2006a

See Also

`isrunning` | `isvalid` | `videoinput`

isrunning

Determine whether video input object is running

Syntax

```
bool = isrunning(obj)
```

Description

`bool = isrunning(obj)` returns `true` if the video input object `obj` is running, otherwise `false`. A video input object is running if the value of its `Running` property is set to `'on'`.

If `obj` is an array of video input objects, `bool` is a logical array where each element in `bool` represents the corresponding element in `obj`. If an object in `obj` is running, the `isrunning` function sets the corresponding element in `bool` to `true`, otherwise `false`. If the video input objects in `obj` is invalid, `isrunning` returns an error.

Examples

Create a video input object, configure a manual trigger, and then start the object. This puts the object in running state.

```
vid = videoinput('winvideo');
triggerconfig(vid,'manual')
start(vid)
```

Use `isrunning` to check the state of the object.

```
bool = isrunning(vid)
bool =
   1
```

Create a second video input object.

```
vid2 = videoinput('winvideo');
```

Use `isrunning` to determine which of the two objects is running.

```
bool = isrunning([vid vid2])
bool =
     1     0
```

Version History

Introduced before R2006a

See Also

`islogging` | `isvalid` | `start` | `stop` | `videoinput` | Running

isvalid

Determine whether image acquisition object is associated with image acquisition device

Syntax

```
bool = isvalid(obj)
```

Description

`bool = isvalid(obj)` returns `true` if the video input object `obj` is valid, otherwise `false`. An object is an invalid image acquisition object if it is no longer associated with any hardware; that is, the object was deleted using the `delete` function. If this is the case, `obj` should be cleared from the workspace.

If `obj` is an array of video input objects, `bool` is a logical array where each element in `bool` represents the corresponding element in `obj`. If an object in `obj` is valid, the `isvalid` function sets the corresponding element in `bool` to `true`, otherwise `false`.

Version History

Introduced before R2006a

See Also

`delete` | `imaqfind` | `videoinput`

load

Load image acquisition object into MATLAB workspace

Syntax

```
load filename
load filename obj1 obj2 ...
S = load(filename,obj1,obj2,...)
```

Description

`load filename` returns all variables from the MAT-file `filename` to the MATLAB workspace.

`load filename obj1 obj2 ...` returns the specified image acquisition objects (`obj1`, `obj2`, etc.) from the MAT-file specified by `filename` to the MATLAB workspace.

`S = load(filename,obj1,obj2,...)` returns the structure `S` with the specified image acquisition objects (`obj1`, `obj2`, etc.) from the MAT-file `filename`. The field names in `S` match the names of the image acquisition objects that were retrieved. If no objects are specified, then all variables existing in the MAT-file are loaded.

Values for read-only properties are restored to their default values when loaded. For example, the `Running` property is restored to `'off'`. Use `propinfo` to determine if a property is read only.

Examples

```
obj = videoinput('winvideo', 1);
obj.SelectedSourceName = 'input1'
save fname obj
load fname
load('fname', 'obj');
```

Version History

Introduced before R2006a

See Also

`imaqhelp` | `propinfo` | `save`

matroxcam

Create `matroxcam` object to acquire images from Matrox frame grabbers

Syntax

```
m = matroxcam(devicenumber, 'DCFfilename')
```

Description

`m = matroxcam(devicenumber, 'DCFfilename')` creates a `matroxcam` object `m`, where `devicenumber` is a numeric scalar value that identifies a particular device by its index number and `DCFfilename` is the name and fully qualified path of your Matrox DCF file, and connects it to that frame grabber.

Examples

Create a matroxcam object

Use the `matroxcam` function with an index as the first input argument to create the object corresponding to that index and connect it to that frame grabber. The index corresponds to the order of boards in the cell array returned by `matroxlist` when you have multiple frame grabbers connected. If you only have one frame grabber, you must use a `1` as the input argument. The second argument must be the name and path of your DCF file, entered as a character vector.

Use the `matroxlist` function to ensure that MATLAB is discovering your frame grabbers.

```
matroxlist
```

```
ans =

   Solios XCL (digitizer 0)
   Solios XCL (digitizer 1)
   VIO (digitizer 0)
```

Create an object, `m`, using the index number. In this example, for the Solios XCL at digitizer 1, use a `2` as the index number, since it is the second device on the list. The second argument must be the name of your DCF file, entered as a character vector. It must contain the fully qualified path to the file as well. In this example, the DCF file is named `mycam.dcf`.

```
m = matroxcam(2, 'C:\Drivers\Solios\dcf\XCL\Basler\A404K\mycam.dcf')
```

```
m =

Display Summary for matroxcam:

         DeviceName: 'Solios XCL (digitizer 1)'
            DCFName: 'C:\Drivers\Solios\dcf\XCL\Basler\A404K\mycam.dcf'
```

```
    FrameResolution: '1300 x 1080'
            Timeout: 10
```

Input Arguments

`devicenumber` — Device number of your frame grabber
numeric scalar

Device number of your frame grabber, specified as a numeric scalar. This number identifies a particular board by its index order. It creates the object corresponding to that index and connects it to that frame grabber. The index corresponds to the order of frame grabbers in the table returned by `matroxlist` when you have multiple boards connected.

If you only have one frame grabber, you must use a `1` as the input argument.

Example: `m = matroxcam(2, 'C:\Drivers\Solios\dcf\XCL\Basler\A404K\mycam.dcf')`

Data Types: `double`

`DCFfilename` — Name of your DCF file
character vector

Name of your DCF file, specified as a character vector. The Digitizer Configuration File (DCF) is used to set acquisition properties and is configured using the Matrox Intellicam software. The DCF file contains properties relating to exposure signal, grab mode, sync signal, camera, video signal, video timing, and pixel clock. Once you have configured these properties in your DCF file, you create the `matroxcam` object using that file as an input argument. It must contain the fully qualified path to the file as well. In this example, the DCF file is named `mycam.dcf`.

Example: `m = matroxcam(2, 'C:\Drivers\Solios\dcf\XCL\Basler\A404K\mycam.dcf')`

Data Types: `char` | `string`

Version History

Introduced in R2014b

See Also

`matroxlist` | `snapshot` | `preview` | `closepreview`

Topics
“Connect to Matrox Frame Grabbers” on page 13-3
“Set Properties for Matrox Acquisition” on page 13-4
“Acquire Images from Matrox Frame Grabbers” on page 13-6
“Matrox Acquisition – matroxcam Object vs videoinput Object” on page 13-2

matroxlist

List of Matrox frame grabbers connected to your system

Syntax

`matroxlist`

Description

`matroxlist` returns a list of available Matrox frame grabbers connected to your system, with model name and digitizer number.

If no boards are detected, it returns an empty cell array.

Examples

Display List of Matrox Frame Grabbers

The output of `matroxlist` shows any Matrox frame grabbers connected to your system.

```
matroxlist
```

```
ans =

   Solios XCL (digitizer 0)
   Solios XCL (digitizer 1)
   VIO (digitizer 0)
```

Version History

Introduced in R2014b

See Also

`matroxcam` | `snapshot` | `preview` | `closepreview`

Topics

"Connect to Matrox Frame Grabbers" on page 13-3
"Set Properties for Matrox Acquisition" on page 13-4
"Acquire Images from Matrox Frame Grabbers" on page 13-6
"Matrox Acquisition – matroxcam Object vs videoinput Object" on page 13-2

obj2mfile

Convert video input objects to MATLAB code

Syntax

```
obj2mfile(obj,filename)
obj2mfile(obj,filename,syntax)
obj2mfile(obj,filename,syntax,mode)
obj2mfile(obj,filename,syntax,mode,reuse)
```

Description

`obj2mfile(obj,filename)` converts the video input object `obj` into an M-file with the name specified by `filename`. The M-file contains the MATLAB code required to create the object and set its properties. `obj` can be a single video input object or an array of objects.

The `obj2mfile` function simplifies the process of restoring an object with specific property settings and can be used to create video input objects. `obj2mfile` also creates and configures the video source object associated with the video input object.

If `filename` does not specify an extension or if it has an extension other than the MATLAB M-file extension (`.m`), `obj2mfile` appends `.m` to the end of `filename`. To recreate `obj`, execute the M-file by calling `filename`.

If the `UserData` property of the object is set, or if any of the callback properties is set to a cell array or to a function handle, `obj2mfile` writes the data stored in those properties to a MAT-file. `obj2mfile` gives the MAT-file the same name as the M-file, but uses the `.mat` filename extension. `obj2mfile` creates the MAT-file in the same directory as the M-file.

Note `obj2mfile` does not restore the values of read-only properties. For example, if an object is saved with a `Logging` property set to `'on'`, the object is recreated with a `Logging` property set to `'off'` (the default value). Use the `propinfo` function to determine if a property is read only.

`obj2mfile(obj,filename,`*`syntax`*`)` converts `obj` to the equivalent MATLAB code where *`syntax`* specifies how `obj2mfile` assigns values to properties of the object. `syntax` can be either of the following character vectors. The default value is enclosed in braces ({}).

Character Vector	Description
`{'set'}`	`obj2mfile` uses the `set` function when specifying property values.
`'dot'`	`obj2mfile` uses subscripted assignment (dot notation) when specifying property values.

`obj2mfile(obj,filename,`*`syntax`*`,`*`mode`*`)` converts `obj` to the equivalent MATLAB code where `mode` specifies which properties are configured. `mode` can be either of the following character vectors. The default value is enclosed in braces ({}).

Character Vector	Description
`{'modified'}`	Configure writable properties that are not set to their default values.
`'all'`	Configure all writable properties. `obj2mfile` does not restore the values of read-only properties.

Note that `obj2mfile(obj,filename,`*`mode`*`)` is a valid syntax. If the *`syntax`* argument is not specified, `obj2mfile` uses the default value.

`obj2mfile(obj,filename,`*`syntax`*`,`*`mode`*`,`*`reuse`*`)` converts `obj` to the equivalent MATLAB code where *`reuse`* specifies whether `obj2mfile` searches for a reusable video input object or creates a new one. *`reuse`* can be either of the following character vectors. The default value is enclosed in braces ({}).

Character Vector	Description
`{'reuse'}`	`Find` and modify an existing object, if the existing object is associated with the same adaptor and the values of the `DeviceID`, `VideoFormat`, and `Tag` properties match the object being created. If no matching object can be found, `obj2mfile` creates a new object.
`'create'`	Create a new object regardless of whether there are reusable objects.

Note that `obj2mfile(obj,filename,`*`reuse`*`)` is a valid syntax. If the *`syntax`* and *`mode`* arguments are not specified, `obj2mfile` uses their default values.

Examples

Create a video input object.

```
 vidobj = videoinput('winvideo', 1, 'RGB24_640x480');
```

Configure several properties of the video input object.

```
vidobj.FramesPerTrigger = 100;
vidobj.FrameGrabInterval = 2;
vidobj.Tag = 'CAM1';
```

Retrieve the selected video source object associated with the video input object.

```
src = getselectedsource(vidobj);
```

Configure the properties of the video source object.

```
src.Contrast = 85;
src.Saturation = 125;
```

Save the video input object.

```
obj2mfile(vidobj, 'myvidobj.m', 'set', 'modified');
```

Delete the object and clear it from the workspace.

```
delete(vidobj);
clear vidobj;
```

Execute the M-file to recreate the object. Note that `obj2mfile` creates and configures the associated video source object as well.

```
vidObj = myvidobj;
```

Version History

Introduced before R2006a

See Also

`getselectedsource` | `imaqhelp` | `propinfo` | `set` | `videoinput`

peekdata

Most recently acquired image data

Syntax

```
data = peekdata(obj,frames)
```

Description

`data = peekdata(obj,frames)` returns `data` containing the latest number of frames specified by `frames`. If `frames` is greater than the number of frames currently acquired, all available frames are returned with a warning message stating that the requested number of frames was not available. `obj` must be a 1-by-1 video input object.

`data` is returned as an H-by-W-by-B-by-F matrix where

H	Image height, as specified in the object's `ROIPosition` property
W	Image width, as specified in the object's `ROIPosition` property
B	Number of color bands, as specified in the `NumberOfBands` property
F	Number of frames returned

`data` is returned to the MATLAB workspace in its native data type using the color space specified by the `ReturnedColorSpace` property.

You can use the MATLAB `image` or `imagesc` functions to view the returned data. Use `imaqmontage` to view multiple frames at once.

`peekdata` is a nonblocking function that immediately returns image frames and execution control to the MATLAB workspace. Not all requested data might be returned.

Note `peekdata` provides a look at the data; it does not remove data from the memory buffer. The object's `FramesAvailable` property value is not affected by the number of frames returned by `peekdata`.

The behavior of `peekdata` depends on the settings of the `Running` and the `Logging` properties.

Running	Logging	Object State	Result
On	Off	The object has been started but is waiting for a trigger. (`TriggerType` is set to `'manual'` or `'hardware'`). No data has been acquired so none is available.	`peekdata` returns a single frame of data and issues a warning, if you requested more than one frame.
On	On	The object has been started, a trigger has executed, and the object is actively acquiring data.	`peekdata` returns the *n* most recently acquired frames of data. The frames are not removed from the buffer.

Running	Logging	Object State	Result
Off	Off	The object has stopped running because it acquired the requested number of frames or you called the `stop` function.	`peekdata` can be called once to return the *n* most recently acquired frames of data, assuming `FramesAvailable` is greater than 0. Otherwise, `peekdata` returns an error. The frames returned are not removed from the memory buffer.

The number of frames available to `peekdata` is determined by recalling the last frame returned by a previous `peekdata` call, and the number of frames that were acquired since then.

`peekdata` can be used only after the `start` command is issued and while the object is running. `peekdata` can also be called once after `obj` has stopped running.

Note The `peekdata` function does not return any data while running if in disk logging mode.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Version History

Introduced before R2006a

See Also

`getdata` | `getsnapshot` | `imaqhelp` | `imaqmontage` | `propinfo` | `start`

preview

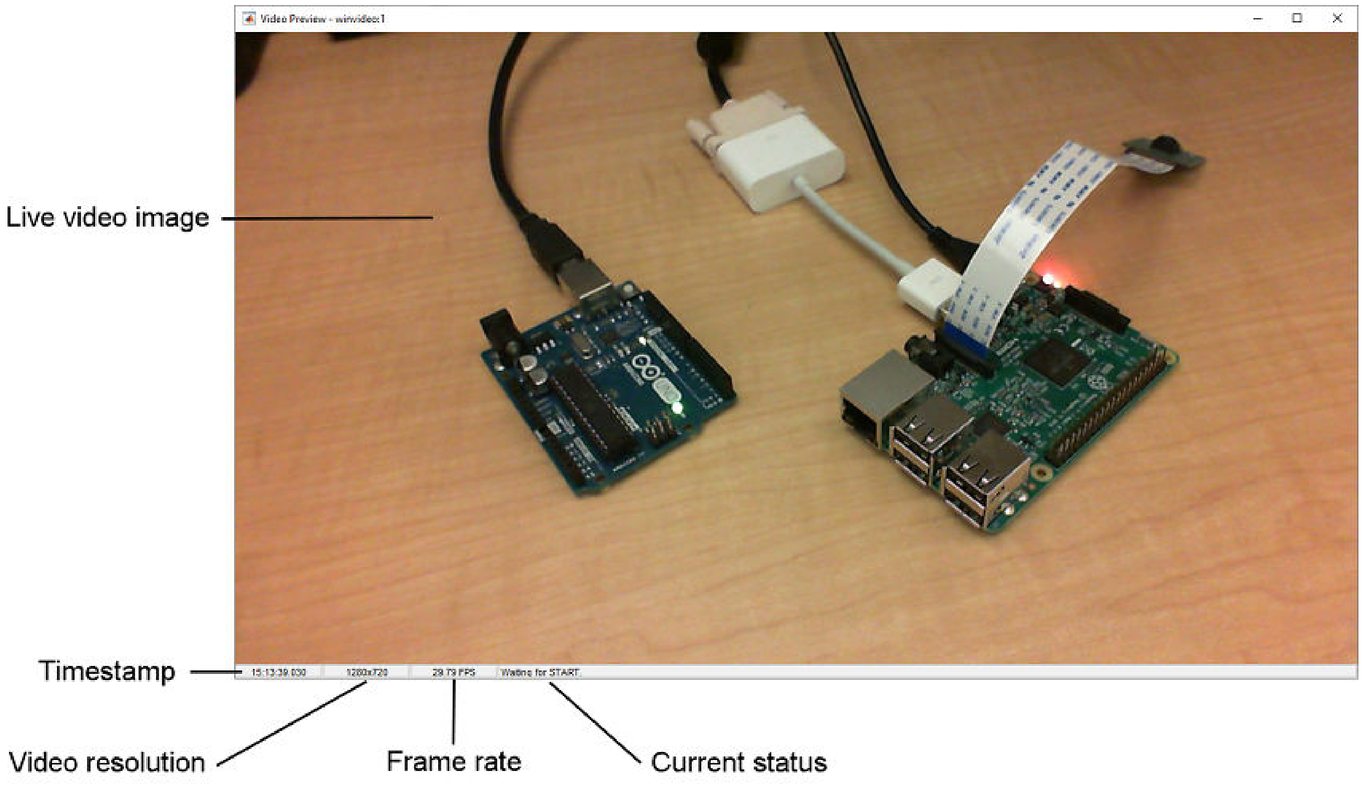
Preview of live video data

Syntax

```
preview(obj)
preview(obj,himage)
himage = preview(...)
```

Description

`preview(obj)` creates a Video Preview window that displays live video data for video input object `obj`. The window also displays the timestamp and video resolution of each frame, the current frame rate, and the current status of `obj`. The Video Preview window displays the video data at 100% magnification. The size of the preview image is determined by the value of the video input object `ROIPosition` property.



Components of a Video Preview Window

The Video Preview window remains active until it is either stopped using `stoppreview` or closed using `closepreview`. If you delete the object, by calling `delete(obj)`, the Video Preview window stops previewing and closes automatically.

`preview(obj,himage)` displays live video data for video input object `obj` in the image object specified by the handle `himage`. `preview` scales the image data to fill the entire area of the image object but does not modify the values of any image object properties. Use this syntax to preview video data in a custom GUI of your own design (see Examples).

`himage = preview(...)` returns `himage`, a handle to the image object containing the previewed data. To obtain a handle to the figure window containing the image object, use the `ancestor` function. For more information about using image objects, see `image`. See the Custom Update Function section for more information about the image object returned.

