Image Acquisition Toolbox Adaptor Kit

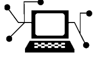
For Use with MATLAB®

- Computation
- Visualization
- Programming

User's Guide

Version 1

How to Contact The MathWorks

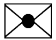


www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support

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

508-647-7000 (Phone)

508-647-7001 (Fax)



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

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

Image Acquisition Toolbox Adaptor Kit User's Guide

© COPYRIGHT 2005 by The MathWorks, Inc.

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

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

Trademarks

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks, and SimBiology, SimEvents, and SimHydraulics are trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Patents

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

Revision History

September 2005	PDF only	New for Version 1.0 (Release 14SP3)

Contents

1 Getting Started

Overview **1-2**
What Knowledge Is Required? **1-3**

Creating an Adaptor **1-4**
Staged Development Model **1-4**

Looking at the Demo Adaptor **1-7**
Finding the Demo Adaptor Source Files **1-7**
Viewing the Demo Adaptor Source Files **1-8**
Setting Breakpoints **1-10**
Building the Demo Adaptor **1-11**
Registering an Adaptor with the Toolbox **1-11**
Running the Demo Adaptor **1-11**

2 Setting Up Your Build Environment

Overview **2-2**
Required Header Files **2-2**
Required Libraries **2-3**

Creating an Adaptor Project **2-4**
Adding the Adaptor Kit Project to Your Solution **2-7**

Specifying Header File Locations **2-8**
Specifying Header Files in Microsoft Visual C++ .Net **2-8**
Using Environment Variables **2-9**

Specifying Libraries and Library Paths **2-11**
Specifying Libraries in Microsoft Visual C++ .Net **2-11**

Configuring Other Project Parameters **2-14**

Exporting Adaptor Functions **2-18**
Setting Up a Module Definition File **2-18**

3 Providing Hardware Information

Adaptor Exported Functions: An Overview **3-3**

Creating a Stub Adaptor **3-5**

Performing Adaptor and Device SDK Initialization ... **3-8**
Example **3-8**

Specifying Device and Format Information **3-9**
Using Objects to Store Device and Format Information ... **3-9**
Suggested Algorithm **3-11**
Storing Device Information **3-12**
Storing Format Information **3-13**
Example: Providing Device and Format Information **3-15**

Defining Classes to Hold Device-Specific Information **3-18**
Defining a Device or Format Information Class **3-18**
Storing Adaptor Data **3-18**

Unloading Your Adaptor DLL **3-20**
Example **3-20**

Returning Warnings and Errors to MATLAB **3-21**

4 Defining Your Adaptor Class

Overview 4-2

Summary of IAdaptor Abstract Class Virtual Functions 4-3

Creating a Stub Implementation of Your Adaptor Class 4-5

Identifying Video Sources 4-10
Suggested Algorithm 4-10

Instantiating an Adaptor Object 4-12
Suggested Algorithm 4-12
Implementing Your Adaptor Class Constructor 4-13
Implementing Your Adaptor Class Destructor 4-14

5 Acquiring Image Data

Overview 5-2
User Scenario 5-2
Triggering 5-2
Overview of Virtual Functions Used to Acquire Data 5-3

Specifying the Format of the Image Data 5-5
Specifying Image Dimensions 5-5
Specifying Frame Type 5-7

Opening and Closing a Connection with a Device 5-10
Suggested Algorithm for openDevice() 5-10
Suggested Algorithm for closeDevice() 5-13

Starting and Stopping Image Acquisition 5-15

Suggested Algorithm for startCapture() **5-15**
Suggested Algorithm for stopCapture() **5-17**

Implementing the Acquisition Thread Function **5-19**
User Scenario **5-19**
Suggested Algorithm **5-19**
Example **5-22**

Supporting ROIs **5-25**
Implementing Software ROI **5-25**
Implementing Hardware ROI **5-27**

Supporting Hardware Triggers **5-28**
Example **5-29**

Using Critical Sections **5-31**
Understanding Critical Sections **5-31**
Example: Using a Critical Section **5-32**

Specifying Device Driver Identification Information .. **5-35**
User Scenario **5-35**
Example **5-35**

6 Defining Device-Specific Properties

Overview **6-2**
User Scenario **6-2**
Suggested Algorithm **6-3**

Creating Device Properties **6-6**
Selecting the Property Creation Function **6-6**
Reading Properties from an IMDF File **6-7**
Creating Property Help **6-8**
Example **6-8**

Defining Hardware Trigger Configurations **6-10**

Setting Up Property Listeners **6-11**
User Scenario **6-11**
Receiving Notification of Property Value Changes **6-11**
Defining a Listener Class **6-12**
Creating the notify() Function **6-13**
Associating a Listener with a Property Container **6-15**

7 Storing Adaptor Information in an IMDF File

Overview **7-3**
User Scenario **7-3**
Elements of the IMDF Markup Language **7-3**

Creating an IMDF File: Toplevel Elements **7-5**

Specifying Help in an IMDF File **7-7**
User Scenario: Viewing Property Help **7-8**
Creating AdaptorHelp Nodes **7-10**

Specifying Device Information **7-13**
Example: Device Node **7-14**

Specifying Property Information **7-16**
Specifying Property Element Attributes **7-17**

Specifying Format Information **7-20**

Specifying Hardware Trigger Information **7-22**
Specifying Trigger Sources **7-23**
Specifying Trigger Conditions **7-23**

Specifying Video Sources **7-24**

Defining and Including Sections **7-25**

Index

Getting Started

This section introduces the Image Acquisition Toolbox Adaptor Kit.

Overview (p. 1-2)	Describes what an adaptor is and why you would build one
Creating an Adaptor (p. 1-4)	Describes the design decisions required to create an adaptor and specifies a recommended procedure for creating an adaptor
Looking at the Demo Adaptor (p. 1-7)	Provides a quick introduction to adaptor development by examining the demo adaptor that is included with the adaptor kit

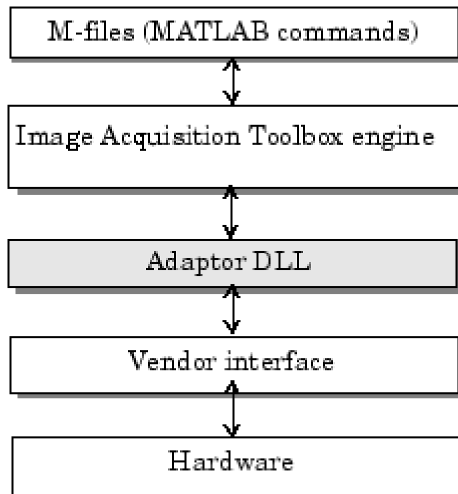
Overview

The Image Acquisition Toolbox Adaptor Kit is a C++ framework that you can use to create an adaptor. A C++ framework is a set of classes that work together to create a particular application. In a framework, the design of the software is already defined. With the adaptor framework, you subclass the framework classes and implement the required member functions to flesh out the design to support your particular hardware.

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 software development kit (SDK).

You develop an adaptor to support new hardware. Adaptors enable the dynamic loading of support for hardware without requiring recompilation and linking of the toolbox. Using an adaptor to add hardware support gives you the advantage of having multiple prepackaged features such as data logging, triggering, and a standardized interface to the image acquisition device.

This diagram shows the relationship of an adaptor to the toolbox engine and a device driver.



Relationship of Adaptor to Toolbox Components

What Knowledge Is Required?

To build an adaptor, you should have a working knowledge of

- C++
- The functionality of your hardware device, and its associated application programming interface (API)
- Image Acquisition Toolbox concepts, functionality, and terminology as described in the Image Acquisition Toolbox User's Guide documentation

Creating an Adaptor

To create an adaptor, you must implement the C++ routines and the classes required by the adaptor framework. The following outlines one way to develop an adaptor that divides the task into several smaller tasks, called stages. This staged development model has been shown to be an effective way to create an adaptor.

- "Stage 1: Familiarize Yourself with the Adaptor Kit and Device SDK" on page 1-4
- "Stage 2: Set Up Your Build Environment" on page 1-5
- "Stage 3: Provide Hardware Information" on page 1-5
- "Stage 4: Creating a Video Input Object" on page 1-5
- "Stage 5: Acquiring Video Data" on page 1-6
- "Stage 6: Creating Device Properties" on page 1-6

Staged Development Model

Stage 1: Familiarize Yourself with the Adaptor Kit and Device SDK

Before you start developing an adaptor, you must gather information about the device (or devices) to help you make design decisions.

- Familiarize yourself with adaptors and adaptor development by looking at the demo adaptor which is included with the adaptor kit — see "Looking at the Demo Adaptor" on page 1-7.
- Familiarize yourself with your device's SDK. Devices provide the tools you need to access and control them programmatically. You must learn your device's requirements for initialization, startup, and acquiring data, and the SDK functions used to perform these tasks.
- Determine what device or devices you want to support with your adaptor. You can create an adaptor to support one particular device, a group of devices offered by a particular vendor, or a group of devices that all support a common interface. You must also determine the formats supported by the

device and the properties of the device that you want to make available to users of your adaptor.

Stage 2: Set Up Your Build Environment

You must set up the required adaptor build environment, which includes specifying the names and locations of required header files and libraries. Chapter 2, "Setting Up Your Build Environment" provides this information, showing how to set up an adaptor project in Microsoft Visual C++ .Net.