Notes

The behavior of the Video Preview window depends on the video input object's current state and trigger configuration.

Object State	Preview Window Behavior
`Running=off`	Displays a live view of the image being acquired from the device, for all trigger types. The image is updated to reflect changes made to configurations of object properties. (The `FrameGrabInterval` property is ignored until a trigger occurs.)
`Running=on`	If `TriggerType` is set to `immediate` or `manual`, the Video Preview window continues to update the image displayed. If `TriggerType` is set to `hardware`, the Video Preview window stops updating the image displayed until a trigger occurs.
`Logging=on`	Video Preview window might drop some data frames, but this will not affect the frames logged to memory or disk.

Note The Image Acquisition Toolbox Preview window supports the display of up to 16-bit image data. The Preview window was designed to only show 8-bit data, but many cameras return 10-, 12-, 14-, or 16-bit data. The Preview window display supports these higher bit-depth cameras. However, larger bit data is scaled to 8-bit for the purpose of displaying previewed data. To capture the image data in the Preview window in its full bit depth for grayscale images, set the `PreviewFullBitDepth` property to `'on'`.

Custom Update Function

`preview` creates application-defined data for the image object, `himage`, assigning it the name `'UpdatePreviewWindowFcn'` and setting its value to an empty array (`[]`). You can configure the value of the `'UpdatePreviewWindowFcn'` application data and retrieve its value using the MATLAB `setappdata` and `getappdata` functions, respectively.

The `'UpdatePreviewWindowFcn'` will not necessarily be called for every frame that is acquired. If a new frame is acquired and the `'UpdatePreviewWindowFcn'` for the previous frame has not yet finished executing, no update will be generated for the new frame. If you need to execute a function for every acquired frame, use the `FramesAcquiredFcn` instead.

You can use this function to define custom processing of the previewed image data. When `preview` invokes the function handle you specify, it passes three arguments to your function:

- `obj` — The video input object being previewed
- `event` — An event structure containing image frame information. For more information, see below.
- `himage` — A handle to the image object that is being updated

The event structure contains the following fields:

Field	Description
`Data`	Current image frame specified as an H-by-W-by-B matrix where H and W are the image height and width, respectively, as specified in the `ROIPosition` property, and B is the number of color bands, as specified in the `NumberOfBands` property.
`Resolution`	Character vector specifying current image width and height, as defined by the `ROIPosition` property.
`Status`	Character vector describing the current acquisition status of the video input object.
`Timestamp`	Character vector specifying the timestamp associated with the current image frame.
`FrameRate`	Character vector specifying the current frame rate of the video input object in frames per second.

Examples

Create a customized GUI.

```
figure('Name', 'My Custom Preview Window');
uicontrol('String', 'Close', 'Callback', 'close(gcf)');
```

Create an image object for previewing.

```
vidRes = obj.VideoResolution;
nBands = obj.NumberOfBands;
hImage = image( zeros(vidRes(2), vidRes(1), nBands) );
preview(obj, hImage);
```

For more information on customized GUIs, see “Previewing Data in Custom GUIs” on page 2-9.

Version History

Introduced before R2006a

See Also

`ancestor` | `closepreview` | `image` | `imaqhelp` | `stoppreview`

propinfo

Property characteristics for image acquisition objects

Syntax

```
out = propinfo(obj)
out = propinfo(obj,PropertyName)
```

Description

`out = propinfo(obj)` returns the structure `out` whose field names are the names of all the properties supported by `obj`. `obj` must be a 1-by-1 image acquisition object. The value of each field is a structure containing the fields shown below.

Field Name	Description
`Type`	Data type of the property. Possible values are `'any'`, `'callback'`, `'double'`, `'character vector'`, and `'struct'`.
`Constraint`	Type of constraint on the property value. Possible values are `'bounded'`, `'callback'`, `'enum'`, and `'none'`.
`ConstraintValue`	List of valid character vector values or a range of valid values.
`DefaultValue`	Default value for the property.
`ReadOnly`	Condition under which a property is read only: • `'always'` — Property cannot be configured. • `'whileRunning'` — Property cannot be configured while `Running` is set to on. • `'never'` — Property can be configured at any time.
`DeviceSpecific`	`1` if the property is device specific; otherwise, `0` (zero).

`out = propinfo(obj,PropertyName)` returns the structure `out` for the property specified by `PropertyName`. If *`PropertyName`* is a cell array of character vectors, `propinfo` returns a structure for each property, stored in a cell array.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Examples

Create the video input object `vid`.

```
vid = videoinput('winvideo',1);
```

Capture all property information for all properties.

```
out = propinfo(vid);
```

Access property information for a particular property.

```
out1 = propinfo(vid,'LoggingMode');
```

Version History

Introduced before R2006a

See Also

`imaqhelp`

save

Save image acquisition objects to MAT-file

Syntax

```
save filename
save filename obj1 obj2 ...
save(filename,obj1,obj2,...)
```

Description

`save filename` saves all variables in the MATLAB workspace to the MAT-file `filename`. If `filename` does not include a file extension, `save` appends the `.MAT` extension to the filename.

`save filename obj1 obj2 ...` saves the specified image acquisition objects (`obj1`, `obj2`, etc.) to the MAT-file `filename`.

`save(filename,obj1,obj2,...)` is the functional form of the command, where the file name and image acquisition objects must be specified as character vectors. If no objects are specified, then all variables existing in the MATLAB workspace are saved.

Note that any data associated with the image acquisition object is not stored in the MAT-file. To save the data, bring it into the MATLAB workspace (using the `getdata` function), and then save the variable to the MAT-file.

To return variables from the MAT-file to the MATLAB workspace, use the `load` command. Values for read-only properties are restored to their default values upon loading. For example, the `Running` property is restored to `'off'`. Use the `propinfo` function to determine if a property is read only.

Examples

```
obj = videoinput('winvideo', 1);
obj.SelectedSourceName = 'input1'
save fname obj
obj.TriggerFcn = {'mycallback', 5};
save('fname1', 'obj')
```

Version History

Introduced before R2006a

See Also

`imaqhelp` | `load` | `propinfo`

set

Configure or display image acquisition object properties

Syntax

```
set(obj)
prop_struct = set(obj)
set(obj,PropertyName)
prop_cell = set(obj,PropertyName)
set(obj,PropertyName,PropertyValue,...)
set(obj,S)
set(obj,PN,PV)
```

Description

`set(obj)` displays property names and any enumerated values for all configurable properties of image acquisition object `obj`. `obj` must be a single image acquisition object.

`prop_struct = set(obj)` returns the property names and any enumerated values for all configurable properties of image acquisition object `obj`. `obj` must be a single image acquisition object. The return value `prop_struct` is a structure whose field names are the property names of `obj`, and whose values are cell arrays of possible property values or empty cell arrays if the property does not have a finite set of possible character vector values.

`set(obj,`*`PropertyName`*`)` displays the possible values for the specified property, *`PropertyName`*, of image acquisition object `obj`. `obj` must be a single image acquisition object. Use the `set(obj)` syntax to get a list of all the properties for a particular image acquisition object that can be set.

`prop_cell = set(obj,`*`PropertyName`*`)` returns the possible values for the specified property, *`PropertyName`*, of image acquisition object `obj`. `obj` must be a single image acquisition object. The returned array `prop_cell` is a cell array of possible values or an empty cell array if the property does not have a finite set of possible character vector values.

`set(obj,`*`PropertyName`*`,PropertyValue,...)` configures the property specified by the character vector *`PropertyName`* to the value specified by `PropertyValue` for image acquisition object `obj`. You can specify multiple property name/property value pairs in a single statement. `obj` can be a single image acquisition object or a vector of image acquisition objects, in which case `set` configures the property values for all the image acquisition objects specified.

`set(obj,S)` configures the properties of `obj` with the values specified in `S`, where `S` is a structure whose field names are object property names.

`set(obj,PN,PV)` configures the properties specified in the cell array of character vectors, `PN`, to the corresponding values in the cell array `PV`, for the image acquisition object `obj`. `PN` must be a vector. If `obj` is an array of image acquisition objects, `PV` can be an M-by-N cell array, where M is equal to the length of the image acquisition object array and N is equal to the length of `PN`. In this case, each image acquisition object is updated with a different set of values for the list of property names contained in `PN`.

Note Parameter/value character vector pairs, structures, and parameter/value cell array pairs can be used in the same call to `set`.

Examples

These examples illustrate the various ways to use the `set` function to set the values of image acquisition object properties.

```
set(obj, 'FramesPerTrigger', 15, 'LoggingMode', 'disk');
set(obj, {'TimerFcn', 'TimerPeriod'}, {@imaqcallback, 25});
set(obj, 'Name', 'MyObject');
set(obj, 'SelectedSourceName')
```

Instead of using `set` to set individual property values, you should use dot notation. So for example, instead of this:

```
set(vid, 'FramesPerTrigger', 100);
```

You should use this syntax:

```
vid.FramesPerTrigger = 100;
```

Version History

Introduced before R2006a

See Also

`get` | `imaqfind` | `videoinput`

snapshot

Acquire single image frame from GigE Vision camera

Syntax

```
img = snapshot(g);
[img, ts] = snapshot(g);
```

Description

`img = snapshot(g);` acquires the current frame as a single image from the GigE Vision camera `g` and assigns it to the variable `img`. If you call `snapshot` in a loop, then it returns a new frame each time. The returned image is based on the Pixel Format of your camera. `snapshot` uses the camera's default resolution or another resolution that you specify using the `Height` and `Width` properties, if available.

Note The `snapshot` function is for use only with the `gigecam` object. To acquire images using the `videoinput` object, use the `getsnapshot` or `getdata` functions.

`[img, ts] = snapshot(g);` acquires the current frame as a single image from the GigE Vision camera `g` and assigns it to the variable `img`, and assigns the timestamp to the variable `ts`.

Examples

Acquire One Image Frame from GigE Vision Camera

Use the `snapshot` function to acquire one image frame from a GigE Vision camera. You then show it using a display function such as `imshow` or `image`.

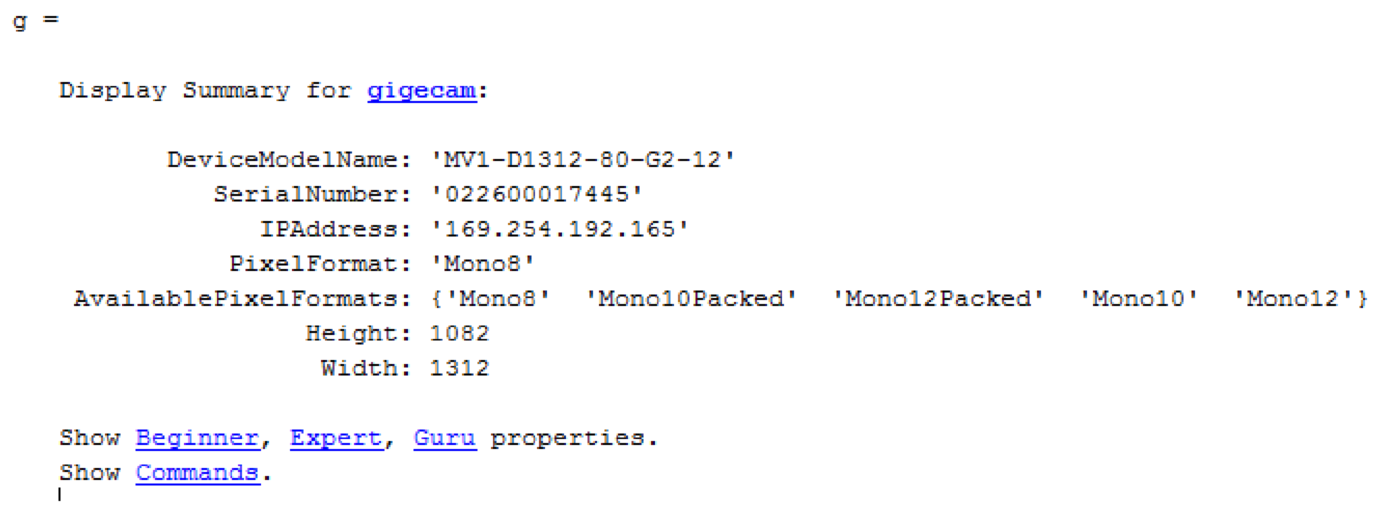
Use the `gigecamlist` function to ensure that MATLAB is discovering your camera.

```
gigecamlist
```

```
ans = 

    Model                 Manufacturer           IPAddress        SerialNumber  
  ____________________    ___________________    _______________    ______________

  'MV1-D1312-80-G2-12'    'Photonofocus AG'      '169.254.192.165'  '022600017445'
```

Use the `gigecam` function to create the object and connect it to the camera.

```
g = gigecam
```



```
g =

   Display Summary for gigecam:

          DeviceModelName: 'MV1-D1312-80-G2-12'
             SerialNumber: '022600017445'
                IPAddress: '169.254.192.165'
              PixelFormat: 'Mono8'
    AvailablePixelFormats: {'Mono8'  'Mono10Packed'  'Mono12Packed'  'Mono10'  'Mono12'}
                   Height: 1082
                    Width: 1312

   Show Beginner, Expert, Guru properties.
   Show Commands.
```

Preview the image from the camera.

```
preview(g)
```

The preview window displays live video stream from your camera. If you change a property while previewing, then the preview dynamically updates, and the image reflects the property change.

Close the preview.

```
closePreview(g)
```

Acquire a single image from the camera using the `snapshot` function, and assign it to the variable `img`.

```
img = snapshot(g);
```

Display the acquired image.

```
imshow(img)
```

Clean up by clearing the object.

```
clear g
```

Version History

Introduced in R2014b

See Also

`gigecamlist` | `gigecam` | `commands` | `executeCommand`

snapshot

Acquire single image frame from Matrox frame grabber

Syntax

```
img = snapshot(m);
[img, ts] = snapshot(m);
```

Description

`img = snapshot(m);` acquires the current frame as a single image from the Matrox frame grabber `m` and assigns it to the variable `img`. If you call `snapshot` in a loop, then it returns a new frame each time.

Note The `snapshot` function is for use only with the `matroxcam` object. To acquire images using the `videoinput` object, use the `getsnapshot` or `getdata` functions.

`[img, ts] = snapshot(m);` acquires the current frame as a single image from the Matrox frame grabber `m` and assigns it to the variable `img`, and assigns the timestamp to the variable `ts`.

Examples

Acquire One Image Frame from Matrox Frame Grabber

Use the `snapshot` function to acquire one image frame from a Matrox frame grabber. You then show it using a display function such as `imshow` or `image`.

Use the `matroxlist` function to ensure that MATLAB is discovering your frame grabber.

```
matroxlist
```

```
ans =

   Solios XCL (digitizer 0)
   Solios XCL (digitizer 1)
   VIO (digitizer 0)
```

Use the `matroxcam` function to create the object and connect it to the frame grabber. If you want to use the second frame grabber in the list, the Solios XCL at digitizer 1, use a `2` as the index number, since it is the second board on the list. The second argument must be the name and path of your DCF file, entered as a character vector.

```
m = matroxcam(2, 'C:\Drivers\Solios\dcf\XCL\Basler\A404K\mycam.dcf');
```

```
m =

Display Summary for matroxcam:

         DeviceName: 'Solios XCL (digitizer 1)'
```

```
          DCFName: 'C:\Drivers\Solios\dcf\XCL\Basler\A404K\mycam.dcf'
  FrameResolution: '1300 x 1080'
          Timeout: 10
```

The DCF file is specified so that the acquisition can use the properties you have set in your DCF file.

Preview the image from the frame grabber.

```
preview(m)
```

You can leave the **Preview** window open, or close it any time. To close the preview:

```
closePreview(m)
```

Acquire a single image from the frame grabber using the `snapshot` function, and assign it to the variable `img`.

```
img = snapshot(m);
```

Display the acquired image.

```
imshow(img)
```

Clean up by clearing the object.

```
clear m
```

Note about Hardware Triggering: If your DCF file is configured for hardware triggering, then you must provide the trigger to acquire images. To do that, call the `snapshot` function as you normally would, as shown in this example, and then perform the hardware trigger to acquire the frame. When you call the `snapshot` function with hardware triggering set, it will not timeout as it normally would. Therefore, the MATLAB command-line will be blocked until you perform the hardware trigger.

Version History
Introduced in R2014b

See Also
`matroxlist` | `matroxcam` | `preview` | `closepreview`

Topics
"Connect to Matrox Frame Grabbers" on page 13-3
"Set Properties for Matrox Acquisition" on page 13-4
"Acquire Images from Matrox Frame Grabbers" on page 13-6
"Matrox Acquisition – matroxcam Object vs videoinput Object" on page 13-2

start

Obtain exclusive use of image acquisition device

Syntax

```
start(obj)
```

Description

`start(obj)` obtains exclusive use of the image acquisition device associated with the video input object `obj` and locks the device's configuration. Starting an object is a necessary first step to acquire image data, but it does not control when data is logged.

`obj` can either be a 1-by-1 video input object or an array of video input objects.

Data logging is controlled with the `TriggerType` property.

Trigger Type	Logging Behavior
`'hardware'`	Data logging occurs when the condition specified in the object's `TriggerCondition` property is met via the `TriggerSource`.
`'immediate'`	Data logging occurs immediately.
`'manual'`	Data logging occurs when the `trigger` function is called.

Use the `triggerconfig` function to configure the object's trigger settings.

When an acquisition is started, `obj` performs the following operations:

1. Transfers the object's configuration to the associated hardware.
2. Executes the object's `StartFcn` callback.
3. Sets the object's `Running` property to `'On'`.

If the object's `StartFcn` errors, the hardware is never started and the object's `Running` property remains `'Off'`.

The start event is recorded in the object's `EventLog` property.

An image acquisition object stops running when one of the following conditions is met:

- The `stop` function is issued.
- The requested number of frames is acquired. This occurs when

  ```
  FramesAcquired = FramesPerTrigger * (TriggerRepeat + 1)
  ```

  where `FramesAcquired`, `FramesPerTrigger`, and `TriggerRepeat` are properties of the video input object.
- A run-time error occurs.
- The object's `Timeout` value is reached.

Examples

The `start` function can be called by a video input object's event callback.

```
obj.StopFcn = {'start'};
```

Version History

Introduced before R2006a

See Also

`imaqfind` | `imaqhelp` | `propinfo` | `stop` | `trigger` | `triggerconfig`

stop

Stop video input object

Syntax

```
stop(obj)
```

Description

`stop(obj)` halts an acquisition associated with the video input object `obj`. `obj` can be either a single video input object or an array of video input objects.

The `stop` function

- Sets the object's `Running` property to `'Off'`
- Sets the object's `Logging` property to `'Off'`, if needed
- Executes the object's `StopFcn` callback

An image acquisition object can also stop running under one of the following conditions:

- The requested number of frames is acquired. This occurs when

  ```
  FramesAcquired = FramesPerTrigger * (TriggerRepeat + 1)
  ```

  where `FramesAcquired`, `FramesPerTrigger`, and `TriggerRepeat` are properties of the video input object.
- A run-time error occurs.
- The object's `Timeout` value is reached.

The stop event is recorded in the object's `EventLog` property.

Examples

The `stop` function can be called by a video input object's event callback.

```
obj.TimerFcn = {'stop'};
```

Version History

Introduced before R2006a

See Also

`imaqfind` | `start` | `trigger` | `propinfo` | `videoinput`

stoppreview

Stop previewing video data

Syntax

```
stoppreview(obj)
```

Description

`stoppreview(obj)` stops the previewing of video data from image acquisition object `obj`.

To restart previewing, call `preview` again.

Examples

Create a video input object and open a Video Preview window.

```
vid = videoinput('winvideo',1);
preview(vid)
```

Stop previewing video data.

```
stoppreview(vid);
```

Restart previewing.

```
preview(vid)
```

Version History

Introduced before R2006a

See Also

`closepreview` | `preview`

trigger

Initiate data logging

Syntax

`trigger(obj)`

Description

`trigger(obj)` initiates data logging for the video input object `obj`. `obj` can be either a single video input object or an array of video input objects.

The `trigger` function

- Executes the object's `TriggerFcn` callback
- Records the absolute time of the first trigger event in the object's `InitialTriggerTime` property
- Configures the object's `Logging` property to `'On'`

`obj` must be running and its `TriggerType` property must be set to `'manual'`. To start an object running, use the `start` function.

The trigger event is recorded in the object's `EventLog` property.

Examples

The `trigger` function can be called by a video input object's event callback.

```
obj.StartFcn = @trigger;
```

Version History

Introduced before R2006a

See Also

`imaqfind` | `start` | `stop` | `videoinput`

triggerconfig

Configure video input object trigger properties

Syntax

```
triggerconfig(obj,type)
triggerconfig(obj,type,condition)
triggerconfig(obj,type,condition,source)
config = triggerconfig(obj)
triggerconfig(obj,config)
```

Description

`triggerconfig(obj,`*`type`*`)` configures the value of the `TriggerType` property of the video input object `obj` to the value specified by the character vector *`type`*. For a list of valid `TriggerType` values, use `triggerinfo(obj)`. *`type`* must specify a unique trigger configuration.

`obj` can be either a single video input object or an array of video input objects. If an error occurs, any video input objects in the array that have already been configured are returned to their original configurations.

`triggerconfig(obj,`*`type`*`,`*`condition`*`)` configures the values of the `TriggerType` and `TriggerCondition` properties of the video input object `obj` to the values specified by the character vectors *`type`* and *`condition`*. For a list of valid `TriggerType` and `TriggerCondition` values, use `triggerinfo(obj)`. *`type`* and *`condition`* must specify a unique trigger configuration.

`triggerconfig(obj,`*`type`*`,`*`condition`*`,`*`source`*`)` configures the values of the `TriggerType`, `TriggerCondition`, and `TriggerSource` properties of the video input object `obj` to the values specified by the character vectors *`type`*, *`condition`*, and *`source`*, respectively. For a list of valid `TriggerType`, `TriggerCondition`, and `TriggerSource` values, use `triggerinfo(obj)`.

`config = triggerconfig(obj)` returns a MATLAB structure `config` containing the object's current trigger configuration. `obj` must be a 1-by-1 video input object. The field names of `config` are `TriggerType`, `TriggerCondition`, and `TriggerSource`. Each field contains the current value of the object's property.

`triggerconfig(obj,config)` configures the `TriggerType`, `TriggerCondition`, and `TriggerSource` property values for video input object `obj` using `config`, a MATLAB structure with field names `TriggerType`, `TriggerCondition`, and `TriggerSource`, each containing the desired property value.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see "Using Tab Completion for Functions" on page 5-17.

Examples

Example 1

Construct a video input object.

```
vid = videoinput('winvideo', 1);
```

Configure trigger properties for the object.

```
triggerconfig(vid, 'manual')
```

Trigger the acquisition.

```
start(vid)
trigger(vid)
```

Remove video input object from memory.

```
delete(vid);
```

Example 2

This example uses a structure returned from `triggerinfo` to configure trigger parameters.

Create a video input object.

```
vid = videoinput('winvideo', 1);
```

Use `triggerinfo` to get all valid configurations for the trigger properties for the object.

```
config = triggerinfo(vid);
```

Pass one of the configurations to the `triggerconfig` function.

```
triggerconfig(vid,config(2));
```

Remove video input object from memory.

```
delete(vid);
```

Version History

Introduced before R2006a

See Also

`imaqhelp` | `trigger` | `triggerinfo` | `videoinput`

triggerinfo

Provide information about available trigger configurations

Syntax

```
triggerinfo(obj)
triggerinfo(obj,type)
config = triggerinfo(...)
```

Description

`triggerinfo(obj)` displays all available trigger configurations for the video input object `obj`. `obj` can only be a 1-by-1 video input object.

`triggerinfo(obj,type)` displays the available trigger configurations for the specified `TriggerType`, *`type`*, for the video input object `obj`. To get a list of valid *`type`* values for a particular image acquisition object, use `triggerinfo(obj)`.

`config = triggerinfo(...)` returns `config`, an array of MATLAB structures, containing all the valid trigger configurations for the video input object `obj`. Each structure in the array contains these fields:

Field	**Description**
`TriggerType`	Name of the trigger type
`TriggerCondition`	Condition that must be met before executing a trigger
`TriggerSource`	Hardware source used for triggering

You can pass one of the structures in `config` to the `triggerconfig` function to specify the trigger configuration.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Examples

This example illustrates how to use the `triggerinfo` function to retrieve valid configurations of the `TriggerType`, `TriggerSource`, and `TriggerCondition` properties.

1 Create a video input object.

```
vid = videoinput('winvideo');
```

2 Get information about the available trigger configurations for this object.

```
config = triggerinfo(vid)
config =

1x2 struct array with fields:
    TriggerType
    TriggerCondition
    TriggerSource
```

3 View one of the trigger configurations returned by `triggerinfo`.

```
config(1)

ans =

         TriggerType: 'immediate'
    TriggerCondition: 'none'
       TriggerSource: 'none'
```

Version History

Introduced before R2006a

See Also

`imaqhelp` | `triggerconfig`

videoinput

Create video input object

Description

A `videoinput` object represents a connection between MATLAB and a particular image acquisition device.

Creation

Syntax

```
vid = videoinput(adaptor)
vid = videoinput(adaptor,deviceID)
vid = videoinput(adaptor,deviceID,format)
vid = videoinput( ___ ,Name,Value)
```

Description

`vid = videoinput(adaptor)` creates a video input object `vid`. `adaptor` is a character vector that specifies the name of the adaptor used to communicate with the device. Use `imaqhwinfo` to determine the adaptors available on your system.

`vid = videoinput(adaptor,deviceID)` creates a video input object `vid`, where `deviceID` is a numeric scalar value that identifies a particular device available through the specified adaptor, `adaptor`. Use `imaqhwinfo(adaptor)` to determine the devices available through the specified adaptor. If `deviceID` is not specified, the first available device ID is used. You can also use a device's name in place of `deviceID`. If multiple devices have the same name, the first available device is used.

`vid = videoinput(adaptor,deviceID,format)` creates a video input object `vid`, where `format` is a character vector that specifies a particular video format supported by the device or the full path of a device configuration file (also known as a camera file). To get a list of the formats supported by a particular device, view the `DeviceInfo` structure for the device that is returned by `imaqhwinfo`. Each `DeviceInfo` structure contains a `SupportedFormats` field. If `format` is not specified, the device's default format is used. When the video input object is created, its `VideoFormat` property contains the format name or device configuration file that you specify.

`vid = videoinput( ___ ,Name,Value)` creates a video input object and sets additional properties using one or more name-value arguments in addition to the input arguments in previous syntaxes. If an invalid property name or property value is specified, the object is not created. Use tab-completion to see a full list of properties that you can set for your adaptor using name-value arguments.

To view a complete list of video input object functions and properties, use `imaqhelp`.

Note The toolbox chooses the first available video source object as the selected source and specifies this video source object's name in the object's `SelectedSourceName` property. Use `getselectedsource(obj)` to access the video source object that is used for acquisition.

Properties

General Properties

`BayerSensorAlignment` — Sensor alignment for Bayer demosaicing

`"grbg"` (default) | `"gbrg"` | `"rggb"` | `"bggr"`

Sensor alignment for Bayer demosaicing, specified as `"grbg"`, `"gbrg"`, `"rggb"`, or `"bggr"`. If the `ReturnedColorSpace` property is set to `"bayer"`, then the Image Acquisition Toolbox will demosaic Bayer patterns returned by the hardware. This color space setting interpolates Bayer pattern encoded images into standard RGB images. If your camera uses Bayer filtering, the toolbox supports the Bayer pattern and can return color if desired.

In order to perform the demosaicing, the toolbox needs to know the pixel alignment of the sensor. This is the order of the red, green, and blue sensors and is normally specified by describing the four pixels in the upper-left corner of the sensor. It is the band sensitivity alignment of the pixels as interpreted by the camera's internal hardware. You must get this information from the camera's documentation and then specify the value for the alignment, as described in the following table.

There are four possible sensor alignments.

Value	Description

`"gbrg"`	The 2-by-2 sensor alignment is
	
	`green blue`
	`red   green`

`"grbg"`	The 2-by-2 sensor alignment is
	
	`green red`
	`blue  green`

`"bggr"`	The 2-by-2 sensor alignment is
	
	`blue  green`
	`green red`

`"rggb"`	The 2-by-2 sensor alignment is
	
	`red   green`
	`green blue`

The value of this property is only used if the `ReturnedColorSpace` property is set to `"bayer"`.

For examples showing how to convert Bayer images, see “Converting Bayer Images” on page 7-15.

Data Types: `char` | `string`

`DeviceID` — Image acquisition device ID

1 (default) | nonnegative integer

This property is read-only.

Image acquisition device ID for the specified adaptor, specified as a nonnegative integer. This property identifies the device represented by the video input object.

A device ID is a number, assigned by an adaptor, that uniquely identifies an image acquisition device. The adaptor assigns the first device it detects the identifier `1`, the second device it detects the identifier `2`, and so on.

You can specify the device ID as an input for the `videoinput` function when you create a video input object. The object stores the value in the `DeviceID` property and also uses the value when constructing the default value of the `Name` property.

To get a list of the IDs of the devices connected to your system, use the `imaqhwinfo` function, specifying the name of a particular adaptor as an argument.

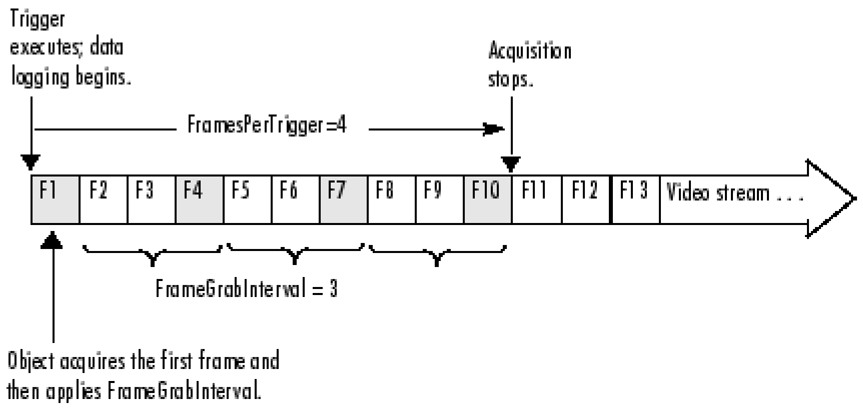
Data Types: `double`

`FrameGrabInterval` — How often to acquire frame from video stream
`1` (default) | positive integer.

How often the video input object acquires a frame from the video stream, specified as a positive integer. By default, objects acquire every frame in the video stream, but you can use this property to specify other acquisition intervals.

Note Do not confuse the frame grab interval with the frame rate. The frame rate describes the rate at which an image acquisition device provides frames, typically measured in seconds, such as 30 frames per second. The frame grab interval is measured in frames, not seconds. If a particular device's frame rate is configurable, the video source object might include the frame rate as a device-specific property.

For example, when you specify a `FrameGrabInterval` value of `3`, the object acquires every third frame from the video stream, as illustrated in this figure. The object acquires the first frame in the video stream before applying the `FrameGrabInterval`.


Trigger executes; data logging begins.
Acquisition stops.
FramesPerTrigger=4
F1
F2
F3
F4
F5
F6
F7
F8
F9
F10
F11
F12
F13
Video stream . . .
FrameGrabInterval = 3
Object acquires the first frame and then applies FrameGrabInterval.

You specify the source of the video stream in the `SelectedSourceName` property.

Data Types: `double`

`FramesAcquired` — Total number of frames acquired
`0` (default) | nonnegative integer

This property is read-only.

Total number of frames that the object has acquired, regardless of how many frames have been extracted from the memory buffer, specified as a nonnegative integer. The video input object continuously updates the value of the `FramesAcquired` property as it acquires frames.

Note When you issue a `start` command, the video input object resets the value of the `FramesAcquired` property to `0` (zero) and flushes the buffer.

To find out how many frames are available in the memory buffer, use the `FramesAvailable` property.

Data Types: `double`

`FramesAvailable` — Number of frames available in memory buffer
`0` (default) | nonnegative integer

This property is read-only.

Total number of frames that are available in the memory buffer, specified as a nonnegative integer. When you extract data, the object reduces the value of the `FramesAvailable` property by the appropriate number of frames. You use the `getdata` function to extract data and move it into the MATLAB workspace.

Note When you issue a `start` command, the video input object resets the value of the `FramesAvailable` property to `0` (zero) and flushes the buffer.

To view the total number of frames that have been acquired since the last `start` command, use the `FramesAcquired` property.

Data Types: `double`

`Name` — Name of image acquisition object
character vector | string scalar

Name of the image acquisition object, specified as a character vector or string scalar.

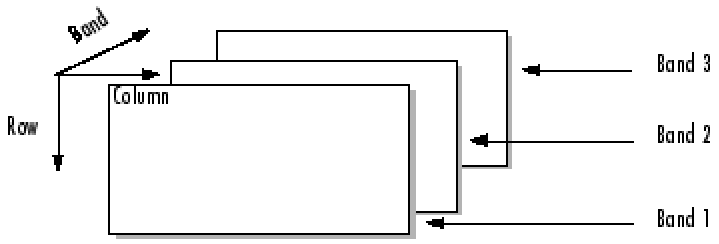
The toolbox creates the default name by combining the values of the `VideoFormat` and `DeviceID` properties with the adaptor name in this format: `VideoFormat` + `'-'` + adaptor name + `'-'` + `DeviceID`

Data Types: `char` | `string`

`NumberOfBands` — Number of color bands in data to be acquired
positive integer

This property is read-only.

Number of color bands in the data to be acquired, specified as a positive integer. The toolbox defines *band* as the third dimension in a 3-D array, as shown in this figure.



The value of the NumberOfBands property indicates the number of color bands in the data returned by getsnapshot, getdata, and peekdata.

Data Types: double

PreviewFullBitDepth — Whether preview data is displayed in full bit depth
"off" (default) | "on"

Whether the image data in the Preview window is being displayed in full bit depth, specified as "off" or "on".

Note The Image Acquisition Toolbox Preview window supports the display of up to 16-bit image data. The Preview window was designed to only show 8-bit data, but many cameras return 10-, 12-, 14-, or 16-bit data. The Preview window display supports these higher bit-depth cameras. However, larger bit data is scaled to 8-bit for the purpose of displaying previewed data. To capture the image data in the Preview window in its full bit depth for grayscale images, set the PreviewFullBitDepth property to 'on'.

If you set this property to "off", image data in the preview window is scaled down from its bit depth to 8-bit. If you set this property to "on", the image data in the preview window is being captured in its full bit depth.

This property can be set to "on" only when the value of the ReturnedColorspace property is set to "grayscale" and for video formats higher than 8-bit depth.

Data Types: char | string

Previewing — Whether object is currently previewing data in separate window
"off" (default) | "on"

This property is read-only.

Whether the object is currently previewing data in a separate window, specified as "off" or "on".

The object sets the Previewing property to "on" when you call the preview function.

The object sets the Previewing property to "off" when you close the preview window using the closepreview function or by clicking the **Close** button in the preview window title bar.

Data Types: char

ReturnedColorSpace — Color space used in MATLAB
"grayscale" | "rgb" | "YCbCr" | "bayer"

Color space you want the toolbox to use when it returns image data to the MATLAB workspace, specified as `"grayscale"`, `"rgb"`, `"YCbCr"`, or `"bayer"`. This is only relevant when you are accessing acquired image data with the `getsnapshot`, `getdata`, and `peekdata` functions.

This property can have any of the following values:

Value	**Description**
`"grayscale"`	MATLAB grayscale color space.
`"rgb"`	MATLAB RGB color space.
`"YCbCr"`	MATLAB YCbCr color space. Note that YCbCr is often imprecisely referred to as YUV. (YUV is similar, but not identical. They differ by the scaling factor applied to the result. YUV refers to a particular scaling factor used in composite NTSC and PAL formats. In most cases, you can specify the YCbCr color space for devices that support YUV.)
`"bayer"`	Convert grayscale Bayer color patterns to RGB images. The `bayer` color space option is only available if your camera's default returned color space is `grayscale`. To use the `BayerSensorAlignment` property, you must set the `ReturnedColorSpace` property to `bayer`.

Note For some adaptors, such as GigE and GenTL, if you use a format that starts with Bayer (e.g. `BayerGB8_640x480`), the raw Bayer pattern is automatically converted to color – the `ReturnedColorSpace` is RGB. If you set the `ReturnedColorSpace` to `"grayscale"`, you'll get the raw pattern.