Note Using Microsoft Visual C++ .Net is not a requirement to create an adaptor. You can use any ANSI compatible C++ compiler. However, the adaptor kit was created using the Microsoft Visual C++ .Net environment and includes Microsoft Visual C++ .Net project files.

Stage 3: Provide Hardware Information

Your adaptor must provide the toolbox with information about the device (or devices) it makes available to users. In this stage, you define the labels you want to use to identify the devices available through your adaptor and the formats they support. The toolbox displays these labels to users who must specify the device and format they want to use for an acquisition.

In this stage, you start adaptor development by creating a stub implementation of your adaptor. After building your adaptor DLL and registering it with the toolbox, you can use the `imaqhwinfo` function and verify that the toolbox can find your adaptor and load it. For more information about this stage, see Chapter 3, "Providing Hardware Information"

Stage 4: Creating a Video Input Object

You must define your adaptor class. Every adaptor must define an adaptor class that is a subclass of the adaptor kit `IAdaptor` class.

In this stage, you add a stub implementation of your adaptor class to your adaptor project. This enables you to call the `videoinput` function and instantiate a video input object with your adaptor. For more information, see Chapter 4, "Defining Your Adaptor Class"

Stage 5: Acquiring Video Data

In this stage, you flesh out the stub implementations of the virtual functions in your adaptor class. After completing this stage, you will be able to acquire data from your device and bring it into the MATLAB workspace.

In addition, in this step you can also implement support for defining a region-of-interest (ROI) in the acquiring data and for using hardware triggering, if your device supports this capability. For more information, see Chapter 5, “Acquiring Image Data”

Stage 6: Creating Device Properties

In this stage, you decide which properties of the device you want to expose to toolbox users. You make this determination by reading the device’s SDK documentation, determining its capabilities, and deciding which capabilities toolbox users will expect to configure. Once you decide to expose a property, you must decide on a name for the property, determine its data type, and, optionally, the range of valid values. As an alternative, you can define device-specific properties in an image device definition file (IMDF). For more information, see Chapter 6, “Defining Device-Specific Properties”

Looking at the Demo Adaptor

A good way to get a quick introduction to adaptors and adaptor development is by looking at the demo adaptor that is included with the Image Acquisition Toolbox Adaptor Kit. The demo adaptor is a functioning adaptor that does not require any hardware. You can build the demo adaptor and run it to get familiar with how an adaptor works.

- "Finding the Demo Adaptor Source Files" on page 1-7
- "Viewing the Demo Adaptor Source Files" on page 1-8
- "Setting Breakpoints" on page 1-10
- "Building the Demo Adaptor" on page 1-11
- "Registering an Adaptor with the Toolbox" on page 1-11
- "Running the Demo Adaptor" on page 1-11

Finding the Demo Adaptor Source Files

The demo adaptor C++ source files reside in the following directory:

```
$MATLAB\toolbox\imaq\imaqadaptor\kit\demo\
```

To look at the source files in Microsoft Visual C++ .Net, open the demo adaptor solution file, `mwdemoimaq.sln`.

The following table lists all the files in the demo directory in alphabetical order, with brief descriptions.

Source File	Description
`DemoAdaptor.cpp`	Demo adaptor class implementation
`DemoAdaptor.h`	Demo adaptor class definition
`DemoDeviceFormat.cpp`	Implementation of class that holds device format information
`DemoDeviceFormat.h`	Definition of class that holds device format information

Source File	Description
DemoPropListener.cpp	Implementation of class that notifies demo adaptor when the value of a device property changes
DemoPropListener.h	Definition of class that notifies demo adaptor when the value of a device property changes
DemoSourceListener.cpp	Implementation of class that listens for changes in the selected video source
DemoSourceListener.h	Definition of class used to listen for changes in the selected video source
mwdemoimaq.cpp	Implementation of the five functions that every adaptor must export.
mwdemoimaq.def	Module definition file that exports the five functions that every adaptor must export
mwdemoimaq.dll	Demo adaptor library. This is the compiled and linked Dynamic Link Library (DLL) that implements the demo adaptor.
mwdemoimaq.h	Header file that defines the five functions that every adaptor must export
mwdemoimaq.ilk	Linker database file created automatically by Microsoft Visual C++
mwdemoimaq.imdf	Demo adaptor image device file (IMDF) that contains property definitions
mwdemoimaq.sln	Microsoft Visual C++ solution file for the demo adaptor
mwdemoimaq.vcproj	Microsoft Visual C++ project file for the demo adaptor

Viewing the Demo Adaptor Source Files

This section describes a suggested order in which you should look at the demo adaptor source files.

mwdemoimaq.h

A good place to start looking at the demo adaptor is to open the `mwdemoimaq.h` file. This file defines the five functions that every adaptor must export. The toolbox engine calls these functions to get information about supported hardware, instantiate a video input object, and acquire data. Implementing these functions is typically the first step an adaptor writer takes. This header file contains comments that explain the purpose of each function. The `mwdemoimaq.def` file is the module definition file in which these functions are exported.

mwdemoimaq.cpp

After seeing the definition of the adaptor exported functions, see how they are implemented in the corresponding C++ implementation file, `mwdemoimaq.cpp`.

DemoAdaptor.h

After viewing the exported functions, take a look at the definition of the `DemoAdaptor` class in `DemoAdaptor.h`. The adaptor class is a subclass of the `IAdaptor` class, which defines the virtual functions an adaptor must implement. This header file contains comments that explain the purpose of each member function.

DemoAdaptor.cpp

After seeing the definition of the adaptor class, look at the implementation of the class in the `DemoAdaptor.cpp` file. This file contains the acquisition thread function which is the main frame acquisition loop. This is where the adaptor connects to the device and acquires image frames.

Other Demo Adaptor Files

The demo directory contains other files that implement optional adaptor kit capabilities.

For example, the `DemoDeviceFormat.h` and corresponding `.cpp` files illustrate one way to store device-specific format information using adaptor data. You define a class that is a subclass of the `IMAQInterface` class to hold the information. See "Defining Classes to Hold Device-Specific Information" on page 3-18 for more information.

The DemoPropListener.h and corresponding .cpp files and the DemoSourceListener.h and .cpp files illustrate how your adaptor can get notified if a user changes the setting of a property. See "Setting Up Property Listeners" on page 6-11 for more information.

Setting Breakpoints

You can use debugger breakpoints to examine which adaptor functions are called when users call toolbox functions, such as imaqhwinfo, videoinput, start, and stop. The following table lists places in the demo adaptor where you can set a breakpoints.

MATLAB Command	Breakpoint
imaqhwinfo	initializeAdaptor()
imaqreset	uninitializeAdaptor()
imaqhwinfo(adaptorname)	getAvailHW()
videoinput	getDeviceAttributes() createInstance()
imaqhwinfo(obj)	getDriverDescription() getDriverVersion() getMaxWidth() getMaxHeight() getFrameType()
videoinput	getNumberOfBands()
start	openDevice()
start or trigger, if manual trigger	startCapture()
stop	stopCapture() closeDevice()

Building the Demo Adaptor

After familiarizing yourself with the demo adaptor source files, build the demo adaptor DLL. By default, the demo adaptor project stores the DLL in the demo adaptor directory. This directory already includes a demo adaptor DLL but you can overwrite it.

Note To build the demo adaptor, you must have an environment variable named `MATLAB` defined on your system. Set the value of this environment variable to the location of your MATLAB installation directory. For information about setting an environment variable on a Windows system, see "Using Environment Variables" on page 2-9.

Registering an Adaptor with the Toolbox

After creating an adaptor, you must inform the Image Acquisition Toolbox of its existence by registering it with the `imaqregister` function. This function tells the toolbox where to find third-party adaptor libraries. You only need to register your adaptor once. The toolbox stores adaptor location information in your MATLAB preferences.

Note Because the toolbox caches adaptor information, you might need to reset the toolbox, using `imaqreset`, before a newly registered adaptor appears in the `imaqhwinfo` listing.

For example, the following code registers the demo adaptor with the toolbox using the `imaqregister` function, where `<your_directory>` represents the name of the directory where you created the demo adaptor.

```
imaqregister('<your_directory>\mwdemoimaq.dll');
```

Running the Demo Adaptor

Start MATLAB and call the `imaqhwinfo` function. You should be able to see the demo adaptor included in the adaptors listed in the `InstalledAdaptors` field.

```
imaqhwinfo
```

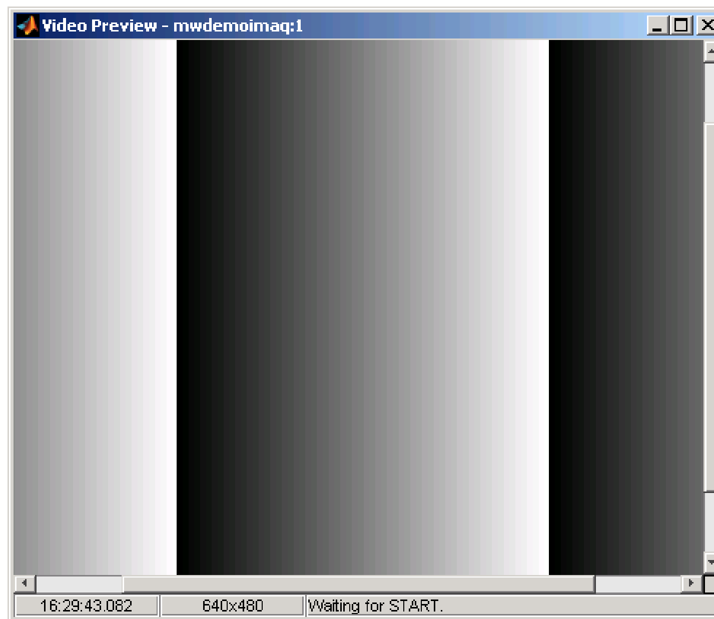
```
ans =

    InstalledAdaptors: {'dcam'  'mwdemoimaq' 'winvideo'}
         MATLABVersion: '7.1 (R14SP3)'
           ToolboxName: 'Image Acquisition Toolbox'
        ToolboxVersion: '1.9 (R14SP3)'
```

Create a video input object with the demo adaptor.