For an example showing how to determine the default color space and change the color space setting, see “Specifying the Color Space” on page 7-14.

Data Types: `char` | `string`

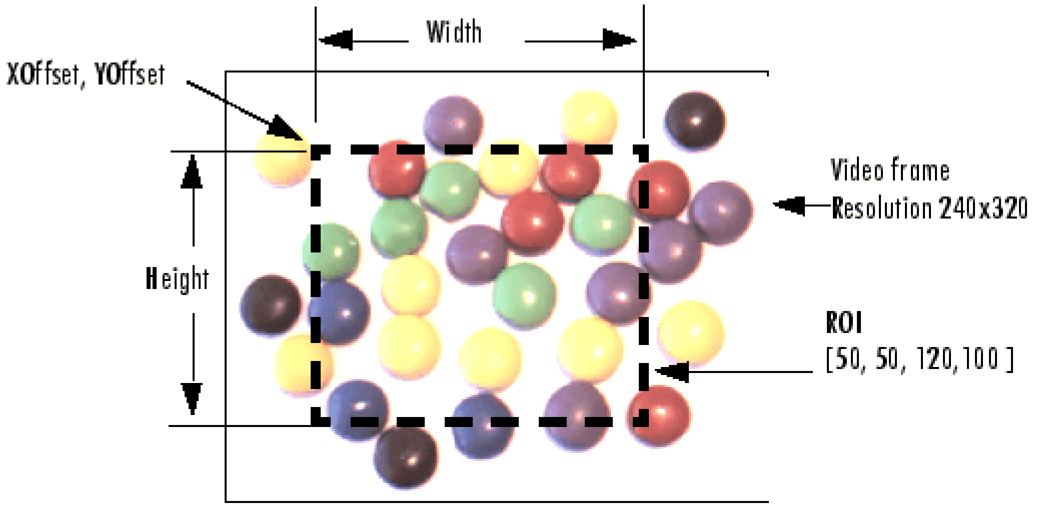
`ROIPosition` — Region-of-interest (ROI) window

`[ 0 0 width height ]` (default) | 1-by-4 element vector

Region-of-interest acquisition window, specified as a 1-by-4 element vector. The ROI defines the actual size of the frame logged by the toolbox, measured with respect to the top left corner of an image frame.

`ROIPosition` is specified as a 1-by-4 element vector `[XOffset YOffset Width Height]`.

`XOffset`	Position of the upper left corner of the ROI, measured in pixels.
`YOffset`	Position of the upper left corner of the ROI, measured in pixels.
`Width`	Width of the ROI, measured in pixels. The sum of `XOffset` and `Width` cannot exceed the width specified in `VideoResolution`.
`Height`	Height of the ROI, measured in pixels. The sum of `YOffset` and `Height` cannot exceed the height specified in `VideoResolution`.



Note The `Width` does not include both end points as well as the width between the pixels. It includes one end point, plus the width between pixels. For example, if you want to capture an ROI of pixels 20 through 30, including both end pixels 20 and 30, set an `XOffset` of `19` and a `Width` of `11`. The same rule applies to `height`.

In the figure shown above, the width of the captured ROI contains pixels 51 through 170, including both end points, because the `XOffset` is set to `50` and the `Width` is set to `120`.

Data Types: `double`

Running — Whether video input object is ready to acquire data
`"off"` (default) | `"on"`

This property is read-only.

Whether the video input object is ready to acquire data, specified as `"off"` or `"on"`.

Along with the `Logging` property, `Running` reflects the state of a video input object. The `Running` property indicates that the object is ready to acquire data, while the `Logging` property indicates that the object is acquiring data.

The object sets the `Running` property to `"on"` when you issue the `start` command. When `Running` is `"on"`, you can acquire data from a video source.

The object sets the `Running` property to `"off"` when any of the following conditions is met:

- The specified number of frames has been acquired.
- A run-time error occurs.
- You issue the `stop` command.

When `Running` is `"off"`, you cannot acquire image data. However, you can acquire one image frame with the `getsnapshot` function.

Data Types: `char`

`Timeout` — Amount of time to wait for image data

10 (default) | positive integer

Amount of time in seconds that the `getdata` and `getsnapshot` functions wait for data to be returned, specified as a positive integer. The `Timeout` property is only associated with these blocking functions. If the specified time period expires, the functions return control to the MATLAB command line.

A timeout is one of the conditions for stopping an acquisition. When a timeout occurs, and the object is running, the MATLAB file function specified by `ErrorFcn` is called.

Note The `Timeout` property is not associated with hardware timeout conditions.

Data Types: `double`

`UserData` — Stored data to associate with image acquisition object

any type

Stored data that you want to associate with an image acquisition object, specified as any MATLAB data type.

Note The object does not use the data in `UserData` directly. However, you can access the data by referencing the property as you would a field in a MATLAB structure using dot notation.

`VideoFormat` — Video format or name of device configuration file

character vector

This property is read-only.

Video format used by the image acquisition device or the name of a device configuration file, depending on which you specified when you created the object, specified as a character vector.

Image acquisition devices typically support multiple video formats. When you create a video input object, you can specify the video format that you want the device to use. If you do not specify the video format as an argument, the `videoinput` function uses the default format. Use the `imaqhwinfo` function to determine which video formats a particular device supports and find out which format is the default.

As an alternative, you can specify the name of a device configuration file, also known as a camera file or digitizer configuration format (DCF) file. Some image acquisition devices use these files to store device configuration information. The `videoinput` function can use this file to determine the video format and other configuration information.

Use the `imaqhwinfo` function to determine if your device supports device configuration files.

Data Types: `char`

`VideoResolution` — Width and height of incoming video stream

`[width height]`

This property is read-only.

Width and height in pixels of the frames in the incoming video stream, specified as a two-element vector `[width height]`.

Note You specify the video resolution when you create the video input object, by passing in the video format argument to the `videoinput` function. If you do not specify a video format, the `videoinput` function uses the default video format. Use the `imaqhwinfo` function to determine which video formats a particular device supports and find out which format is the default.

Data Types: `double`

Data Logging Properties

`Logging` — Whether object is currently logging data
`"off"` (default) | `"on"`

This property is read-only.

Whether video input object is currently logging data, specified as `"off"` or `"on"`.

When a trigger occurs, the object sets the `Logging` property to `"on"` and logs data to memory, a disk file, or both, depending on the value of the `LoggingMode` property.

The object sets the `Logging` property to `"off"` when it acquires the requested number of frames, an error occurs, or you issue a `stop` command.

To acquire data when the object is running but not logging, use the `peekdata` function. The `peekdata` function does not guarantee that all the requested image data is returned. To acquire all the data without gaps, you must have the object log the data to memory or to a disk file.

Data Types: `char`

`LoggingMode` — Destination for acquired data
`"memory"` (default) | `"disk"` | `"disk&memory"`

Destination for acquired data, specified as `"memory"`, `"disk"`, or `"disk&memory"`. This property specifies where you want the video input object to store the acquired data. You can specify any of the following values:

Value	Description
`"disk"`	Log acquired data to a disk file.
`"disk&memory"`	Log acquired data to both a disk file and to a memory buffer.
`"memory"`	Log acquired data to a memory buffer.

If you select `"disk"` or `"disk&memory"`, you must specify the AVI file object used to access the disk file as the value of the `DiskLogger` property.

Note When logging data to memory, you must extract the acquired data in a timely manner with the `getdata` function to avoid using up all the memory that is available on your system.

Note The `peekdata` function does not return any data while running if in disk logging mode.

Data Types: `char` | `string`

`DiskLogger` — MATLAB `VideoWriter` file used to log data
`[]` (default) | `VideoWriter` object

MATLAB `VideoWriter` file used to log data, specified as a `VideoWriter` object. This property specifies the `VideoWriter` file object used to log data when the `LoggingMode` property is set to `"disk"` or `"disk&memory"`.

For the best performance, logging to disk requires a MATLAB `VideoWriter` object, which is a MATLAB object, not an Image Acquisition Toolbox object. After you create and configure a `VideoWriter` object, you can specify it with the `DiskLogger` property.

A MATLAB `VideoWriter` object specifies the file name and other characteristics. For example, you can use `VideoWriter` properties to specify the profile used for data compression and the desired quality of the output. For complete information about the `VideoWriter` object and its properties, see the `VideoWriter`.

Note Do not use the variable returned by the `VideoWriter` function to perform any operation on a `VideoWriter` file while it is being used by a video input object for data logging. For example, do not change any of the `VideoWriter` file properties, add frames, or close the object. Your changes could conflict with the video input object.

After `Logging` and `Running` are off, it is possible that the `DiskLogger` might still be writing data to disk. When the `DiskLogger` finishes writing data to disk, the value of the `DiskLoggerFrameCount` property should equal the value of the `FramesAcquired` property. Do not close or modify the `DiskLogger` until this condition is met.

For more information about logging image data using a `VideoWriter` file, see “Logging Image Data to Disk” on page 6-32.

Note The `peekdata` function does not return any data while running if in disk logging mode.

`DiskLoggerFrameCount` — Number of frames written to disk
nonnegative integer

This property is read-only.

Number of frames written to disk, specified as any nonnegative integer. This property indicates the current number of frames written to disk by the `DiskLogger`. This value is only updated when the `LoggingMode` property is set to `"disk"` or `"disk&memory"`.

After `Logging` and `Running` are off, it is possible that the `DiskLogger` might still be writing data to disk. When the `DiskLogger` finishes writing data to disk, the value of the `DiskLoggerFrameCount` property should equal the value of the `FramesAcquired` property. Do not close or modify the `DiskLogger` until this condition is met.

Data Types: `double`

Event and Callback Properties

`EventLog` — Information about events
array of structures

This property is read-only.

Information about events, specified as an array of structures. Each structure in the array represents one event. Events are recorded in the order in which they occur. The first `EventLog` structure reflects the first event recorded, the second `EventLog` structure reflects the second event recorded, and so on.

Each event log structure contains two fields: `Type` and `Data`.

The `Type` field stores a character array that identifies the event type. The Image Acquisition Toolbox defines many different event types, listed in this table. Note that not all event types are logged.

Event Type	Description	Included in Log
Error	Run-time error occurred. Run-time errors include timeouts and hardware errors.	Yes
Frames Acquired	The number of frames specified in the `FramesAcquiredFcnCount` property has been acquired.	No
Start	Object was started by calling the `start` function.	Yes
Stop	Object stopped executing.	Yes
Timer	Timer expired.	No
Trigger	Trigger executed.	Yes

The `Data` field stores information associated with the specific event. For example, all events return the absolute time the event occurred in the `AbsTime` field. Other event-specific fields are included in `Data`. For more information, see “Retrieving Event Information” on page 8-7.

`EventLog` can store a maximum of 1000 events. If this value is exceeded, then the most recent 1000 events are stored.

Data Types: `struct`

`ErrorFcn` — Callback function to execute when run-time error occurs
`imaqcallback` (default) | character vector | function handle | cell array

Callback function to execute when an error event occurs, specified as a character vector, function handle, or cell array. A run-time error event is generated immediately after a run-time error occurs.

Run-time errors include hardware errors and timeouts. Run-time errors do not include configuration errors such as setting an invalid property value.

Run-time error event information is stored in the `EventLog` property. You can retrieve any error message with the `Data.Message` field of `EventLog`.

Note Callbacks, including `ErrorFcn`, are executed only when the video object is in a running state. If you need to use the `ErrorFcn` callback for error handling during previewing, you must start the video object before previewing. To do that without logging data, use a manual trigger.

Data Types: `char` | `string` | `cell` | `function_handle`

`FramesAcquiredFcn` — Callback function to execute when specified number of frames have been acquired
`[]` (default) | character vector | function handle | cell array

Callback function to execute every time a predefined number of frames have been acquired, specified as a character vector, function handle, or cell array.

A frames acquired event is generated immediately after the number of frames specified by the `FramesAcquiredFcnCount` property is acquired from the selected video source. This event executes the MATLAB file specified for `FramesAcquiredFcn`.

Use the `FramesAcquiredFcn` callback if you must access each frame that is acquired. If you do not have this requirement, you might want to use the `TimerFcn` property.

Frames acquired event information is not stored in the `EventLog` property.

Data Types: `char` | `string` | `cell` | `function_handle`

`FramesAcquiredFcnCount` — Number of frames that must be acquired before frames acquired event is generated
`0` (default) | positive integer

Number of frames to acquire from the selected video source before a frames acquired event is generated, specified as a positive integer.

The object generates a frames acquired event immediately after the number of frames specified by `FramesAcquiredFcnCount` is acquired from the selected video source.

Data Types: `double`

`StartFcn` — Callback function to execute when start event occurs
`[]` (default) | character vector | function handle | cell array

Callback function to execute when a start event occurs, specified as a character vector, function handle, or cell array. A start event occurs immediately after you issue the `start` command.

The `StartFcn` callback executes synchronously. The toolbox does not set the object's `Running` property to `"on"` until the callback function finishes executing. If the callback function encounters an error, the object never starts running.

Start event information is stored in the `EventLog` property.

Data Types: `char` | `string` | `cell` | `function_handle`

`StopFcn` — Callback function to execute when stop event occurs
`[]` (default) | character vector | function handle | cell array

Callback function to execute when a stop event occurs, specified as a character vector, function handle, or cell array. A stop event occurs immediately after you issue the `stop` command.

The `StopFcn` callback executes synchronously. Under most circumstances, the image acquisition object will be stopped and the `Running` property will be set to `"off"` by the time the MATLAB file completes execution.

Stop event information is stored in the `EventLog` property.

Data Types: `char` | `string` | `cell` | `function_handle`

`TimerFcn` — Callback function to execute when timer event occurs
`[]` (default) | character vector | function handle | cell array

Callback function to execute when a timer event occurs, specified as a character vector, function handle, or cell array. A timer event occurs when the time period specified by the `TimerPeriod` property expires.

The toolbox measures time relative to when the object is started with the `start` function. Timer events stop being generated when the image acquisition object stops running.

Note Some timer events might not be processed if your system is significantly slowed or if the `TimerPeriod` value you specify is too small.

Data Types: `char` | `string` | `cell` | `function_handle`

`TimerPeriod` — Number of seconds between timer events
1 (default) | positive value greater than 0.01

Amount of time, in seconds, that must pass before a timer event is triggered, specified as a positive value greater than 0.01.

The toolbox measures time relative to when the object is started with the `start` function. Timer events stop being generated when the image acquisition object stops running.

Note Some timer events might not be processed if your system is significantly slowed or if the `TimerPeriod` value you specify is too small.

Data Types: `double`

`TriggerFcn` — Callback function to execute when trigger event occurs
`[]` (default) | character vector | function handle | cell array

Callback function to execute when a trigger event occurs, specified as a character vector, function handle, or cell array. The toolbox generates a trigger event when a trigger is executed based on the configured `TriggerType`, and data logging is initiated.

Under most circumstances, the MATLAB file callback function is not guaranteed to complete execution until sometime after the toolbox sets the `Logging` property to `"on"`.

Trigger event information is stored in the `EventLog` property.

Data Types: `char` | `string` | `cell` | `function_handle`

Trigger Properties

`TriggerType` — Type of trigger used by video input object
`"immediate"` (default) | `"hardware"` | `"manual"`

This property is read-only.

Type of trigger used by the video input object, specified as `"immediate"`, `"hardware"`, or `"manual"`. Triggers initiate data acquisition.

You use the `triggerconfig` function to specify one of the following values for this property.

TriggerType Value	Description
`"hardware"` (if available for your device)	Trigger executes when a specified condition is met. You specify the condition using the `TriggerCondition` property and you specify the hardware source to monitor for the condition in the `TriggerSource` property. You use the `triggerconfig` function to set the values of these properties.
`"immediate"`	Trigger executes immediately after you call the `start` function.
`"manual"`	Trigger executes immediately after you call the `trigger` function.

Data Types: `char`

`TriggerCondition` — Required condition before trigger event occurs
character vector

This property is read-only.

Condition that must be met, via the `TriggerSource`, before a trigger event occurs, specified as a character vector. The trigger conditions that you can specify depend on the value of the `TriggerType` property.

TriggerType Value	Conditions Available
`"hardware"` (if available for your device)	Device-specific. For example, some Matrox hardware supports conditions such as `"risingEdge"` and `"fallingEdge"`. Use the `triggerinfo` function to view a list of valid values to use with your image acquisition hardware.
`"immediate"`	`"none"`
`"manual"`	`"none"`

You must use the `triggerconfig` function to set the value of this property.

Data Types: `char`

`TriggerSource` — Hardware source to monitor for trigger conditions
character vector

This property is read-only.

Hardware source the image acquisition object monitors for trigger conditions, specified as a character vector. When the condition specified in the `TriggerCondition` property is met, the object executes the trigger and starts acquiring data.

You use the `triggerconfig` function to specify this value. The value of the `TriggerSource` property is device specific. You specify whatever mechanism a particular device uses to generate triggers.

For example, for Matrox hardware, the `TriggerSource` property could have values such as `"Port0"` or `"Port1"`. Use the `triggerinfo` function to view a list of values that are valid for your image acquisition device.

You must use the `triggerconfig` function to set the value of this property.

Note The `TriggerSource` property is only used when the `TriggerType` property is set to `"hardware"`.

Data Types: `char`

`FramesPerTrigger` — Number of frames to acquire per trigger using selected video source
`10` (default) | positive integer

Number of frames the video input object acquires each time it executes a trigger using the selected video source, specified as a positive integer.

When the value of the `FramesPerTrigger` property is set to `Inf`, the object keeps acquiring frames until an error occurs or you issue a `stop` command.

Note When the `FramesPerTrigger` property is set to `Inf`, the object ignores the value of the `TriggerRepeat` property.

Data Types: `double`

`InitialTriggerTime` — Absolute time of first trigger
`[]` (default) | MATLAB clock vector

This property is read-only.

Absolute time of the first trigger, returned as a MATLAB clock vector.

For all trigger types, `InitialTriggerTime` records the time when the `Logging` property is set to `"on"`.

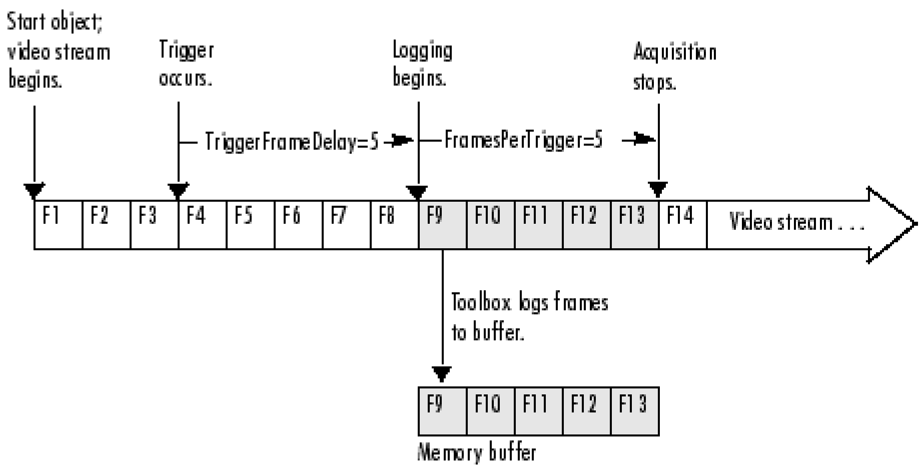
To find the time when a subsequent trigger executed, view the `Data.AbsTime` field of the `EventLog` property for the particular trigger.

Data Types: `double`

`TriggerFrameDelay` — Number of frames to skip before acquiring frames after trigger occurs
0 (default) | integer

Number of frames to skip before acquiring frames after a trigger occurs, specified as an integer. The object waits the specified number of frames after the trigger before starting to log frames.

In this figure, the `TriggerFrameDelay` is set to `5`, so the object lets five frames pass before starting to acquire frames. The number of frames captured is defined by the `FramesPerTrigger` property.



Data Types: `double`

`TriggerRepeat` — Number of additional times to execute trigger
0 (default) | nonnegative integer

Number of additional times you want the object to execute a trigger, specified as a nonnegative integer. This table describes the behavior for several typical `TriggerRepeat` values.

Value	Behavior
`0` (default)	Execute the trigger once when the trigger condition is met.
Any positive integer	Execute the trigger the specified number of additional times when the trigger condition is met.
`Inf`	Keep executing the trigger every time the trigger condition is met until the `stop` function is called or an error occurs.

To determine how many triggers have executed, check the value of the `TriggersExecuted` property.

Note If the `FramesPerTrigger` property is set to `Inf`, the object ignores the value of the `TriggerRepeat` property.

Data Types: `double`

`TriggersExecuted` — Total number of executed triggers
0 (default) | nonnegative integer

This property is read-only.

Total number of triggers that the video input object has executed, specified as a nonnegative integer.

Data Types: `double`

Video Source Object Properties

`Parent` — Video input object that is parent of video source object
`videoinput` object

This property is read-only.

Video input object that is the parent of a video source object, specified as a `videoinput` object.

The parent of a video source object is defined as the video input object owning the video source object.

`Selected` — Whether video source object will be used for acquisition
`"off"` (default) | `"on"`

This property is read-only.

Whether the video source object will be used for acquisition, specified as `"off"` or `"on"`. You select a video source object by specifying its name as the value of the video input object's `SelectedSourceName` property. The video input object `Source` property is an array of all the video source objects associated with the video input object.

If `Selected` is `"on"`, the video source object is selected. If the value is `"off"`, the video source object is not selected.

A video source is defined to be a collection of one or more physical data sources that are treated as a single entity. For example, hardware supporting multiple RGB sources, each of which is made up of three physical connections (red, green, and blue), is treated as a single video source object.

Data Types: `char`

`SourceName` — Name of video source object
character vector

This property is read-only.

Name of a video source object, specified as a character vector.

`SourceName` is one of the values in the video input object's `SelectedSourceName` property.

Data Types: `char`

`Tag` — Descriptive text to associate with image acquisition object
character vector | string scalar

Descriptive text that you want to associate with an image acquisition object, specified as a character vector or string scalar.

The `Tag` property can be useful when you are constructing programs that would otherwise need to define the image acquisition object as a global variable, or pass the object as an argument between callback routines.

You can use the value of the `Tag` property to search for particular image acquisition objects when using the `imaqfind` function.

Data Types: `char` | `string`

`Type` — Type of image acquisition object
`"videoinput"` | `"videosource"`

This property is read-only.

Type of image acquisition object, specified as `"videoinput"` or `"videosource"`. An image acquisition object can be either one of two types:

- Video input object
- Video source object

Data Types: `char`

Acquisition Source Properties

`SelectedSourceName` — Name of currently selected video source
character vector

Name of the video source object from which the video input object acquires data, specified as a character vector. By default, the video input object selects the first available video source object stored in the `Source` property.

The toolbox defines a video source as one or more hardware inputs that are treated as a single entity. For example, hardware supporting multiple RGB sources, each of which is made up of three physical connections (red-green-blue), is treated as a single video source object.

Data Types: `char` | `string`

`Source` — Video source objects associated with video input object
video source object

This property is read-only.

Vector of video source objects that represent the physical data sources connected to a device. When a video input object is created, the toolbox creates a vector of video source objects associated with the video input object.

Each video source object created is provided a unique source name. You can use the source name to select the desired acquisition source by configuring the `SelectedSourceName` property of the video input object.

A video source object's name is stored in its `SourceName` property. If a video source object's `SourceName` is equivalent to the video input object's `SelectedSourceName`, the video source object's `Selected` property has a value of `"on"`.

The video source object supports a set of common properties, such as `SourceName`. Each video source object can also support device-specific properties that control characteristics of the physical device such as brightness, hue, and saturation. Different image acquisition devices expose different sets of properties.

A video source is defined to be a collection of one or more physical data sources that are treated as a single entity. For example, hardware supporting multiple RGB sources, each of which is made up of three physical connections (red-green-blue), is treated as a single video source object.

The `Source` property encapsulates one or more video sources. To reference a video source, you use a numerical integer to index into the vector of video source objects.

GigE Properties

`IgnoreDroppedFrames` — Whether the acquisition continues when it encounters a dropped frame
`"off"` (default) | `"on"`

Whether the acquisition continues when it encounters a dropped frame, specified as `"off"` or `"on"`.

If this property is set to `"off"`, the acquisition stops when it encounters a dropped frame. If this property is set to `"on"`, the acquisition continues by ignoring the dropped frames.

When this property is `"on"`, keep track of the number of frames dropped while the acquisition is running with the `NumDroppedFrames` property.

Note This property is only supported on `videoinput` objects using the `gige` adaptor and is not supported on `gigecam` objects.

Data Types: `char` | `string`

`NumDroppedFrames` — Number of frames dropped while acquisition is running
0 (default) | nonnegative integer

This property is read-only.

Number of frames dropped while the acquisition is running if the `IgnoreDroppedFrames` property is set to `'on'`, specified as a nonnegative integer.

This property does not keep track of the number of frames dropped while previewing.

Note This property is only supported on `videoinput` objects using the `gige` adaptor and is not supported on `gigecam` objects.

Data Types: `double`

Object Functions

Configuration

get	Return image acquisition object properties
set	Configure or display image acquisition object properties

Execution

getselectedsource	Return currently selected video source object
start	Obtain exclusive use of image acquisition device
stop	Stop video input object
wait	Wait until image acquisition object stops running or logging

Trigger Functions

trigger	Initiate data logging
triggerconfig	Configure video input object trigger properties
triggerinfo	Provide information about available trigger configurations

Data Functions

flushdata	Remove data from memory buffer used to store acquired image frames
getdata	Acquired image frames to MATLAB workspace
getsnapshot	Immediately return single image frame
peekdata	Most recently acquired image data

Tools

closepreview	Close Video Preview window
imaqmontage	Sequence of image frames as montage
preview	Preview of live video data

General

delete	Remove image acquisition object from memory
imaqfind	Find image acquisition objects
imaqreset	Disconnect and delete all image acquisition objects
isvalid	Determine whether image acquisition object is associated with image acquisition device

Information and Help

imaqhelp	Image acquisition object function and property help
imaqhwinfo	Information about available image acquisition hardware
propinfo	Property characteristics for image acquisition objects

Examples

Create Video Input Object

Construct a video input object.

```
obj = videoinput("matrox",1);
```

Select the source to use for acquisition.

```
obj.SelectedSourceName = "input1"
```

View the properties for the selected video source object.

```
src_obj = getselectedsource(obj);
get(src_obj)
```

Preview a stream of image frames.

```
preview(obj);
```

Acquire and display a single image frame.

```
frame = getsnapshot(obj);
image(frame);
```

Remove video input object from memory.

```
delete(obj);
```

Use `VideoWriter`

Create a video input object that accesses a GigE Vision image acquisition device and uses grayscale format at 10 bits per pixel.

```
vidobj = videoinput("gige",1,"Mono10");
```

You can log acquired data to memory, to disk, or both. By default, data is logged to memory. To change the logging mode to disk, configure the video input object's `LoggingMode` property.

```
vidobj.LoggingMode = "disk"
```

Create a `VideoWriter` object with the profile set to Motion JPEG 2000. Motion JPEG 2000 allows writing the full 10 bits per pixel data to the file.

```
vidobj.DiskLogger = VideoWriter("logfile.mj2","Motion JPEG 2000")
```

Now that the video input object is configured for logging data to a Motion JPEG 2000 file, initiate the acquisition.

```
start(vidobj)
```

Wait for the acquisition to finish.

```
wait(vidobj)
```

When logging large amounts of data to disk, disk writing occasionally lags behind the acquisition. To determine whether all frames are written to disk, you can optionally use the `DiskLoggerFrameCount` property.

```
while (vidobj.FramesAcquired ~= vidobj.DiskLoggerFrameCount)
    pause(.1)
end
```

You can verify that the `FramesAcquired` and `DiskLoggerFrameCount` properties have identical values by using these commands and comparing the output.

```
vidobj.FramesAcquired
vidobj.DiskLoggerFrameCount
```

When the video input object is no longer needed, delete it and clear it from the workspace.

```
delete(vidobj)
clear vidobj
```

Version History

Introduced before R2006a

wait

Wait until image acquisition object stops running or logging

Syntax

```
wait(obj)
wait(obj,waittime)
wait(obj,waittime,state)
```

Description

`wait(obj)` blocks the MATLAB command line until the video input object `obj` stops running (`Running` = `'off'`). `obj` can be either a single video input object or an array of video input objects. When `obj` is an array of objects, the `wait` function waits until all objects in the array stop running. If `obj` is not running or is an invalid object, `wait` returns immediately. The `wait` function can be useful when you want to guarantee that data is acquired before another task is performed.

`wait(obj,waittime)` blocks the MATLAB command line until the video input object or array of objects `obj` stops running or until `waittime` seconds have expired, whichever comes first. By default, `waittime` is set to the value of the object's `Timeout` property.

`wait(obj,waittime,`*`state`*`)` blocks the MATLAB command line until the video input object or array of objects `obj` stops running or logging, or until `waittime` seconds have expired, whichever comes first. `state` can be either of the following character vectors. The default value is enclosed in braces ({}).

State	Description
`{'running'}`	Blocks until the value of the object's `Running` property is `'off'`.
`'logging'`	Blocks until the value of the object's `Logging` property is `'off'`.

Note The video input object's stop event callback function (`StopFcn`) might not be called before this function returns.

An image acquisition object stops running or logging when one of the following conditions is met:

- The `stop` function is issued.
- The requested number of frames is acquired. This occurs when

  ```
  FramesAcquired = FramesPerTrigger * (TriggerRepeat + 1)
  ```

  where `FramesAcquired`, `FramesPerTrigger`, and `TriggerRepeat` are properties of the video input object.
- A run-time error occurs.
- The object's `Timeout` value is reached.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For

information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 5-17.

Examples

Create a video input object.

```
vid = videoinput('winvideo');
```

Specify an acquisition that should take several seconds. The example sets the `FramesPerTrigger` property to 300.

```
vid.FramesPerTrigger = 300;
```

Start the object. Because it is configured with an immediate trigger (the default), acquisition begins when the object is started. The example calls the `wait` function after calling the `start` function. Notice how `wait` blocks the MATLAB command line until the acquisition is complete.

```
start(vid), wait(vid);
```

Version History

Introduced before R2006a

See Also

`imaqhelp` | `start` | `stop` | `trigger` | `propinfo`

19

Properties

BayerSensorAlignment

Specify sensor alignment for Bayer demosaicing

Description

If the `ReturnedColorSpace` property is set to `'bayer'`, then the Image Acquisition Toolbox software will demosaic Bayer patterns returned by the hardware. This color space setting will interpolate Bayer pattern encoded images into standard RGB images. If your camera uses Bayer filtering, the toolbox supports the Bayer pattern and can return color if desired.

In order to perform the demosaicing, the toolbox needs to know the pixel alignment of the sensor. This is the order of the red, green, and blue sensors and is normally specified by describing the four pixels in the upper-left corner of the sensor. It is the band sensitivity alignment of the pixels as interpreted by the camera's internal hardware. You must get this information from the camera's documentation and then specify the value for the alignment, as described in the following table.

There are four possible sensor alignments.

Value	Description

`'gbrg'`	The 2-by-2 sensor alignment is
	
	`green blue`
	`red   green`

`'grbg'`	The 2-by-2 sensor alignment is
	
	`green red`
	`blue  green`

`'bggr'`	The 2-by-2 sensor alignment is
	
	`blue  green`
	`green red`

`'rggb'`	The 2-by-2 sensor alignment is
	
	`red   green`
	`green blue`

The value of this property is only used if the `ReturnedColorSpace` property is set to `'bayer'`.

For examples showing how to convert Bayer images, see “Converting Bayer Images” on page 7-15.

Characteristics

Access	Read/write
Data type	Character vector
Values	`[ {'grbg'} | 'gbrg' | 'rggb'| 'bggr']`

See Also

Functions

`getdata`, `getsnapshot`, `peekdata`, `videoinput`

Properties

ReturnedColorSpace, VideoFormat

How To's

"Specifying the Color Space" on page 7-14

"Converting Bayer Images" on page 7-15

DeviceID

Identify image acquisition device represented by video input object

Description

The `DeviceID` property identifies the device represented by the video input object.

A device ID is a number, assigned by an adaptor, that uniquely identifies an image acquisition device. The adaptor assigns the first device it detects the identifier `1`, the second device it detects the identifier `2`, and so on.

You must specify the device ID as an argument to the `videoinput` function when you create a video input object. The object stores the value in the `DeviceID` property and also uses the value when constructing the default value of the `Name` property.

To get a list of the IDs of the devices connected to your system, use the `imaqhwinfo` function, specifying the name of a particular adaptor as an argument.

Characteristics

Access	Read only
Data type	`double`
Values	Any nonnegative integer

Examples

Use the `imaqhwinfo` function to determine which adaptors are connected to devices on your system.

```
imaqhwinfo

ans =

    InstalledAdaptors: {'matrox'  'winvideo'}
        MATLABVersion: '7.4 (R2007a)'
          ToolboxName: 'Image Acquisition Toolbox'
       ToolboxVersion: '2.1 (R2007a)'
```

Use the `imaqhwinfo` function again, specifying the name of the adaptor, to find out how many devices are available through that adaptor. The `imaqhwinfo` function returns the device IDs for all the devices in the `DeviceIds` field.

```
info = imaqhwinfo('winvideo')

info =

       AdaptorDllName: [1x73 char]
    AdaptorDllVersion: '2.0 (R2006a+)'
          AdaptorName: 'winvideo'
            DeviceIDs: {[1]}
           DeviceInfo: [1x1 struct]
```

See Also

Functions

`imaqhwinfo`, `videoinput`

Properties

Name

DiskLogger

Specify MATLAB VideoWriter file used to log data

Description

The `DiskLogger` property specifies the VideoWriter file object used to log data when the `LoggingMode` property is set to `'disk'` or `'disk&memory'`. For the best performance, VideoWriter is the recommended file type.

VideoWriter File

For the best performance, logging to disk requires a MATLAB `VideoWriter` object, which is a MATLAB object, not an Image Acquisition Toolbox object. After you create and configure a VideoWriter object, you provide it to the `DiskLogger` property.

A MATLAB VideoWriter object specifies the file name and other characteristics. For example, you can use VideoWriter properties to specify the profile used for data compression and the desired quality of the output. For complete information about the VideoWriter object and its properties, see the `VideoWriter` documentation.

Note Do not use the variable returned by the `VideoWriter` function to perform any operation on a VideoWriter file while it is being used by a video input object for data logging. For example, do not change any of the VideoWriter file properties, add frames, or close the object. Your changes could conflict with the video input object.

After `Logging` and `Running` are off, it is possible that the `DiskLogger` might still be writing data to disk. When the `DiskLogger` finishes writing data to disk, the value of the `DiskLoggerFrameCount` property should equal the value of the `FramesAcquired` property. Do not close or modify the `DiskLogger` until this condition is met.

For more information about logging image data using a VideoWriter file, see “Logging Image Data to Disk” on page 6-32.

Note The `peekdata` function does not return any data while running if in disk logging mode.

Characteristics

Access	Read only while running
Data type	VideoWriter object
Values	The default value is `[]`.

Examples

Using VideoWriter

Create a video input object that accesses a GigE Vision image acquisition device and uses grayscale format at 10 bits per pixel.

```
vidobj = videoinput('gige', 1, 'Mono10');
```

You can log acquired data to memory, to disk, or both. By default, data is logged to memory. To change the logging mode to disk, configure the video input object's `LoggingMode` property.

```
vidobj.LoggingMode = 'disk'
```

Create a VideoWriter object with the profile set to Motion JPEG 2000. Motion JPEG 2000 allows writing the full 10 bits per pixel data to the file.

```
vidobj.DiskLogger = VideoWriter('logfile.mj2', 'Motion JPEG 2000')
```

Now that the video input object is configured for logging data to a Motion JPEG 2000 file, initiate the acquisition.

```
start(vidobj)
```

Wait for the acquisition to finish.

```
wait(vidobj)
```

When logging large amounts of data to disk, disk writing occasionally lags behind the acquisition. To determine whether all frames are written to disk, you can optionally use the `DiskLoggerFrameCount` property.

```
while (vidobj.FramesAcquired ~= vidobj.DiskLoggerFrameCount)
    pause(.1)
end
```

You can verify that the `FramesAcquired` and `DiskLoggerFrameCount` properties have identical values by using these commands and comparing the output.

```
vidobj.FramesAcquired
vidobj.DiskLoggerFrameCount
```

When the video input object is no longer needed, delete it and clear it from the workspace.

```
delete(vidobj)
clear vidobj
```

See Also

Functions

`videoinput`

Properties

DiskLoggerFrameCount, Logging, LoggingMode

DiskLoggerFrameCount

Specify number of frames written to disk

Description

The `DiskLoggerFrameCount` property indicates the current number of frames written to disk by the `DiskLogger`. This value is only updated when the `LoggingMode` property is set to `'disk'` or `'disk&memory'`.