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

Get a preview of the data being returned by the demo adaptor using the `preview` function. Note that the demo adaptor generates a grayscale pattern to mimic the data returned by a real image acquisition device. The demo adaptor does not connect to an actual device.

```
preview(vid);
```



Preview Window Containing Demo Adaptor Data

2

Setting Up Your Build Environment

This chapter describes the libraries and include files you need to build an adaptor. The chapter also provides details about setting up the build environment in Microsoft Visual C++ .Net.

Overview (p. 2-2)	Provides a summary of the header files and libraries required to create an adaptor
Creating an Adaptor Project (p. 2-4)	Describes how to create a project in Microsoft Visual C++ .Net
Specifying Header File Locations (p. 2-8)	Describes how to specify the locations of header files required by the adaptor kit and your device's SDK
Specifying Libraries and Library Paths (p. 2-11)	Describes how to specify the names and locations of the link libraries required by the adaptor kit and your device's SDK
Configuring Other Project Parameters (p. 2-14)	Describes some other Microsoft Visual C++ .Net parameter configurations required to create an adaptor
Exporting Adaptor Functions (p. 2-18)	Describes how to create a module definition file and configure it in the Microsoft Visual C++ .Net environment

Overview

Setting up the build environment involves specifying the header files and libraries that you need to create an adaptor. For those familiar with their IDE environment, see the following sections for lists of these required include files and libraries.

- "Required Header Files" on page 2-2
- "Required Libraries" on page 2-3

For detailed instructions on setting up your build environment in the Microsoft Visual C++ .Net environment, start at "Creating an Adaptor Project" on page 2-4.

Note Users of Microsoft Visual C++ .Net should be aware that there are certain project parameters that they must set. See "Configuring Other Project Parameters" on page 2-14.

Required Header Files

The following table lists the locations of the header files that are required to build an adaptor. In the table, the path dereferences the environment variable `MATLAB` which contains the name of your MATLAB installation directory. (For more information, see "Using Environment Variables" on page 2-9.) To learn how to specify this header file location information in Microsoft Visual C++ .Net, see "Specifying Header Files in Microsoft Visual C++ .Net" on page 2-8.

Note You must also specify the location of the header files required by your device. Read your device's SDK documentation to get this information.

Header Files	Location
Adaptor kit header files	`$(MATLAB)\toolbox\imaq\imaqadaptors\kit\include`
MATLAB external interfaces header files	`$(MATLAB)\extern\include`

Required Libraries

The following table lists the libraries required by the adaptor kit. In the table, the path dereferences the environment variable `MATLAB`, which is set to the name of your MATLAB installation directory. (See "Using Environment Variables" on page 2-9 for more information.) To learn how to specify the locations of these libraries in Microsoft Visual C++ .Net, see "Specifying Libraries in Microsoft Visual C++ .Net" on page 2-11.

Note You must also specify the libraries required by your device. Read your device's SDK documentation to determine which libraries are required and get their locations.

Library	Description	Location
`libmex.lib`	MATLAB MEX interface library	`$(MATLAB)\extern\lib\win32\microsoft\msvc71\`
`libmx.lib`	MATLAB array interface library	`$(MATLAB)\extern\lib\win32\microsoft\msvc71\`

`imaqmex.lib`	Image Acquisition Toolbox engine library	Debug version:
		`$(MATLAB)\toolbox\imaq\imaqadaptors\kit\lib\win32\debug\`
		Release version:
		`$(MATLAB)\toolbox\imaq\imaqadaptors\kit\lib\win32\release\`

Creating an Adaptor Project

As the first step toward building an adaptor, open Microsoft Visual C++ .Net and create a new project.

Note Using Microsoft Visual C++ .Net is not a requirement to create an adaptor. You can use any ANSI compatible C++ compiler. However, the adaptor kit was created using the Visual C++ .Net environment and includes a Visual C++ .Net solution file.

---

1. Start Microsoft Visual C++ .Net.

2. On the Visual C++ .Net **Start Page**, click **New Project**. Visual C++ opens the New Project dialog box, shown in the following figure. You can also open this dialog box by from the **File** menu: **File->New->Project**.

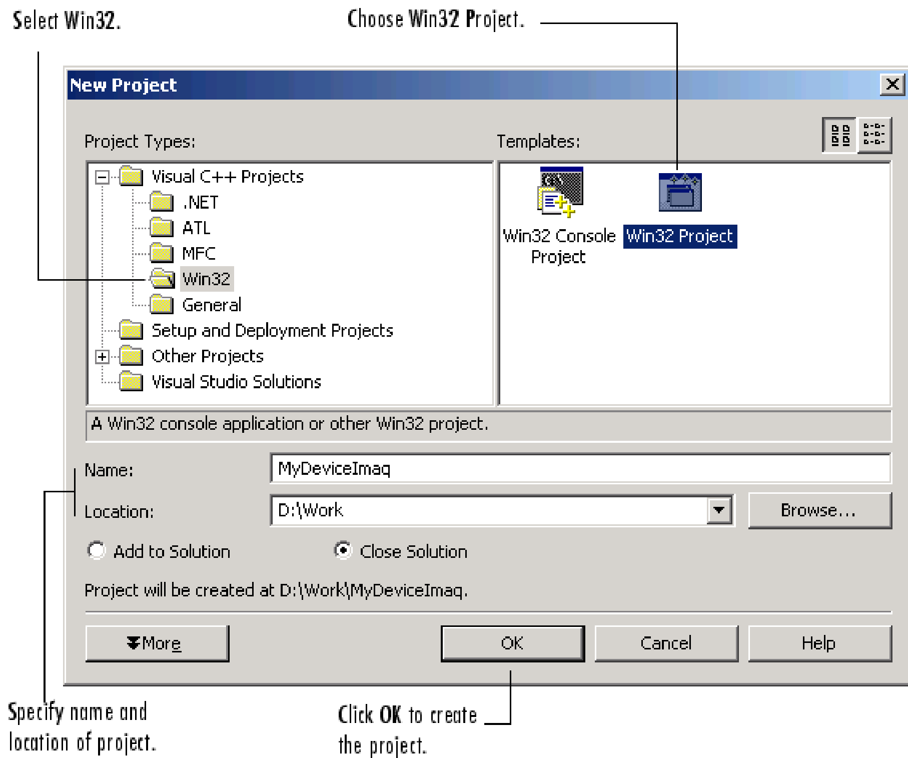
3. In the New Project dialog box, under Project Types, expand the Visual C++ Projects folder, open Win32 folder, and click **Win32 project**.

4. Enter the name you want to assign to the project and specify where you want to locate the project in the **Name** and **Location** fields.

   You can give your adaptor project any name. A convention used by the toolbox is to name adaptors as follows:

   ```
   vendor_name + imaq
   ```

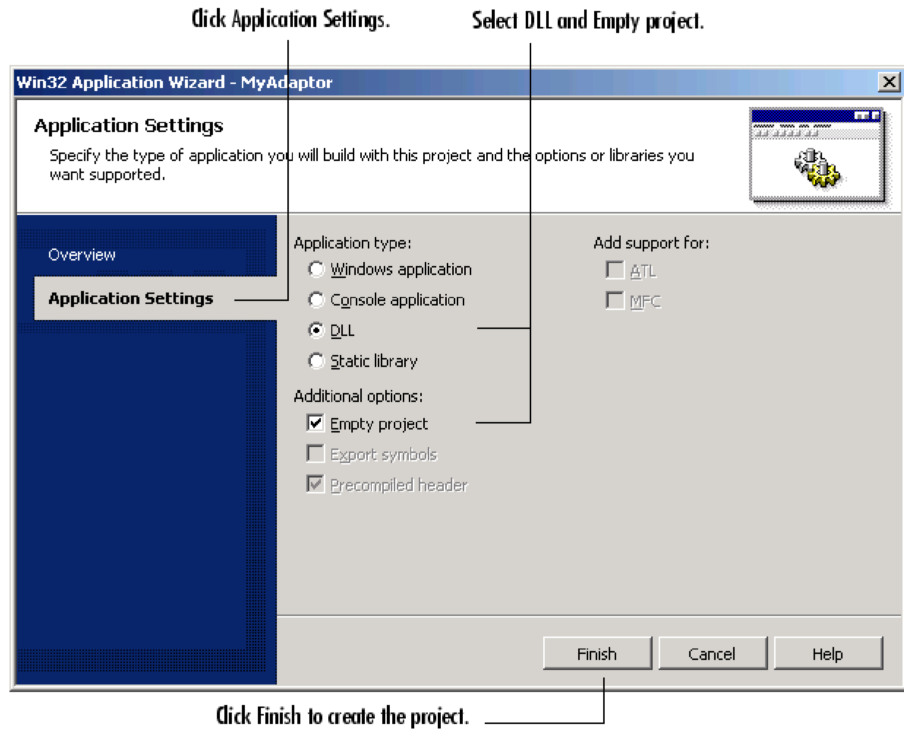
   where you replace the string `vendor_name` with something appropriate to your project. For example, this documentation specifies the project name `mydeviceimaq`. Microsoft Visual C++ .Net uses the project name as the default adaptor DLL name. To specify a different name for the adaptor DLL, open the project property pages (**Project->Properties**), open the Linker folder, click **General** and edit the Output File field.



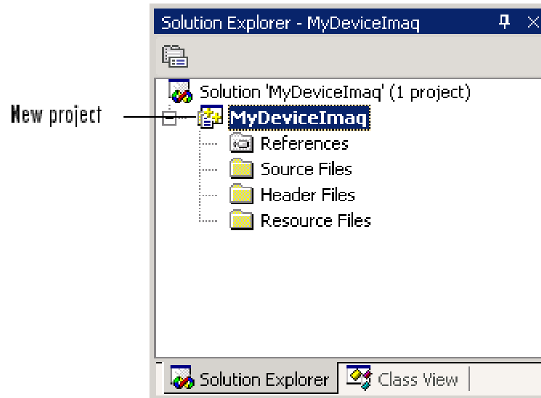
When you click **OK**, Visual C++ opens the Win32 Application Wizard.

5 In the Win32 Application Wizard, click **Application Settings**.

6 On the Application Settings page, select **DLL** from the list of application types and select **Empty project** from the Application options section. Click **Finish** to create the project.



After you create the project, Visual C++ displays the project in its **Solution Explorer**, with separate folders for source files, header files, and other project resources, as shown in the following figure.



Adding the Adaptor Kit Project to Your Solution

When you create a project, Microsoft Visual C++ .Net automatically creates a solution that contains your project. As a convenience, while you are developing your adaptor, you might want to add the adaptor kit project to your solution to make it easier to refer to adaptor kit files. Adding the adaptor kit project to your solution does not affect the compilation or linkage of your adaptor DLL.

To add the adaptor kit project to your solution, go to the **File** menu, select the **Add Project** option, and then select **Add an Existing Project**. In the **Add Existing Project** dialog box, open the following project file,

```
$MATLAB\toolbox\imaq\imaqadaptors\kit\imaqadaptorkit.vcproj
```

where `$MATLAB` represents your MATLAB installation directory.

Specifying Header File Locations

Before you can compile your adaptor, you must specify the locations of the header files required by the adaptor kit and by your device's SDK. For a list of the header files required by the adaptor kit, see "Overview" on page 2-2. The following section describes how to specify these header file locations in the Microsoft Visual C++ .Net environment.

Note The examples in the following section use environment variables. For more information, see "Using Environment Variables" on page 2-9.

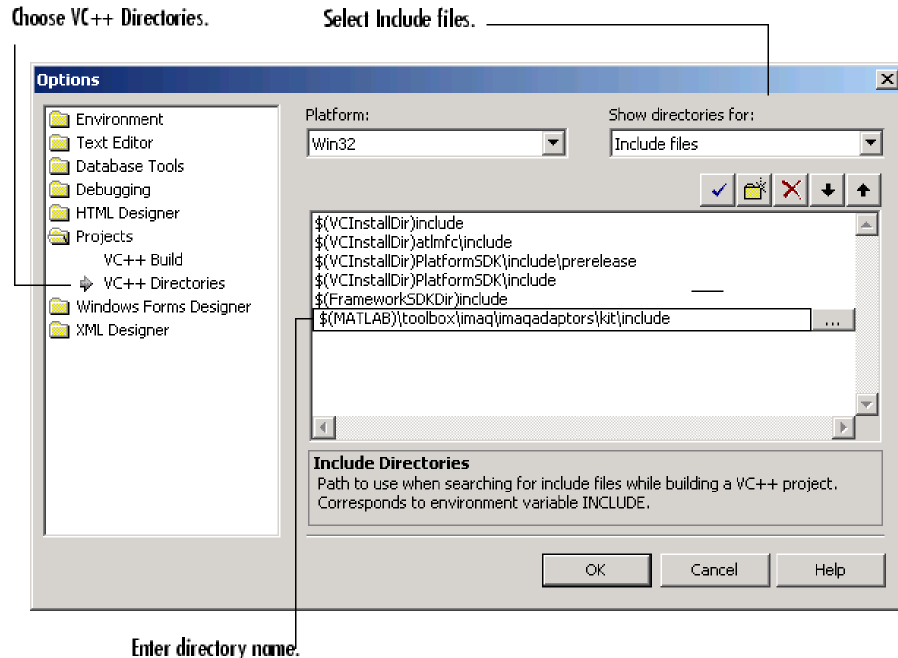
Specifying Header Files in Microsoft Visual C++ .Net

To specify the locations of the adaptor kit header files in Microsoft Visual C++ .Net, follow these instructions:

1. From the **Tools** menu, select **Options**.
2. In the Options dialog box, open the Projects folder and select **VC++ Directories**.
3. Select **Include files** from the **Show directories for** field.
4. Add the locations of adaptor kit header files and the header files required by your device's SDK to the list of directories displayed, each on a separate line.

   ```
   $(MATLAB)\toolbox\imaq\imaqadaptors\kit\include
   $(MATLAB)\extern\include
   ```

   In this example, `$(MATLAB)` dereferences the environment variable `MATLAB`, which is set to the name of your installation directory. (See "Using Environment Variables" on page 2-9 for more information.)



5 After specifying the header file directories, click **OK**.

Using Environment Variables

To set an environment variable for your MATLAB installation directory in Windows, follow this procedure:

1 Open the System Properties dialog box. One way to do this is to right-click the My Computer icon on your desktop and select **Properties**.

2 Click the **Advanced** tab.

3 On the **Advanced** tab, click the **Environment Variables** button.

4 In the Environment Variables dialog box, in the **User variables** section, click **New** to create an environment variable.

5 In the New User Variable dialog box, assign the name `MATLAB` to the variable and set the value of the variable to your MATLAB installation directory path. Click **OK**.

6 Click **OK** in the Environment Variables dialog box.

7 Click **OK** in the System Properties dialog box.

Note If Microsoft Visual C++ .Net is running when you create this variable, you must restart it.

Specifying Libraries and Library Paths

Before you can create your adaptor DLL, you must specify the libraries required by the adaptor kit and by your device's SDK. For a list of required libraries, see "Overview" on page 2-2. The following section describes how to specify these libraries in the Microsoft Visual C++ .Net environment.

Note The examples in the following section use environment variables. For more information, see "Using Environment Variables" on page 2-9.

Specifying Libraries in Microsoft Visual C++ .Net

Specifying libraries in Microsoft Visual C++ .Net is a two-step process. First you specify the locations of the libraries and then, in a separate field, you specify the names of the libraries. The following shows this procedure.

Specifying Library Locations

1. Open the Property Pages for your project by right-clicking on your project in the Solution Explorer and choosing **Properties**, or by selecting **Properties** from the Project menu.

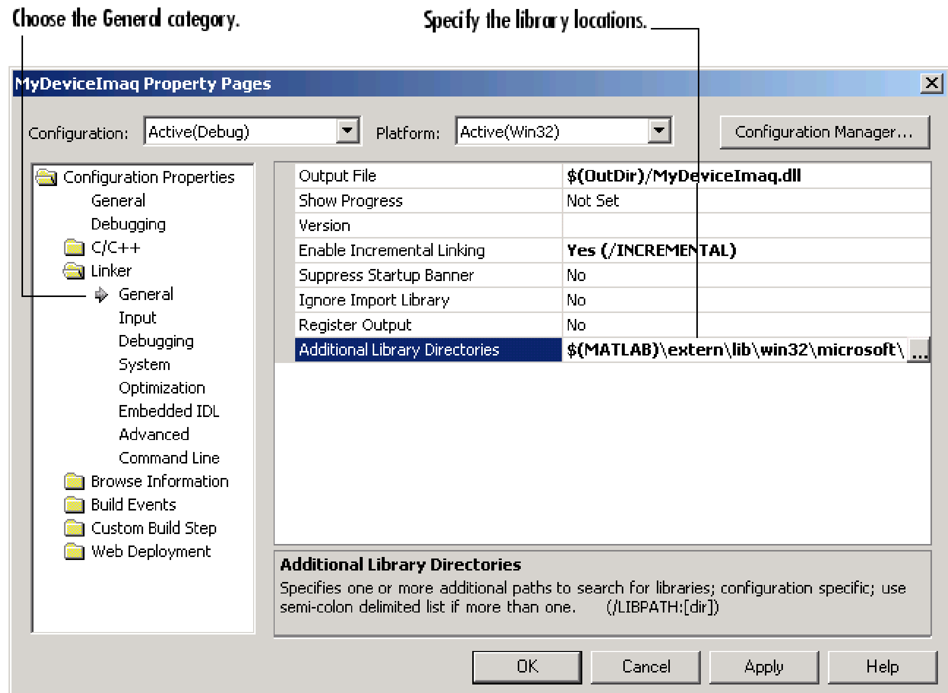
2. Open the **Linker** folder and select the **General** category.