After `Logging` and `Running` are off, it is possible that the `DiskLogger` might still be writing data to disk. When the `DiskLogger` finishes writing data to disk, the value of the `DiskLoggerFrameCount` property should equal the value of the `FramesAcquired` property. Do not close or modify the `DiskLogger` until this condition is met.

Characteristics

Access	Read only
Data type	`double`
Values	Any nonnegative integer

See Also

Functions

`videoinput`

Properties

DiskLogger, FramesAcquired, Logging, Running

ErrorFcn

Specify callback function to execute when run-time error occurs

Description

The `ErrorFcn` property specifies the function to execute when an error event occurs. A run-time error event is generated immediately after a run-time error occurs.

Run-time errors include hardware errors and timeouts. Run-time errors do not include configuration errors such as setting an invalid property value.

Run-time error event information is stored in the `EventLog` property. You can retrieve any error message with the `Data.Message` field of `EventLog`.

Note Callbacks, including `ErrorFcn`, are executed only when the video object is in a running state. If you need to use the `ErrorFcn` callback for error handling during previewing, you must start the video object before previewing. To do that without logging data, use a manual trigger.

Characteristics

Access	Read only while running
Data type	Character vector, function handle, or cell array
Values	`imaqcallback` is the default callback function.

See Also

Properties

EventLog, Timeout

EventLog

Store information about events

Description

The `EventLog` property is an array of structures that stores information about events. Each structure in the array represents one event. Events are recorded in the order in which they occur. The first `EventLog` structure reflects the first event recorded, the second `EventLog` structure reflects the second event recorded, and so on.

Each event log structure contains two fields: `Type` and `Data`.

The `Type` field stores a character array that identifies the event type. The Image Acquisition Toolbox software defines many different event types, listed in this table. Note that not all event types are logged.

Event Type	Description	Included in Log
Error	Run-time error occurred. Run-time errors include timeouts and hardware errors.	Yes
Frames Acquired	The number of frames specified in the `FramesAcquiredFcnCount` property has been acquired.	No
Start	Object was started by calling the `start` function.	Yes
Stop	Object stopped executing.	Yes
Timer	Timer expired.	No
Trigger	Trigger executed.	Yes

The `Data` field stores information associated with the specific event. For example, all events return the absolute time the event occurred in the `AbsTime` field. Other event-specific fields are included in `Data`. For more information, see “Retrieving Event Information” on page 8-7.

`EventLog` can store a maximum of 1000 events. If this value is exceeded, then the most recent 1000 events are stored.

Characteristics

Access	Read only
Data type	Structure array
Values	Default is empty structure array.

Examples

Create a video input object.

```
vid = videoinput('winvideo');
```

Start the object.

```
start(vid)
```

View the event log to see which events occurred.

```
elog = vid.EventLog;

{elog.Type}

ans =
    'Start'    'Trigger'    'Stop'
```

View the data associated with a trigger event.

```
elog(2).Data
ans =

             AbsTime: [2003 2 11 17 22 18.9740]
         FrameNumber: 0
       RelativeFrame: 0
        TriggerIndex: 1
```

See Also

Properties

Logging

FrameGrabInterval

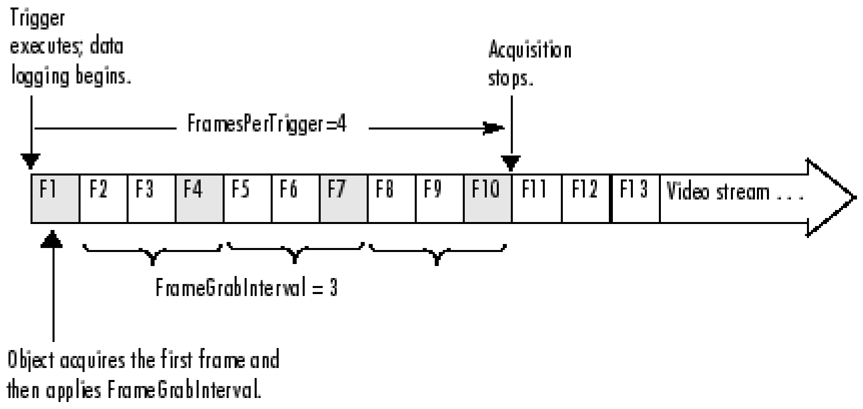
Specify how often to acquire frame from video stream

Description

The `FrameGrabInterval` property specifies how often the video input object acquires a frame from the video stream. By default, objects acquire every frame in the video stream, but you can use this property to specify other acquisition intervals.

Note Do not confuse the frame grab interval with the frame rate. The frame rate describes the rate at which an image acquisition device provides frames, typically measured in seconds, such as 30 frames per second. The frame grab interval is measured in frames, not seconds. If a particular device's frame rate is configurable, the video source object might include the frame rate as a device-specific property.

For example, when you specify a `FrameGrabInterval` value of `3`, the object acquires every third frame from the video stream, as illustrated in this figure. The object acquires the first frame in the video stream before applying the `FrameGrabInterval`.


Trigger executes; data logging begins.
Acquisition stops.
FramesPerTrigger=4
F1
F2
F3
F4
F5
F6
F7
F8
F9
F10
F11
F12
F13
Video stream . . .
FrameGrabInterval = 3
Object acquires the first frame and then applies FrameGrabInterval.

You specify the source of the video stream in the `SelectedSourceName` property.

Characteristics

Access	Read only while running
Data type	`double`
Values	Any positive integer. The default value is `1` (acquire every frame).

See Also

Functions

`videoinput`

Properties

SelectedSourceName

FramesAcquired

Indicate total number of frames acquired

Description

The `FramesAcquired` property indicates the total number of frames that the object has acquired, regardless of how many frames have been extracted from the memory buffer. The video input object continuously updates the value of the `FramesAcquired` property as it acquires frames.

Note When you issue a `start` command, the video input object resets the value of the `FramesAcquired` property to `0` (zero) and flushes the buffer.

To find out how many frames are available in the memory buffer, use the `FramesAvailable` property.

Characteristics

Access	Read only
Data type	`double`
Values	Any nonnegative integer. The default value is `0` (zero).

See Also

Functions

`start`

Properties

FramesAvailable, FramesAcquiredFcn, FramesAcquiredFcnCount

FramesAcquiredFcn

Specify MATLAB file executed when specified number of frames have been acquired

Description

The `FramesAcquiredFcn` specifies the MATLAB file function to execute every time a predefined number of frames have been acquired.

A frames acquired event is generated immediately after the number of frames specified by the `FramesAcquiredFcnCount` property is acquired from the selected video source. This event executes the MATLAB file specified for `FramesAcquiredFcn`.

Use the `FramesAcquiredFcn` callback if you must access each frame that is acquired. If you do not have this requirement, you might want to use the `TimerFcn` property.

Frames acquired event information is not stored in the `EventLog` property.

Characteristics

Access	Read/write
Data type	Character vector, function handle, or cell array
Values	The default value is an empty matrix (`[]`).

See Also

Properties

EventLog, FramesAcquiredFcnCount, TimerFcn

FramesAcquiredFcnCount

Specify number of frames that must be acquired before frames acquired event is generated

Description

The `FramesAcquiredFcnCount` property specifies the number of frames to acquire from the selected video source before a frames acquired event is generated.

The object generates a frames acquired event immediately after the number of frames specified by `FramesAcquiredFcnCount` is acquired from the selected video source.

Characteristics

Access	Read only while running
Data type	`double`
Values	Any positive integer. The default value is `0` (zero).

See Also

Properties

FramesAcquiredFcn

FramesAvailable

Indicate number of frames available in memory buffer

Description

The `FramesAvailable` property indicates the total number of frames that are available in the memory buffer. When you extract data, the object reduces the value of the `FramesAvailable` property by the appropriate number of frames. You use the `getdata` function to extract data and move it into the MATLAB workspace.

Note When you issue a `start` command, the video input object resets the value of the `FramesAvailable` property to `0` (zero) and flushes the buffer.

To view the total number of frames that have been acquired since the last `start` command, use the `FramesAcquired` property.

Characteristics

Access	Read only
Data type	`double`
Values	Any nonnegative integer. The default value is `0` (zero).

See Also

Functions

`getdata`, `start`

Properties

FramesAcquired

FramesPerTrigger

Specify number of frames to acquire per trigger using selected video source

Description

The `FramesPerTrigger` property specifies the number of frames the video input object acquires each time it executes a trigger using the selected video source.

When the value of the `FramesPerTrigger` property is set to `Inf`, the object keeps acquiring frames until an error occurs or you issue a `stop` command.

Note When the `FramesPerTrigger` property is set to `Inf`, the object ignores the value of the `TriggerRepeat` property.

Characteristics

Access	Read only while running
Data type	`double`
Values	Any positive integer. The default value is `10`.

See Also

Functions

`stop`

Properties

TriggerRepeat

IgnoreDroppedFrames

Continue acquiring frames when acquisition drops a frame

Description

Note This property is only supported on `videoinput` objects using the `gige` adaptor and is not supported on `gigecam` objects.

The `IgnoreDroppedFrames` property indicates whether the acquisition continues when it encounters a dropped frame.

If this property is set to `'off'`, the acquisition stops when it encounters a dropped frame. If this property is set to `'on'`, the acquisition continues by ignoring the dropped frames.

When this property is `'on'`, keep track of the number of frames dropped while the acquisition is running with the `NumDroppedFrames` property.

Characteristics

Default value is enclosed in braces ({}).

Access	Read/write
Data type	Character vector
Values	`[ {'off'} | 'on' ]`

See Also

NumDroppedFrames

InitialTriggerTime

Record absolute time of first trigger

Description

The `InitialTriggerTime` property records the absolute time of the first trigger. The absolute time is recorded as a MATLAB clock vector.

For all trigger types, `InitialTriggerTime` records the time when the `Logging` property is set to `'on'`.

To find the time when a subsequent trigger executed, view the `Data.AbsTime` field of the `EventLog` property for the particular trigger.

Characteristics

Access	Read only
Data type	Six-element vector of doubles (MATLAB clock vector)
Values	The default value is `[]`.

Examples

Create an image acquisition object, `vid`, for a USB-based webcam.

```
vid = videoinput('winvideo',1);
```

Start the object. Because the `TriggerType` property is set to `'immediate'` by default, the trigger executes immediately. The object records the time of the initial trigger.

```
start(vid)

abstime = vid.InitialTriggerTime

abstime =

   1.0e+003 *

      1.9990    0.0020    0.0190    0.0130    0.0260    0.0208
```

Convert the clock vector into an integer form for display.

```
t = fix(abstime);

sprintf('%d:%d:%d', t(4),t(5),t(6))

ans =

13:26:20
```

See Also

Functions

`getdata`

Properties

EventLog, Logging, TriggerType

Logging

Indicate whether object is currently logging data

Description

The `Logging` property indicates whether the video input object is currently logging data.

When a trigger occurs, the object sets the `Logging` property to `'on'` and logs data to memory, a disk file, or both, depending on the value of the `LoggingMode` property.

The object sets the `Logging` property to `'off'` when it acquires the requested number of frames, an error occurs, or you issue a `stop` command.

To acquire data when the object is running but not logging, use the `peekdata` function. Note, however, that the `peekdata` function does not guarantee that all the requested image data is returned. To acquire all the data without gaps, you must have the object log the data to memory or to a disk file.

Characteristics

Default value is enclosed in braces (`{}`).

Access	Read only
Data type	Character vector
Values	`[ {'off'} | 'on' ]`

See Also

Functions

`getdata`, `islogging`, `peekdata`, `stop`, `trigger`

Properties

LoggingMode, Running

LoggingMode

Specify destination for acquired data

Description

The `LoggingMode` property specifies where you want the video input object to store the acquired data. You can specify any of the following values:

Value	Description
`'disk'`	Log acquired data to a disk file.
`'disk&memory'`	Log acquired data to both a disk file and to a memory buffer.
`'memory'`	Log acquired data to a memory buffer.

If you select `'disk'` or `'disk&memory'`, you must specify the AVI file object used to access the disk file as the value of the `DiskLogger` property.

Note When logging data to memory, you must extract the acquired data in a timely manner with the `getdata` function to avoid using up all the memory that is available on your system.

Note The `peekdata` function does not return any data while running if in disk logging mode.

Characteristics

Access	Read only while running
Data type	Character vector
Values	`[ 'disk' | 'disk&memory' | {'memory'} ]`

Default value is enclosed in braces (`{}`).

See Also

Functions

`getdata`

Properties

DiskLogger, Logging

Name

Specify name of image acquisition object

Description

The `Name` property specifies a descriptive name for the image acquisition object.

Characteristics

Access	Read/write
Data type	Character vector
Values	Any character vector. The toolbox creates the default name by combining the values of the `VideoFormat` and `DeviceID` properties with the adaptor name in this format: `VideoFormat` + `'-'` + adaptor name + `'-'` + `DeviceID`

Examples

Create an image acquisition object.

```
vid = videoinput('winvideo');
```

Retrieve the value of the `Name` property.

```
vid.Name
```

```
ans =

     RGB555_128x96-winvideo-1
```

See Also

Functions

`videoinput`

Properties

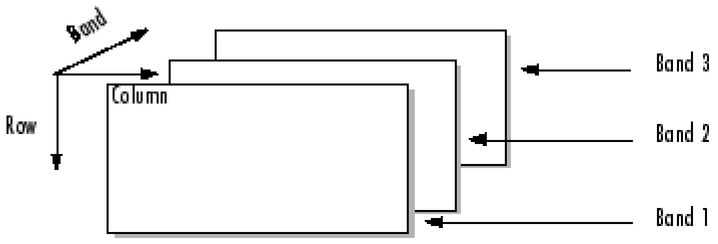
DeviceID, VideoFormat

NumberOfBands

Indicate number of color bands in data to be acquired

Description

The `NumberOfBands` property indicates the number of color bands in the data to be acquired. The toolbox defines *band* as the third dimension in a 3-D array, as shown in this figure.



The value of the `NumberOfBands` property indicates the number of color bands in the data returned by `getsnapshot`, `getdata`, and `peekdata`.

Characteristics

Access	Read only
Data type	`double`
Values	Any positive integer. The default value is defined at object creation time based on the video format.

Examples

Create an image acquisition object.

```
vid = videoinput('winvideo');
```

Retrieve the value of the `NumberOfBands` property.

```
vid.NumberOfBands

ans =

     3
```

If you retrieve the value of the `VideoFormat` property, you can see that the video data is in RGB format.

```
vid.VideoFormat

ans =

RGB24_320x240
```

See Also

Functions

`getdata`, `getsnapshot`, `peekdata`

NumDroppedFrames

Number of frames dropped while acquisition is running

Description

Note This property is only supported on `videoinput` objects using the `gige` adaptor and is not supported on `gigecam` objects.

The `NumDroppedFrames` property indicates the number of frames dropped while the acquisition is running if the `IgnoreDroppedFrames` property is set to `'on'`.

This property does not keep track of the number of frames dropped while previewing.

Characteristics

Access	Read only
Data type	`double`
Values	Any nonnegative integer. The default value is `0` (zero).

See Also

IgnoreDroppedFrames

Parent

Identify video input object that is parent of video source object

Description

The `Parent` property identifies the video input object that is the parent of a video source object.

The parent of a video source object is defined as the video input object owning the video source object.

Characteristics

Access	Read only
Data type	Video input object
Values	Defined at object creation time

See Also

Functions

`videoinput`

PreviewFullBitDepth

Configure preview data to display in full bit depth

Description

The `PreviewFullBitDepth` property indicates whether the image data in the Preview window is being displayed in full bit depth.

Note The Image Acquisition Toolbox Preview window supports the display of up to 16-bit image data. The Preview window was designed to only show 8-bit data, but many cameras return 10-, 12-, 14-, or 16-bit data. The Preview window display supports these higher bit-depth cameras. However, larger bit data is scaled to 8-bit for the purpose of displaying previewed data. To capture the image data in the Preview window in its full bit depth for grayscale images, set the `PreviewFullBitDepth` property to `'on'`.

If you set this property to `'off'`, image data in the preview window is scaled down from its bit depth to 8-bit. If you set this property to `'on'`, the image data in the preview window is being captured in its full bit depth.

This property can be set to `'on'` only when the value of the `ReturnedColorspace` property is set to `'grayscale'` and for video formats higher than 8-bit depth.

Characteristics

Default value is enclosed in braces ({}).

Access	Read/write
Data type	Character vector
Values	`[ {'off'} | 'on' ]`

See Also

`preview` | ReturnedColorSpace

Previewing

Indicate whether object is currently previewing data in separate window

Description

The `Previewing` property indicates whether the object is currently previewing data in a separate window.

The object sets the `Previewing` property to `'on'` when you call the `preview` function.

The object sets the `Previewing` property to `'off'` when you close the preview window using the `closepreview` function or by clicking the **Close** button in the preview window title bar.

Characteristics

Default value is enclosed in braces ({}).

Access	Read only
Data type	Character vector
Values	`[ {'off'} | 'on' ]`

See Also

Functions

`closepreview`, `preview`

ReturnedColorSpace

Specify color space used in MATLAB

Description

The `ReturnedColorSpace` property specifies the color space you want the toolbox to use when it returns image data to the MATLAB workspace. This is only relevant when you are accessing acquired image data with the `getsnapshot`, `getdata`, and `peekdata` functions.

This property can have any of the following values:

Value	**Description**
`'grayscale'`	MATLAB grayscale color space.
`'rgb'`	MATLAB RGB color space.
`'YCbCr'`	MATLAB YCbCr color space. Note that YCbCr is often imprecisely referred to as YUV. (YUV is similar, but not identical. They differ by the scaling factor applied to the result. YUV refers to a particular scaling factor used in composite NTSC and PAL formats. In most cases, you can specify the YCbCr color space for devices that support YUV.)
`'bayer'`	Convert grayscale Bayer color patterns to RGB images. The `bayer` color space option is only available if your camera's default returned color space is `grayscale`. To use the `BayerSensorAlignment` property, you must set the `ReturnedColorSpace` property to `bayer`.