3. Add the locations of adaptor kit libraries and the libraries required by your device's SDK in the **Additional Library Directories** field. Use a semicolon to separate the directories.

Note The following example specifies the debug version of the `imaqmex` library. If you want to use the release version of this library, change the last directory name in the path to `release`.

```
$(MATLAB)\extern\lib\win32\microsoft\msvc71\;$(MATLAB)\toolbox\imaq\
imaqadaptors\kit\lib\win32\debug
```

In this example, `$(MATLAB)` dereferences the environment variable `MATLAB`, which is set to the name of your installation directory. (See "Using Environment Variables" on page 2-9 for more information.)



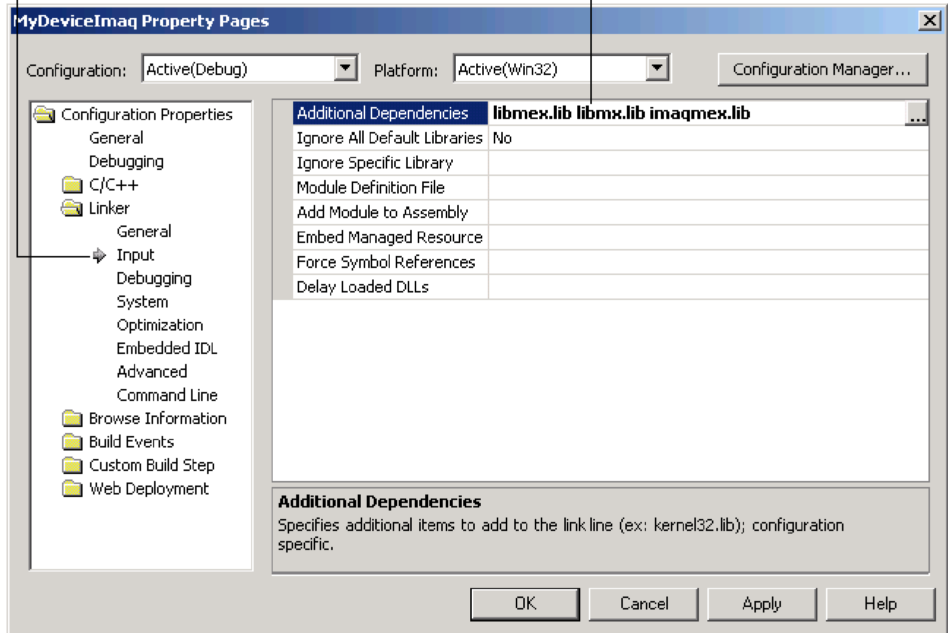
Specifying Library Names

After specifying the library directories, you must specify the library names.

1 In the Property Pages dialog box for your project, open the Linker folder and select the **Input** category.

2 In the **Additional Dependencies** field, specify the names of the adaptor kit libraries and the names of the libraries required by your device's SDK. Use spaces to separate the names of the libraries. The following shows the adaptor kit libraries.

```
libmex.lib libmx.lib imaqmex.lib
```



3 Click **OK**.

Configuring Other Project Parameters

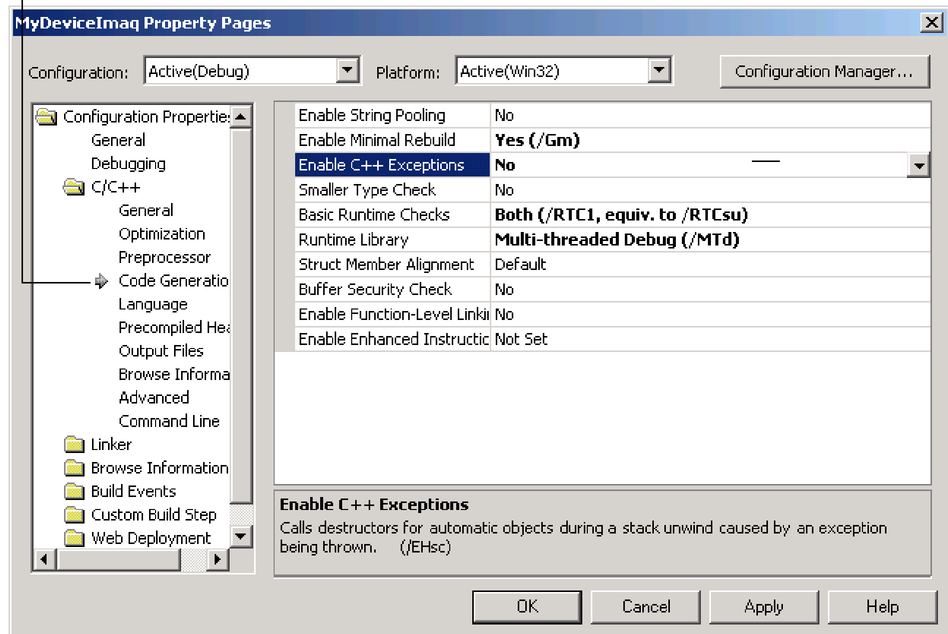
In addition to specifying the header files and libraries, an adaptor project requires these additional settings. These settings must be done for both release and debug builds; otherwise, run-time problems might occur.

Note To set the values of these properties, your project must contain files. You can add an empty source file to your project now. Select **New** on the **File** menu, save the file, and then add it to your project. Alternatively, you can wait to perform these configurations until you start writing adaptor code, described in .

---

1. Open the Property Pages for your project by right-clicking on your project in the Solution Explorer and choosing **Properties**, or by selecting **Properties** from the Project menu.

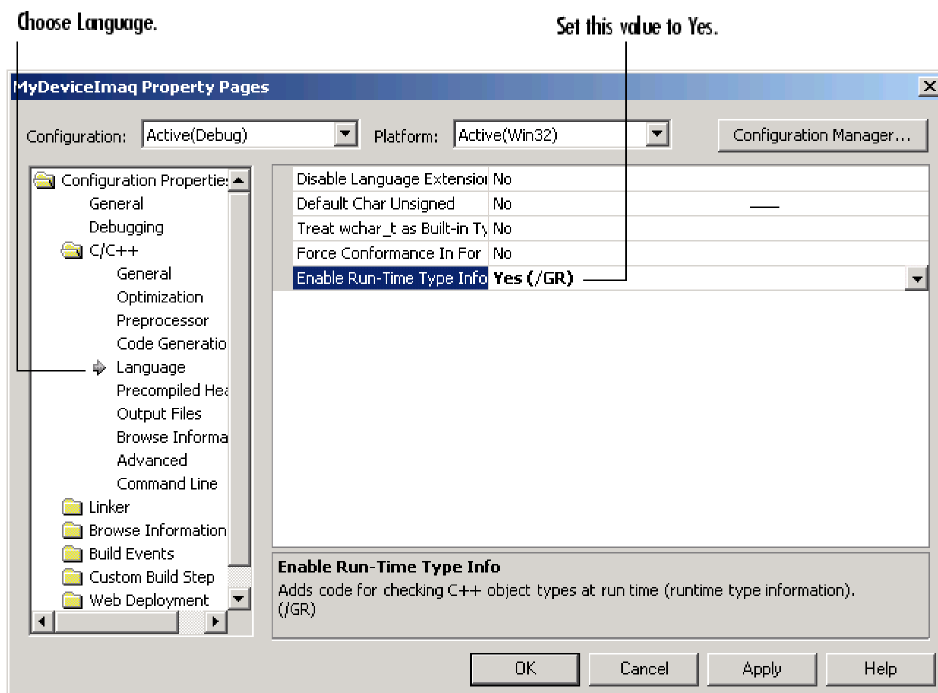
2. Open the C/C++ folder and select **Code Generation**.