Note For some adaptors, such as GigE and GenTL, if you use a format that starts with Bayer, e.g. `BayerGB8_640x480`, we automatically convert the raw Bayer pattern to color – the `ReturnedColorSpace` is RGB. If you set the `ReturnedColorSpace` to `'grayscale'`, you'll get the raw pattern.

For an example showing how to determine the default color space and change the color space setting, see “Specifying the Color Space” on page 7-14.

Characteristics

Access	Read/write
Data type	Character vector
Values	Defined at object creation time and depends on the video format selected

See Also

Functions

`getdata`, `getsnapshot`, `peekdata`, `videoinput`

Properties

BayerSensorAlignment, VideoFormat

How To's

"Specifying the Color Space" on page 7-14

ROIPosition

Specify region-of-interest (ROI) window

Description

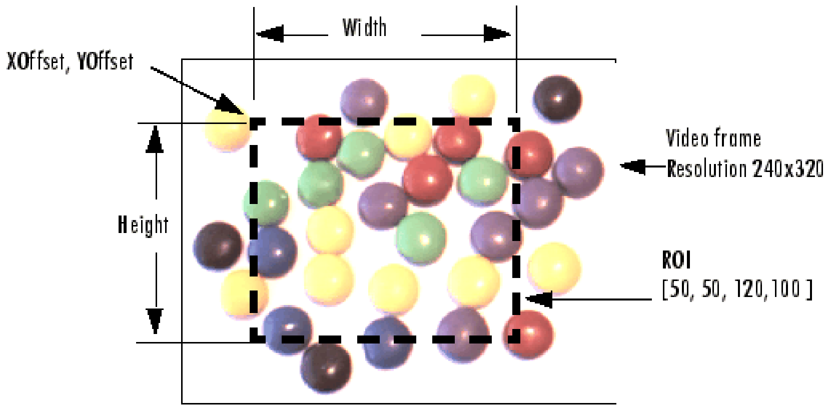
The `ROIPosition` property specifies the region-of-interest acquisition window. The ROI defines the actual size of the frame logged by the toolbox, measured with respect to the top left corner of an image frame.

`ROIPosition` is specified as a 1-by-4 element vector

```
[XOffset YOffset Width Height]
```

where

`XOffset`	Position of the upper left corner of the ROI, measured in pixels.
`YOffset`	Position of the upper left corner of the ROI, measured in pixels.
`Width`	Width of the ROI, measured in pixels. The sum of `XOffset` and `Width` cannot exceed the width specified in `VideoResolution`.
`Height`	Height of the ROI, measured in pixels. The sum of `YOffset` and `Height` cannot exceed the height specified in `VideoResolution`.



Note The `Width` does not include both end points as well as the width between the pixels. It includes one end point, plus the width between pixels. For example, if you want to capture an ROI of pixels 20 through 30, including both end pixels 20 and 30, set an `XOffset` of `19` and a `Width` of `11`. The same rule applies to `height`.

In the figure shown above, the width of the captured ROI contains pixels 51 through 170, including both end points, because the `XOffset` is set to `50` and the `Width` is set to `120`.

Characteristics

Access	Read only while running

Data type	1-by-4 element vector of doubles
Values	Default is `[ 0 0 width height ]` where `width` and `height` are determined by `VideoResolution`.

See Also

Properties

VideoResolution

Running

Indicate whether video input object is ready to acquire data

Description

The `Running` property indicates if the video input object is ready to acquire data.

Along with the `Logging` property, `Running` reflects the state of a video input object. The `Running` property indicates that the object is ready to acquire data, while the `Logging` property indicates that the object is acquiring data.

The object sets the `Running` property to `'on'` when you issue the `start` command. When `Running` is `'on'`, you can acquire data from a video source.

The object sets the `Running` property to `'off'` when any of the following conditions is met:

- The specified number of frames has been acquired.
- A run-time error occurs.
- You issue the `stop` command.

When `Running` is `'off'`, you cannot acquire image data. However, you can acquire one image frame with the `getsnapshot` function.

Characteristics

Default value is enclosed in braces (`{}`).

Access	Read only
Data type	Character vector
Values	`[ {'off'} | 'on' ]`

See Also

Properties

`getsnapshot`, `start`, `stop`

Properties

Logging

Selected

Indicate whether video source object will be used for acquisition

Description

The `Selected` property indicates if the video source object will be used for acquisition. You select a video source object by specifying its name as the value of the video input object's `SelectedSourceName` property. The video input object `Source` property is an array of all the video source objects associated with the video input object.

If `Selected` is `'on'`, the video source object is selected. If the value is `'off'`, the video source object is not selected.

A video source is defined to be a collection of one or more physical data sources that are treated as a single entity. For example, hardware supporting multiple RGB sources, each of which is made up of three physical connections (red, green, and blue), is treated as a single video source object.

Characteristics

Default value is enclosed in braces (`{}`).

Access	Read only
Data type	Character vector
Values	`[ {'off'} | 'on' ]`

Examples

Create an image acquisition object.

```
vid = videoinput('winvideo');
```

Determine the currently selected video source object.

```
vid.SelectedSourceName

ans =

input1
```

Retrieve the currently selected video source object.

```
src = getselectedsource(vid);
```

View its `Name` and `Selected` properties.

```
src.SourceName

ans =

input1
```

```
src.Selected

ans =

on
```

See Also

Functions

`getselectedsource`

Properties

SelectedSourceName

SelectedSourceName

Specify name of currently selected video source

Description

The `SelectedSourceName` property specifies the name of the video source object from which the video input object acquires data. The name is specified as a character vector. By default, the video input object selects the first available video source object stored in the `Source` property.

The toolbox defines a video source as one or more hardware inputs that are treated as a single entity. For example, hardware supporting multiple RGB sources, each of which is made up of three physical connections (red-green-blue), is treated as a single video source object.

Characteristics

Access	Read only while running
Data type	`Character vector`
Values	The video input object assigns a name to each video source object it creates. Names are defined at object creation time and are vendor specific. By default, the toolbox uses the first available source name.

Examples

To see a list of all available sources, create a video input object.

```
vid = videoinput('matrox');
```

View a list of all available video source objects.

```
src_names = vid.SelectedSourceName;
```

See Also

Functions

`set`

Properties

Source

Source

Indicate video source objects associated with video input object

Description

The `Source` property is a vector of video source objects that represent the physical data sources connected to a device. When a video input object is created, the toolbox creates a vector of video source objects associated with the video input object.

Each video source object created is provided a unique source name. You can use the source name to select the desired acquisition source by configuring the `SelectedSourceName` property of the video input object.

A video source object's name is stored in its `SourceName` property. If a video source object's `SourceName` is equivalent to the video input object's `SelectedSourceName`, the video source object's `Selected` property has a value of `'on'`.

The video source object supports a set of common properties, such as `SourceName`. Each video source object can also support device-specific properties that control characteristics of the physical device such as brightness, hue, and saturation. Different image acquisition devices expose different sets of properties.

A video source is defined to be a collection of one or more physical data sources that are treated as a single entity. For example, hardware supporting multiple RGB sources, each of which is made up of three physical connections (red-green-blue), is treated as a single video source object.

The `Source` property encapsulates one or more video sources. To reference a video source, you use a numerical integer to index into the vector of video source objects.

Characteristics

Access	Read only
Data type	Vector of video source objects
Values	Defined at object creation time

Examples

Create an image acquisition object.

```
vid = videoinput('matrox');
```

To access all the video source objects associated with a video input object, use the `Source` property of the video input object. (To view only the currently selected video source object, use the `getselectedsource` function.)

```
sources = vid.Source;
src = sources(1);
```

To view the properties of the video source object `src`, use the `get` function.

```
get(src)
  General Settings:
    Parent = [1x1 videoinput]
    Selected = on
    SourceName = CH1
    Tag =
    Type = videosource

  Device Specific Properties:
    InputFilter = lowpass
    UserOutputBit3 = off
    UserOutputBit4 = off
    XScaleFactor = 1
    YScaleFactor = 1
```

See Also

Functions

`videoinput`, `getselectedsource`

Properties

SelectedSourceName

SourceName

Indicate name of video source object

Description

The `SourceName` property indicates the name of a video source object.

`SourceName` is one of the values in the video input object's `SelectedSourceName` property.

Characteristics

Access	Read only
Data type	Character vector
Values	Defined at object creation time

See Also

Functions

`getselectedsource`

Properties

SelectedSourceName, Source

StartFcn

Specify MATLAB file executed when start event occurs

Description

The `StartFcn` property specifies the MATLAB file function to execute when a start event occurs. A start event occurs immediately after you issue the `start` command.

The `StartFcn` callback executes synchronously. The toolbox does not set the object's `Running` property to `'on'` until the callback function finishes executing. If the callback function encounters an error, the object never starts running.

Start event information is stored in the `EventLog` property.

Characteristics

Access	Read/write
Data type	Character vector, function handle, or cell array
Values	The default value is an empty matrix (`[]`).

See Also

Properties

EventLog, Running

StopFcn

Specify MATLAB file executed when stop event occurs

Description

The `StopFcn` property specifies the MATLAB file function to execute when a stop event occurs. A stop event occurs immediately after you issue the `stop` command.

The `StopFcn` callback executes synchronously. Under most circumstances, the image acquisition object will be stopped and the `Running` property will be set to `'off'` by the time the MATLAB file completes execution.

Stop event information is stored in the `EventLog` property.

Characteristics

Access	Read/write
Data type	Character vector, function handle, or cell array
Values	The default value is an empty matrix (`[]`).

See Also

Properties

EventLog, Running

Tag

Specify descriptive text to associate with image acquisition object

Description

The `Tag` property specifies any descriptive text that you want to associate with an image acquisition object.

The `Tag` property can be useful when you are constructing programs that would otherwise need to define the image acquisition object as a global variable, or pass the object as an argument between callback routines.

You can use the value of the `Tag` property to search for particular image acquisition objects when using the `imaqfind` function.

Characteristics

Access	Read/Write
Data type	Character vector
Values	Any character vector

See Also

Functions

`imaqfind`

Timeout

Specify how long to wait for image data

Description

The `Timeout` property specifies the amount of time (in seconds) that the `getdata` and `getsnapshot` functions wait for data to be returned. The `Timeout` property is only associated with these blocking functions. If the specified time period expires, the functions return control to the MATLAB command line.

A timeout is one of the conditions for stopping an acquisition. When a timeout occurs, and the object is running, the MATLAB file function specified by `ErrorFcn` is called.

Note The `Timeout` property is not associated with hardware timeout conditions.

Characteristics

Access	Read only while running
Data type	`double`
Values	Any positive integer. The default value is 10 seconds.

See Also

Functions

`getdata`, `getsnapshot`

Properties

EventLog, ErrorFcn

TimerFcn

Specify MATLAB file callback function to execute when timer event occurs

Description

The `TimerFcn` property specifies the MATLAB file callback function to execute when a timer event occurs. A timer event occurs when the time period specified by the `TimerPeriod` property expires.

The toolbox measures time relative to when the object is started with the `start` function. Timer events stop being generated when the image acquisition object stops running.

Note Some timer events might not be processed if your system is significantly slowed or if the `TimerPeriod` value you specify is too small.

Characteristics

Access	Read/write
Data type	Character vector, function handle, or cell array
Values	The default value is an empty matrix (`[]`).

See Also

Functions

`start`, `stop`

Properties

TimerPeriod

TimerPeriod

Specify number of seconds between timer events

Description

The `TimerPeriod` property specifies the amount of time, in seconds, that must pass before a timer event is triggered.

The toolbox measures time relative to when the object is started with the `start` function. Timer events stop being generated when the image acquisition object stops running.

Note Some timer events might not be processed if your system is significantly slowed or if the `TimerPeriod` value you specify is too small.

Characteristics

Access	Read only while running
Data type	`double`
Values	Any positive value. The minimum value is 0.01 seconds. The default value is `1` (second).

See Also

Functions

`start`, `stop`

Properties

EventLog, TimerFcn

TriggerCondition

Indicate required condition before trigger event occurs

Description

The `TriggerCondition` property indicates the condition that must be met, via the `TriggerSource`, before a trigger event occurs. The trigger conditions that you can specify depend on the value of the `TriggerType` property.

TriggerType Value	Conditions Available
`'hardware'` (if available for your device)	Device-specific. For example, some Matrox hardware supports conditions such as `'risingEdge'` and `'fallingEdge'`. Use the `triggerinfo` function to view a list of valid values to use with your image acquisition hardware.
`'immediate'`	`'none'`
`'manual'`	`'none'`

You must use the `triggerconfig` function to set the value of this property.

Characteristics

Access	Read only. Use the `triggerconfig` function to set the value of this property.
Data type	Character vector
Values	Device specific. Use the `triggerinfo` function to view a list of valid values to use with your image acquisition hardware.

See Also

Functions

`trigger`, `triggerconfig`, `triggerinfo`

Properties

TriggerSource, TriggerType

TriggerFcn

Specify MATLAB file callback function to execute when trigger event occurs

Description

The `TriggerFcn` property specifies the MATLAB file callback function to execute when a trigger event occurs. The toolbox generates a trigger event when a trigger is executed based on the configured `TriggerType`, and data logging is initiated.

Under most circumstances, the MATLAB file callback function is not guaranteed to complete execution until sometime after the toolbox sets the `Logging` property to `'on'`.

Trigger event information is stored in the `EventLog` property.

Characteristics

Access	Read/write
Data type	Character vector, function handle, or cell array
Values	The default value is an empty matrix (`[]`).

See Also

Functions

`trigger`

Properties

EventLog, Logging

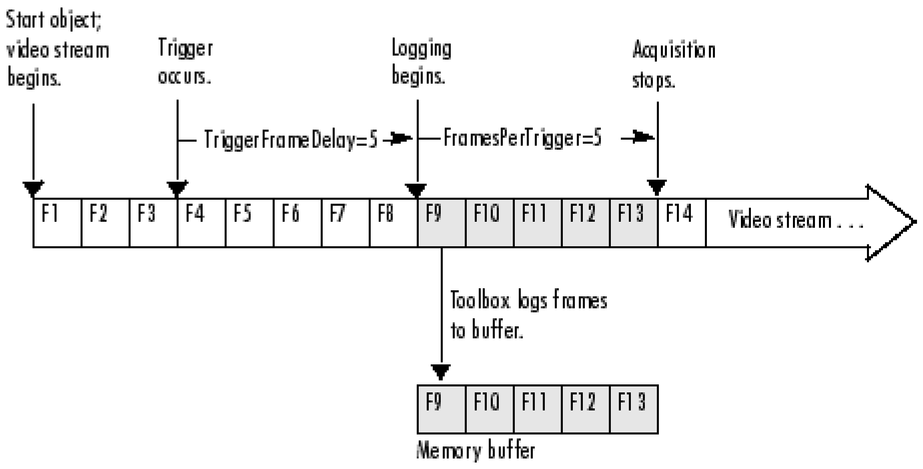
TriggerFrameDelay

Specify number of frames to skip before acquiring frames after trigger occurs

Description

The `TriggerFrameDelay` property specifies the number of frames to skip before acquiring frames after a trigger occurs. The object waits the specified number of frames after the trigger before starting to log frames.

In this figure, the `TriggerFrameDelay` is set to `5`, so the object lets five frames pass before starting to acquire frames. The number of frames captured is defined by the `FramesPerTrigger` property.



Characteristics

Access	Read only while running
Data type	`double`
Values	Any integer. The default value is 0 (zero).

See Also

Functions

`trigger`

Properties

FramesPerTrigger

TriggerRepeat

Specify number of additional times to execute trigger

Description

The `TriggerRepeat` property specifies the number of additional times you want the object to execute a trigger. This table describes the behavior for several typical `TriggerRepeat` values.

Value	Behavior
`0` (default)	Execute the trigger once when the trigger condition is met.
Any positive integer	Execute the trigger the specified number of additional times when the trigger condition is met.
`Inf`	Keep executing the trigger every time the trigger condition is met until the `stop` function is called or an error occurs.

To determine how many triggers have executed, check the value of the `TriggersExecuted` property.

Note If the `FramesPerTrigger` property is set to `Inf`, the object ignores the value of the `TriggerRepeat` property.

Characteristics

Access	Read only while running
Data type	`double`
Values	Any nonnegative integer. The default value is `0` (zero).

See Also

Functions

`stop`, `trigger`

Properties

FramesPerTrigger, TriggersExecuted, TriggerType

TriggersExecuted

Indicate total number of executed triggers

Description

The `TriggersExecuted` property indicates the total number of triggers that the video input object has executed.

Characteristics

Access	Read only
Data type	`double`
Values	Any nonnegative integer. The default value is `0` (zero).

See Also

Functions

`trigger`

Properties

EventLog

TriggerSource

Indicate hardware source to monitor for trigger conditions

Description

The `TriggerSource` property indicates the hardware source the image acquisition object monitors for trigger conditions. When the condition specified in the `TriggerCondition` property is met, the object executes the trigger and starts acquiring data.

You use the `triggerconfig` function to specify this value. The value of the `TriggerSource` property is device specific. You specify whatever mechanism a particular device uses to generate triggers.

For example, for Matrox hardware, the `TriggerSource` property could have values such as `'Port0'` or `'Port1'`. Use the `triggerinfo` function to view a list of values that are valid for your image acquisition device.

You must use the `triggerconfig` function to set the value of this property.

Note The `TriggerSource` property is only used when the `TriggerType` property is set to `'hardware'`.

Characteristics

Access	Read only. Use the `triggerconfig` function to set the value of this property.
Data type	Character vector
Values	Device-specific. Use the `triggerinfo` function to get a list of valid values.

See Also

Functions

`trigger`, `triggerconfig`, `triggerinfo`

Properties

TriggerCondition, TriggerType

TriggerType

Indicate type of trigger used by video input object

Description

The `TriggerType` property indicates the type of trigger used by the video input object. Triggers initiate data acquisition.

You use the `triggerconfig` function to specify one of the following values for this property.

Trigger Type	Description
`'hardware'` (if available for your device)	Trigger executes when a specified condition is met. You specify the condition using the `TriggerCondition` property and you specify the hardware source to monitor for the condition in the `TriggerSource` property. You use the `triggerconfig` function to set the values of these properties.
`'immediate'`	Trigger executes immediately after you call the `start` function.
`'manual'`	Trigger executes immediately after you call the `trigger` function.