3 On the Code Generation page, set the following values.

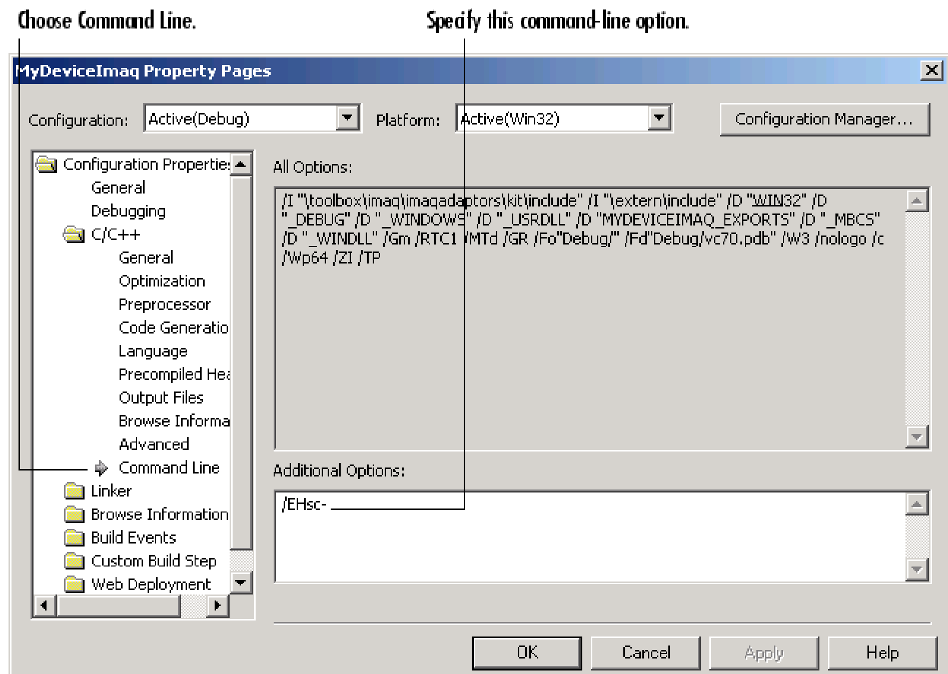
Field	Setting
Enable C++ Exceptions	No
Run-time library	Multi-threaded DLL (for release builds) or Multi-threaded Debug DLL (for debug builds)

4 In the **C/C++** folder, select **Language** and set the **Enable Run-time Type Info** field to **Yes**.



5 In the **C/C++** folder select **Command Line** and specify the following compiler flag.

```
/EHsc-
```



6 Click **OK**.

Exporting Adaptor Functions

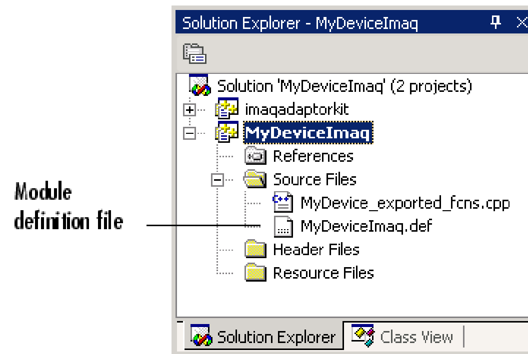
Your adaptor must export certain functions that are expected by the Image Acquisition Toolbox engine. You use a module definition file (`.def`) to export functions in a DLL. (For general information about module definition files, go to the MSDN Web site.)

The following section describes how to set up a module definition file in your adaptor project. To learn more about what an adaptor's module definition file must contain, see "Creating a Stub Adaptor" on page 3-5.

Setting Up a Module Definition File

To set up a module definition file in Microsoft Visual C++ .Net, follow these instructions.

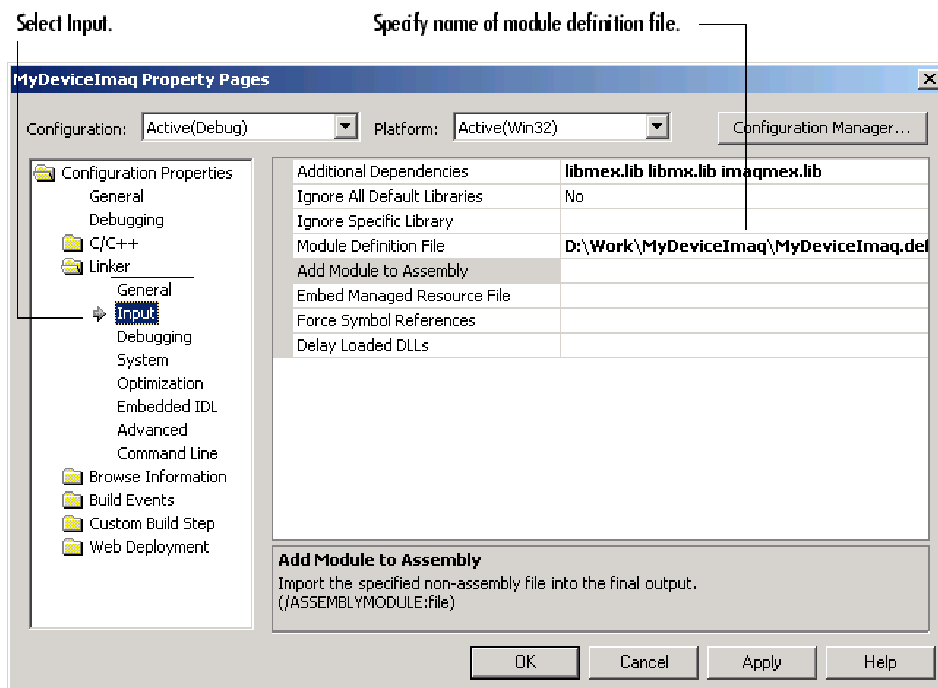
1. From the **File** menu in Microsoft Visual C++ .Net, select **New** and then choose **File**. Visual C++ opens the New File dialog box.
2. Under Categories, select **General**. From the file types listed in this category, select **Text File** and click **Open**, or double-click the icon representing the file type.
3. From the **File** menu, select the **Save as** option and specify the name you want to assign to the file, using the file extension `.def`. Adaptor writers typically use the project name as the name of the module definition file.
4. Add the module definition file to your adaptor project, using the **Move File into Project** option.



5 After you create a module definition file and add it to your project, you must tell Visual C++ where to find it. Select your project in the Solutions Explorer, open the **Project** menu and choose **Properties.**

6 In the **Properties Pages** dialog box, open the **Linker** folder and select the **Input** category.

7 Select Edit in the **Module Definition File** field and specify the name of the module definition file, as in the following example.



8 Click **OK**.

3

Providing Hardware Information

This chapter describes how an adaptor provides the toolbox engine with information about the image acquisition device (or devices) available on a user's system. After completing the tasks outlined in this chapter, you will be able to create your adaptor DLL, register it with the toolbox, and see it included in the list of available adaptors returned by `imaqhwinfo`.

Adaptor Exported Functions: An Overview (p. 3-3)	Provides an overview of the five required functions that every adaptor must export, including a flow-of-control diagram
Creating a Stub Adaptor (p. 3-5)	Describes how to create a stub implementation of your adaptor
Performing Adaptor and Device SDK Initialization (p. 3-8)	Describes how to perform any initialization required by your adaptor or by your device's SDK
Specifying Device and Format Information (p. 3-9)	Describes how to write the function that provides the toolbox with information about the devices currently available on the user's system
Defining Classes to Hold Device-Specific Information (p. 3-18)	Describes an optional method for storing additional device-specific information

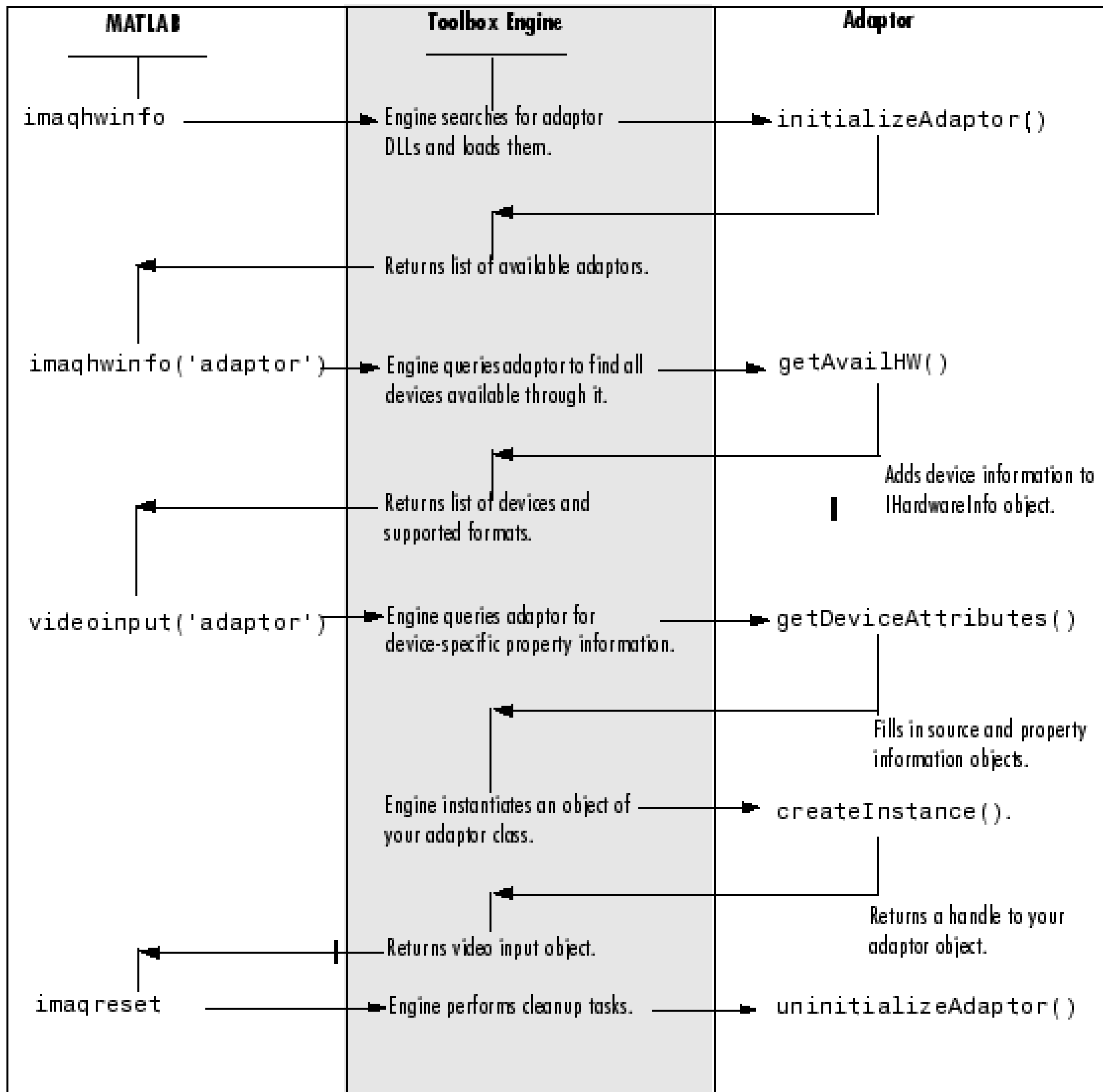
Unloading Your Adaptor DLL (p. 3-20)	Describes how to write the function that can perform any cleanup required when your adaptor DLL is unloaded
Returning Warnings and Errors to MATLAB (p. 3-21)	Describes how to return errors and warnings that are consistent with the MATLAB error format

Adaptor Exported Functions: An Overview

The Image Acquisition Toolbox engine requires that every adaptor export five functions. The toolbox calls these functions to communicate with the device and acquire data. One of your primary tasks as an adaptor writer is to provide implementations of these functions. The following table lists these five exported functions with pointers to sections that provide more detail about how to implement the function.

Export Function	Purpose
`initializeAdaptor()`	Performs any initialization required by your adaptor or your device's SDK. See "Performing Adaptor and Device SDK Initialization" on page 3-8.
`getAvailHW()`	Provides the toolbox engine with information about the device (or devices) available through your adaptor
`getDeviceAttributes()`	Specifies the video source, device-specific properties, and hardware trigger information, if supported. See Chapter 6, "Defining Device-Specific Properties".
`createInstance()`	Instantiates an object of a C++ class that represents the communication between the toolbox and the device. **Note**: Because you cannot create a stub of this function until you define an adaptor class, this function is described in Chapter 4, "Defining Your Adaptor Class".
`uninitializeAdaptor()`	Performs any cleanup required by your adaptor and unloads the adaptor DLL. See "Unloading Your Adaptor DLL" on page 3-20.

The following figure shows the flow of control between the MATLAB command line, the toolbox engine, and the exported adaptor functions. Note that the figure does not show how the adaptor communicates with the device's SDK to get information. This varies with each device's SDK.



Flow of Control Among MATLAB, Toolbox Engine, and Adaptor

Creating a Stub Adaptor

The easiest way to start building an adaptor is to create a stub implementation, compile and link it, and then test your work. This method can be effective because it provides immediate results and lets you verify that your build environment is setup properly.

This section shows a stub implementations of an adaptor that you can copy and paste into a file in your adaptor Microsoft Visual C++ .Net project. After compiling and linking this code, you can see your adaptor included in the list of available adaptors returned by the `imaqhwinfo` function.

Note You will not be able to instantiate an object of your adaptor class, however. That is described in Chapter 4, "Defining Your Adaptor Class"

To create a stub adaptor, follow this procedure:

1 Add a C++ source file to the adaptor C++ project. See Chapter 2, "Setting Up Your Build Environment" for information about creating an adaptor C++ project. This source file will hold your implementations of your adaptor's exported C++ functions. You can give this file any name. This example uses the name of the adaptor for this file, with the text string `"_exported_fcns"` appended to it, `mydevice_exported_fcns.cpp`

2 Copy the following lines of C++ code into this new file. This code provides stub implementations of several adaptor exported functions. Note that you must include the adaptor kit header file `mwadaptorimaq.h`. This header file includes all other required adaptor kit header files.

```
#include "mwadaptorimaq.h"

void initializeAdaptor(){

}
void getAvailHW(imaqkit::IHardwareInfo* hardwareInfo){

}
```

```
void uninitializeAdaptor(){

}
```

3 Add a module definition file to the adaptor C++ project to export the functions. A module definition file is a text file that contains statements that affect linker behavior. To learn how to create a module definition file, see "Exporting Adaptor Functions" on page 2-18.

   You can give your module definition file any name. This example gives it the same name as the adaptor project, with the `.def` file extension, `mydeviceimaq.def`.

   To export the functions created in the previous step, copy the following lines of code into your module definition file.

```
LIBRARY mydeviceimaq
EXPORTS
    initializeAdaptor   @1
    getAvailHW          @2
    uninitializeAdaptor @3
```

4 Build the adaptor DLL. Select the **Build Solution** option on the **Build** menu.

5 Start MATLAB.

6 Tell the toolbox where to find this new adaptor using the `imaqregister` function. See "Registering an Adaptor with the Toolbox" on page 1-11 for more information. You only need to perform this step once.

7 Call the `imaqhwinfo` function. Note that the stub adaptor, named `mydeviceimaq`, is included in the list of available adaptors returned.

```
imaqhwinfo

ans =

    InstalledAdaptors: {'mydeviceimaq'  'winvideo'}
        MATLABVersion: '7.1 (R14SP3)'
          ToolboxName: 'Image Acquisition Toolbox'
```

```
        ToolboxVersion: '1.9 (R14SP3)'
```

To get more information about the stub adaptor, call `imaqhwinfo` again, this time specifying the name of the adaptor. Note that the `DeviceIDs` field and the `DeviceInfo` fields are empty.

```
imaqhwinfo('mydeviceimaq')

ans =

       AdaptorDllName: 'D:\Work\mydeviceimaq\mydeviceimaq.dll'
    AdaptorDllVersion: '1.8 (R14SP2)'
          AdaptorName: 'mydeviceimaq'
            DeviceIDs: {1x0 cell}
           DeviceInfo: [1x0 struct]
```

Performing Adaptor and Device SDK Initialization

Every adaptor must include an `initializeAdaptor()` function. In this function, you should perform any initialization required by your adaptor or your device's SDK. Check the documentation that came with your device to find out what, if any, initialization the SDK requires.

For example, some device SDKs provide a function that loads required DLLs into memory. Not every device's SDK requires initialization; however, every adaptor must include the `initializeAdaptor()` function, even if it is an empty implementation.

Note You do not perform device initialization in this function. For information about performing device initialization, see "Opening and Closing a Connection with a Device" on page 5-10.

Example

As defined by the adaptor kit, the `initializeAdaptor()` function accepts no arguments and does not return a value. The following example implements an empty `initializeAdaptor()` function.

```
void initializeAdaptor()
{
     // Perform initialization required by adaptor or device SDK.

}
```

Specifying Device and Format Information

Every adaptor must include a `getAvailHW()` function. In this function, you provide the toolbox with information about the device (or devices) that are currently connected to the user's system. An adaptor can represent one particular device, multiple devices supported by a particular vendor, or a class of devices. For example, the toolbox includes an adaptor for Matrox devices that supports many different framegrabbers provided by that vendor.

When a user calls the `imaqhwinfo` function to find out which devices are available on their system, the toolbox engine calls your adaptor's `getAvailHW()` function to get this information. When you implement this function, you determine the names, device IDs, and format names that the toolbox displays to users.

This section includes the following topics

- "Using Objects to Store Device and Format Information" on page 3-9
- "Suggested Algorithm" on page 3-11
- "Storing Device Information" on page 3-12
- "Storing Format Information" on page 3-13
- "Storing Adaptor Data" on page 3-18

Using Objects to Store Device and Format Information

The adaptor kit provides three classes to store device and format information. The following table lists these classes with a brief description.

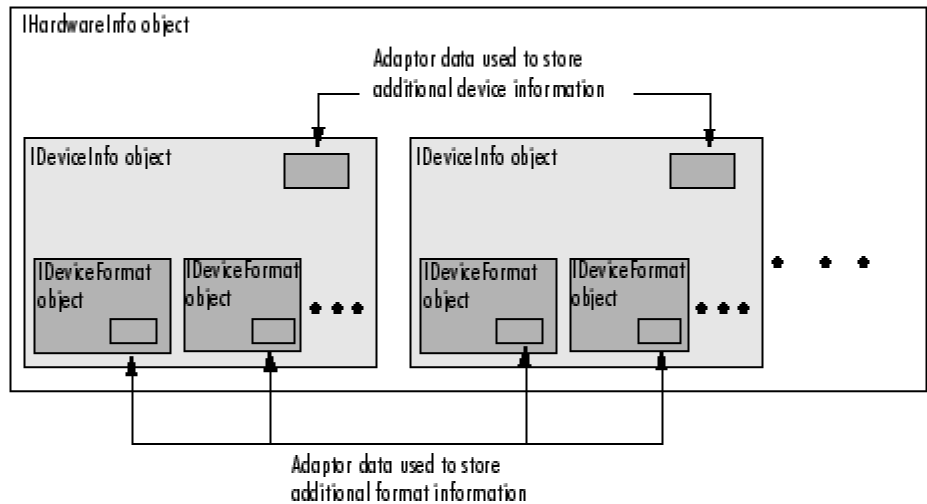
Adaptor Kit Object	Purpose
`IHardwareInfo`	Overall container class for hardware information
`IDeviceInfo`	Container for information about a particular device
`IDeviceFormat`	Container for information about the formats supported by a particular device

When the toolbox engine calls your adaptor's `getAvailHW()` function, it passes your adaptor a handle to an `IHardwareInfo` object.