Characteristics

Default value is enclosed in braces ({}).

Access	Read only. Use the `triggerconfig` function to set the value of this property.
Data type	Character vector
Values	`[ 'hardware' | {'immediate'} | 'manual' ]` The `'hardware'` option is only included for devices that support hardware triggers.

See Also

Functions

`trigger`, `triggerconfig`, `triggerinfo`

Properties

TriggerCondition, TriggerSource

Type

Identify type of image acquisition object

Description

The `Type` property identifies the type of image acquisition object. An image acquisition object can be either one of two types:

- Video input object
- Video source object

Characteristics

Access	Read only
Data type	Character vector
Values	`[ 'videoinput' | 'videosource' ]` Defined at object creation time

Examples

```
vid = videoinput('winvideo',1)

vid.Type

ans =

videoinput
```

This example gets the type of a video source object.

```
src = getselectedsource(vid);
src.Type
ans =
 videosource
```

See Also

Functions

`getselectedsource`, `videoinput`

UserData

Store data to associate with image acquisition object

Description

The `UserData` property specifies any data that you want to associate with an image acquisition object.

Note The object does not use the data in `UserData` directly. However, you can access the data by referencing the property as you would a field in a MATLAB structure using dot notation.

Characteristics

Access	Read/Write
Data type	Any
Values	User-defined

See Also

Functions

`get`

VideoFormat

Specify video format or name of device configuration file

Description

The `VideoFormat` property specifies the video format used by the image acquisition device or the name of a device configuration file, depending on which you specified when you created the object using the `videoinput` function.

Image acquisition devices typically support multiple video formats. When you create a video input object, you can specify the video format that you want the device to use. If you do not specify the video format as an argument, the `videoinput` function uses the default format. Use the `imaqhwinfo` function to determine which video formats a particular device supports and find out which format is the default.

As an alternative, you can specify the name of a device configuration file, also known as a camera file or digitizer configuration format (DCF) file. Some image acquisition devices use these files to store device configuration information. The `videoinput` function can use this file to determine the video format and other configuration information.

Use the `imaqhwinfo` function to determine if your device supports device configuration files.

Characteristics

Access	Read only
Data type	Character vector
Values	Device-specific. The example describes how to get a list of all the formats supported by a particular image acquisition device.

Examples

To determine the video formats supported by a device, check the `SupportedFormats` field in the device information structure returned by `imaqhwinfo`.

```
info = imaqhwinfo('winvideo')

info =

       AdaptorDllName: [1x73 char]
    AdaptorDllVersion: '2.1 (R2007a)'
          AdaptorName: 'winvideo'
            DeviceIDs: {[1]}
           DeviceInfo: [1x1 struct]

info.DeviceInfo

ans =

          DefaultFormat: 'RGB555_128x96'
    DeviceFileSupported: 0
```

```
             DeviceName: 'IBM PC Camera'
               DeviceID: 1
  VideoInputConstructor: 'videoinput('winvideo', 1)'
 VideoDeviceConstructor: 'imaq.VideoDevice('winvideo', 1)'
       SupportedFormats: {1x34 cell}
```

See Also

Functions

`imaqhwinfo`, `videoinput`

VideoResolution

Indicate width and height of incoming video stream

Description

The `VideoResolution` property is a two-element vector indicating the width and height in pixels of the frames in the incoming video stream. `VideoResolution` is specified as

```
[Width Height]
```

Note You specify the video resolution when you create the video input object, by passing in the video format argument to the `videoinput` function. If you do not specify a video format, the `videoinput` function uses the default video format. Use the `imaqhwinfo` function to determine which video formats a particular device supports and find out which format is the default.

Characteristics

Access	Read only
Data type	Vector of `doubles`
Values	Defined by video format

See Also

Functions

`imaqhwinfo`, `videoinput`

Properties

ROIPosition, VideoFormat

20

Blocks

From Video Device

Capture live image data from image acquisition device

Libraries:
Image Acquisition Toolbox

Description

The From Video Device block lets you capture image and video data streams from image acquisition devices, such as cameras and frame grabbers, in order to bring the image data into a Simulink model. The block also lets you configure and preview the acquisition directly from Simulink.

The From Video Device block opens, initializes, configures, and controls an acquisition device. The block opens, initializes, and configures only once, at the start of the model execution. While the Read All Frames option is selected, the block queues incoming image frames in a FIFO (first in, first out) buffer and delivers one image frame for each simulation time step. If the buffer underflows, the block waits for up to 10 seconds until a new frame is in the buffer.

The block has no input ports. You can configure the block to have either one output port or three output ports corresponding to the uncompressed color bands red, green, and blue or Y, Cb, and Cr. For more information about configuring the output ports, see the “Output” on page 20-2 section.

For an example of how to use this block, see “Save Video Data to a File” on page 9-6.

Other Supported Features

- The From Video Device block supports the use of Simulink Accelerator mode. This feature speeds up the execution of Simulink models.
- The From Video Device block supports the use of model referencing. This feature lets your model include other Simulink models as modular components.
- The From Video Device block supports the use of “Code Generation” on page 20-7 along with the `packNGo` function to group required source code and dependent shared libraries.

Ports

Output

Port_1 — Video output signal
m-by-*n*-by-3 matrix

Video output signal, specified as an *m*-by-*n*-by-3 matrix, where *m* represents the height of the video image and *n* represents the width of the video image.

Dependencies

- To enable this port, set **Ports mode** to `One multidimensional signal`.
- To specify the output video signal data type for this port, set **Data type**.

Data Types: `single` | `double` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`

R, G, B — RGB video output signal
m-by-*n* matrix

RGB video output signal, specified as an *m*-by-*n* matrix, where *m* represents the height of the video image and *n* represents the width of the video image. R, G, and B are separate output ports that each have the same dimensions.

Dependencies

- To enable this port, set **Output color space** to `rgb` and **Ports mode** to `Separate color signals`.
- To specify the output video signal data type for this port, set **Data type**.

Data Types: `single` | `double` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`

Y, Cb, Cr — YCbCr video output signal
m-by-*n* matrix

YCbCr video output signal, specified as an *m*-by-*n* matrix, where *m* represents the height of the video image and *n* represents the width of the video image. Y, Cb, and Cr are separate output ports that each have the same dimensions.

Dependencies

- To enable this port, set **Output color space** to `YCbCr` and **Ports mode** to `Separate color signals`.
- To specify the output video signal data type for this port, set **Data type**.

Data Types: `single` | `double` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`

Parameters

The following fields appear in the Block Parameters dialog box. If your selected device does not support a feature, it will not appear in the dialog box.

Device — Image acquisition device
available devices

The image acquisition device to which you want to connect. The items in the list vary, depending on which devices you have connected to your system. All video capture devices supported by Image Acquisition Toolbox software are supported by the block.

Video format — Video formats supported by device
available video formats

Shows the video formats supported by the selected device. This list varies with each device. If your device supports the use of camera files, `From camera file` is one of the choices in the list.

Dependencies

- To enable the **Camera file** parameter, set **Video format** to `From camera file`. This option only appears if your selected device supports camera raw image files. Enter the camera file path and file name, or use the **Browse** button to locate it.

Video source — Video input sources supported by device
available video input sources

Available input sources for the specified device and format. Click the **Edit properties...** button to open the Property Inspector and edit the source properties.

Edit properties... — Video source properties
button

Open the Property Inspector to edit video source device-specific properties, such as brightness and contrast. The properties that are listed vary by device. Properties that can be edited are indicated by a pencil icon or a drop-down list in the table. Properties that are grayed out cannot be edited. When you close the Property Inspector, your edits are saved.

Enable hardware triggering — Hardware-triggered acquisition
`off` (default) | `on`

This option only appears if the selected device supports hardware triggering. Select the check box to enable hardware triggering. After you enable triggering, you can select the **Trigger configuration**.

Dependencies

- To enable the **Trigger configuration** parameter, select the **Enable hardware triggering** parameter. This option only appears if the selected device supports hardware triggering. The configuration choices are listed by trigger source/trigger condition. For example, `TTL/fallingEdge` means that TTL is the trigger source and the falling edge of the signal is the condition that triggers the hardware.

ROI position — Region of interest in video image
[0, 0, maximum height, maximum width] (default) | [row, column, height, width]

Use this field to input a row vector that specifies the region of acquisition in the video image. The format is [row, column, height, width]. The default values for row and column are 0. The default values for height and width are set to the maximum allowable value, indicated by the resolution of the video format. Change the values in this field only if you do not want to capture the full image size.

Output color space — Video output color space
`rgb` (default) | `grayscale` | `YCbCr` | `bayer`

Use this field to select the color space for devices that support color. If your device supports Bayer sensor alignment, `bayer` is also available.

Dependencies

- To enable the **Bayer sensor alignment** parameter, set **Output color space** to `bayer`. This option is only available if your device supports Bayer sensor alignment. Use this to set the 2-by-2 pixel alignment of the Bayer sensor. Possible sensor alignment options are `grbg` (default), `gbrg`, `rggb`, and `bggr`.

Preview... — Preview of live video data
button

Preview the video image. Clicking this button opens the Video Preview window. While preview is running the image adjusts to changes you make in the parameter dialog box. Use the Video Preview window to set up your image acquisition in the way you want it to be acquired by the block when you run the model.

Block sample time — Block sampling rate
1/30 (default) | numeric

Specify the sample time of the block during the simulation. The sample time is the rate at which the block is executed during simulation.

Note The block sample time does not set the frame rate on the device that is used in simulation. The frame rate is determined by the video format specified (standard format or from a camera file). Some devices even list frame rate as a device-specific source property. Frame rate is not related to the **Block sample time** option in the dialog. The block sample time defines the rate at which the block executes during simulation time.

Ports mode — Type of video output signal
`One multidimensional signal` | `Separate color signals`

This option appears only if your device supports using either one output port or multiple output ports for the color bands. Use this option to specify either a single output port for all color spaces, or one port for each band (for example, R, G, and B). When you select `One multidimensional signal`, the output signal is combined into one line consisting of signal information for all color signals. Select `Separate color signals` if you want to use three ports corresponding to the uncompressed red, green, and blue color bands. Note that some devices use YCbCr for the separate color signals.

Note The block acquires data in the default `ReturnedColorSpace` setting for the specified device and format.

Data type — Video output data type
`single` (default) | `double` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`

The image data type when the block outputs frames. This data type indicates how image frames are returned from the block to Simulink. This option supports all MATLAB numeric data types.

Read All Frames — All available image frames captured
`off` (default) | `on`

Select to capture all available image frames. If you do not select this option, the block takes the latest snapshot of one frame, which is equivalent to using the `getsnapshot` function in the toolbox. If you select this option, the block queues incoming image frames in a FIFO (first in, first out) buffer. The block still gives you one frame, the oldest from the buffer, every timestep and ensures that no frames are lost. This option is equivalent to using the `getdata` function in the toolbox.

Kinect for Windows

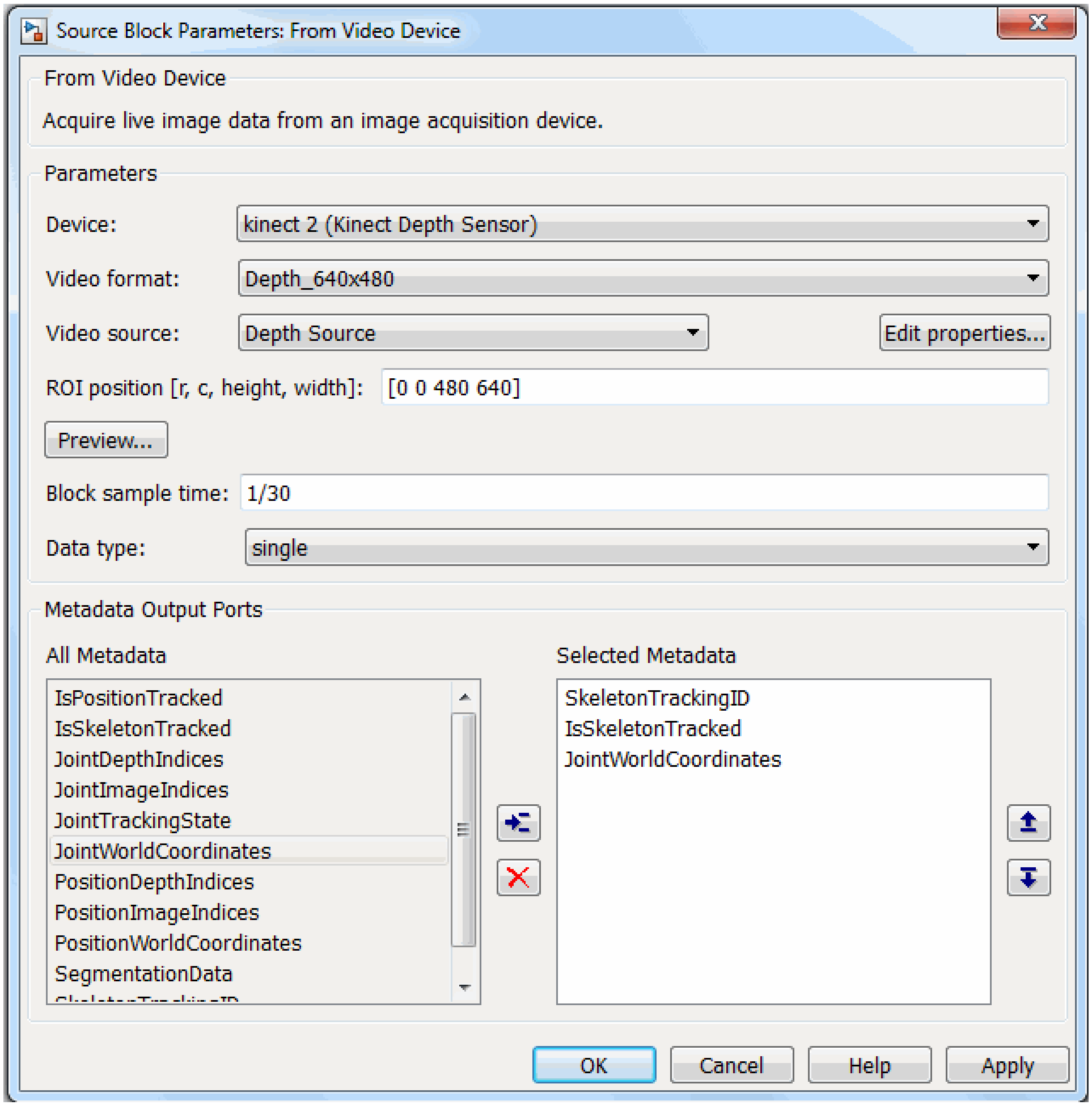
Metadata Output Ports — Kinect for Windows output ports

This option only appears if:

- You use a Kinect for Windows camera
- You select `Kinect Depth Sensor` as **Device**, and
- You select `Depth Source` as **Video source**.

Use this option to return skeleton information in Simulink during simulation and code generation. You can output metadata information in normal, accelerator, and deployed simulation modes. Each metadata item in the **Selected Metadata** list becomes an output port on the block.

The **All Metadata** section lists the metadata that is associated with the Kinect depth sensor.



Source Block Parameters: From Video Device

From Video Device

Acquire live image data from an image acquisition device.

Parameters

Device: kinect 2 (Kinect Depth Sensor)

Video format: Depth_640x480

Video source: Depth Source

ROI position [r, c, height, width]: [0 0 480 640]

Block sample time: 1/30

Data type: single

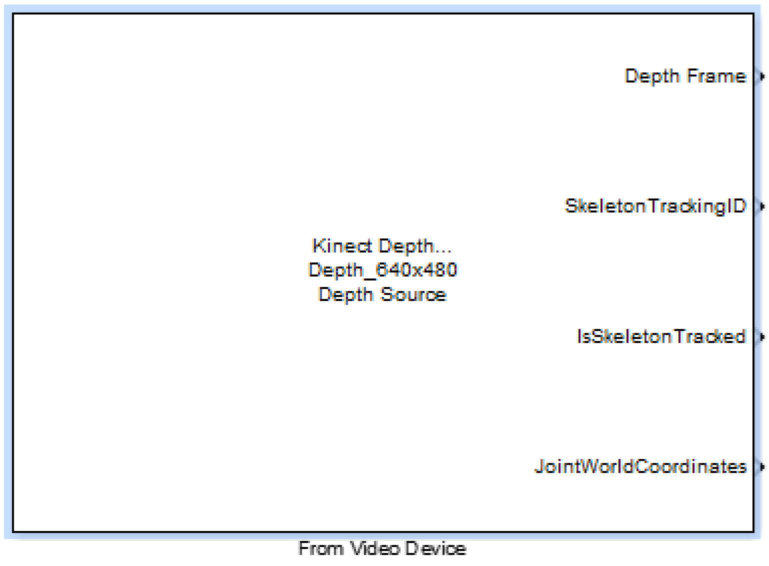
Metadata Output Ports

All Metadata: IsPositionTracked, IsSkeletonTracked, JointDepthIndices, JointImageIndices, JointTrackingState, JointWorldCoordinates, PositionDepthIndices, PositionImageIndices, PositionWorldCoordinates, SegmentationData

Selected Metadata: SkeletonTrackingID, IsSkeletonTracked, JointWorldCoordinates

This section is only visible when a Kinect depth sensor is selected. The **All Metadata** list shows the available metadata. The **Selected Metadata** list shows which metadata items are returned to Simulink. This is empty by default. To use a metadata item, add it from the **All Metadata** to the **Selected Metadata** list by selecting it in the **All Metadata** list and clicking the **Add** button (blue arrow icon). The **Remove** button (red X icon) removes an item from the **Selected Metadata** list. You can also use the **Move up** and **Move down** buttons to change the order of items in the **Selected Metadata** list. You can select multiple items at once.

You can see in the example above that three metadata items have been put in the **Selected Metadata** list. When you click **Apply**, output ports are created on the block for these metadata, as shown here. The first port is the depth frame.



For descriptions and information on these metadata fields and using Kinect for Windows with the Image Acquisition Toolbox, see “Acquiring Image and Skeletal Data Using Kinect” on page 12-7.

Version History

Introduced in R2007a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

See “Code Generation with From Video Device Block” on page 9-4.

See Also

Topics

“Save Video Data to a File” on page 9-6
“Acquiring Image and Skeletal Data Using Kinect” on page 12-7