For each device you want to make available through your adaptor, you must create an `IDeviceInfo` object and then store the object in the `IHardwareInfo` object. For each format supported by a device, you must create an `IDeviceFormat` object and then store the object in the `IDeviceInfo` object.

The following figure shows the relationship of these adaptor kit objects. The figure shows the `IHardwareInfo` object containing two `IDeviceInfo` objects, but it can contain more. Similarly, each `IDeviceInfo` object is shown containing two `IDeviceFormat` objects, but it can also contain more.

Note in the figure that both the `IDeviceInfo` and `IDeviceFormat` objects contain adaptor data. Adaptor data is an optional way to store additional information about a device or format in an `IDeviceInfo` or `IDeviceFormat` object. See "Defining Classes to Hold Device-Specific Information" on page 3-18 for more information.



Adaptor Kit Objects Used to Store Device and Format Information

Suggested Algorithm

The `getAvailHW()` function accepts one argument: the handle to an `IHardwareInfo` object. The toolbox engine creates this `IHardwareInfo` object and passes the handle to your adaptor when it calls your adaptor's `getAvailHW()` function. The `getAvailHW()` function does not return a value.

```
void getAvailHW(imaqkit::IHardwareInfo* hardwareInfo)
```

Your adaptor's `getAvailHW()` function must provide the engine with the following information for each device:

- Device ID
- Device name
- Formats supported by the device, including the default format
- Whether or not the device supports device configuration files (also known as camera files)

Note You can optionally store additional device-specific information in the adaptor data of an `IDeviceInfo` object or an `IDeviceFormat` object. See "Defining Classes to Hold Device-Specific Information" on page 3-18 for more information.

The following outlines the steps typically performed by a `getAvailHW()` function. The figure that follows presents this algorithm in flowchart form.

1. Determine which devices are available through the adaptor. Adaptors typically make calls to the device's SDK to get this information.

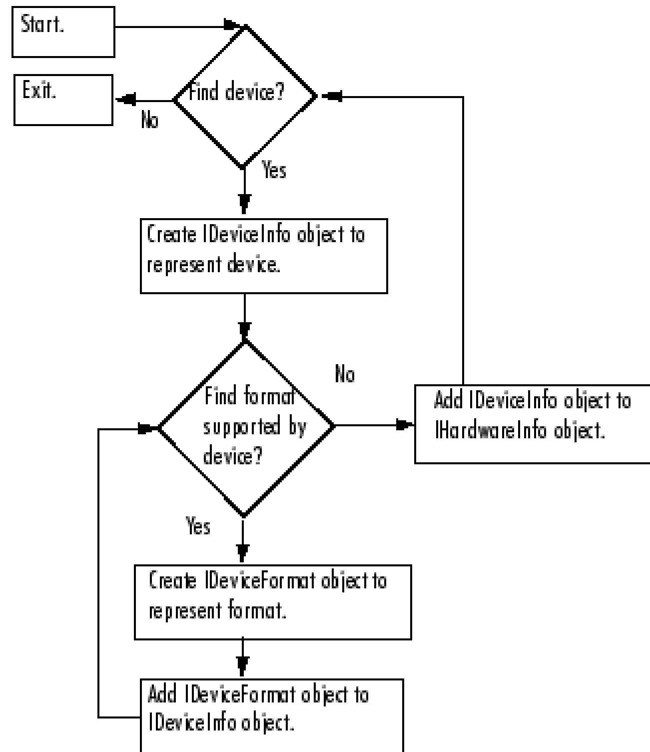
2. For each device found, create an `IDeviceInfo` object — see "Storing Device Information" on page 3-12.

   a. For each format supported by the device, create an `IDeviceFormat` object — see "Storing Format Information" on page 3-13.

   b. Add each device format object that you create to the `IDeviceInfo` object.

3. Add the `IDeviceInfo` object to the `IHardwareInfo` object passed to your `getAvailHW()` function by the toolbox engine.

4 Repeat this procedure for each device available on the user's system.



Suggested Algorithm for getAvailHW() Function

Storing Device Information

You store device information in an `IDeviceInfo` object. To create this object, use the `createDeviceInfo()` member function of the `IHardwareInfo` object, as in the following example:

```
imaqkit::IDeviceInfo* deviceInfo =
          hardwareInfo->createDeviceInfo(1,"MyDevice");
```

As arguments to `createDeviceInfo()`, you specify:

- Name you want to assign to the device

- ID you want to assign to the device

You can specify any values for these arguments, but note that they are visible to toolbox users in the structure returned by `imaqhwinfo`.

For device name, specify a text string that easily identifies the device. For example, you might use the manufacturer's model number.

The ID you specify for the device must be unique because it identifies the device for the adaptor. Because MATLAB indexing starts at 1, start the numbering of device IDs at 1, not zero. The device with ID 1 is the default device for your adaptor.

The `IDeviceInfo` object you create supports member functions to perform many tasks, such as creating, adding, and retrieving the `IDeviceFormat` objects associated with the device, and indicating whether the device supports device configuration files (also known as camera files). For more information about this class, see the Image Acquisition Toolbox Adaptor Kit API Reference documentation.

Adding the IDeviceInfo Object to the IHardwareInfo Object

After you create the `IDeviceInfo` object, you must add it to the `IHardwareInfo` object that the engine passed to your `getAvailHW()` function. Use the `addDevice()` member function of the `IHardwareInfo` object, as in the following example:

```
hardwareInfo->addDevice(deviceInfo);
```

Storing Format Information

You store format information in an `IDeviceFormat` object. To create this object, use the `createDeviceFormat()` member function of an `IDeviceInfo` object, as in the following example:

```
imaqkit::IDeviceFormat* deviceFormat =
          deviceInfo->createDeviceFormat(1,"RS170");
```

As arguments to `createDeviceFormat()`, you specify

- Name you want to assign to the format

- ID you want to assign to the format

For the format name, specify a text string that describes the format. Note that the format name is visible to toolbox users. Use names that might be familiar to users of the device, such as a name similar to the format names used by the device manufacturer.

Because the ID is not exposed to users, you can specify any convenient value. For example, if the device's SDK uses numerical identifiers to indicate a format, use these values for your format IDs.

You can use `IDeviceFormat` member functions to perform many tasks, such as, retrieving the format name and format ID, and determining whether the format is the default format. For more information about this class, see the Image Acquisition Toolbox Adaptor Kit API Reference documentation.

Adding an IDeviceFormat Object to an IDeviceInfo Object

After you create the `IDeviceFormat` object, add it to the `IDeviceInfo` object that represents the device. Use the `addDeviceFormat()` member function of the `IDeviceInfo` object, as in the following example:

```
deviceInfo->addDeviceFormat(deviceFormat,true);
```

Specifying the Default Format

When you add a format to an `IDeviceInfo` object, you use the second argument to the `addDeviceFormat()` function to specify whether the format should be used as the default format for the device. The `imaqhwinfo` function returns the name of the default format in the `DefaultFormat` field. To make a format the default, set this argument to `true`.

Configuring Device Configuration File (Camera File) Support

Some devices use device configuration files (also known as camera files) to configure formats and other properties. If a device supports device configuration files, you do not need to create `IDeviceFormat` objects. Instead, use the `setDeviceFileSupport()` member function of the `IDeviceInfo` object to indicate that the device supports device configuration files, as in the following example:

```
deviceInfo->setDeviceFileSupport(true);
```

For these devices, users pass the full path of the device configuration file as the third argument to the `videoinput` function, instead of specifying a device format string. Adaptor writers do not need to perform any processing of the device configuration file; you just pass the file name to the device.

Example: Providing Device and Format Information

The following example presents a simple implementation of a `getAvailHW()` function that specifies information for one device with two formats. The intent of this example is to show how you create the objects necessary to store device and format information. If you add this code to the `mydeviceimaq` adaptor, you can run `imaqhwinfo('mydeviceimaq')` to view the device information.

1 Replace the stub implementation of the `getAvailHW()` function, created in "Creating a Stub Adaptor" on page 3-5, with this code:

```
void getAvailHW(imaqkit::IHardwareInfo* hardwareInfo)
{
 // Create a Device Info object.
 imaqkit::IDeviceInfo* deviceInfo =
                 hardwareInfo->createDeviceInfo(1,"MyDevice");

   // Create a Device Format object.
 imaqkit::IDeviceFormat* deviceFormat =
                 deviceInfo->createDeviceFormat(1,"RS170");

   // Add the format object to the Device object.
   // Specifying "true' makes this format the default format.
 deviceInfo->addDeviceFormat(deviceFormat, true);

   // Create a second Device Format object.
 imaqkit::IDeviceFormat* deviceFormat2 =
                  deviceInfo->createDeviceFormat(2,"PAL");

   // Add the second format object to the Device object.
   deviceInfo->addDeviceFormat(deviceFormat2, false);

    // Add the device object to the hardware info object.
```

```
    hardwareInfo->addDevice(deviceInfo);

   }
```

2 Rebuild the `mydeviceimaq` project to create a new DLL.

3 Start MATLAB and run the `imaqhwinfo` function, specifying the adaptor name `mydeviceimaq` as an argument. Note how the `DeviceIDs` field and the `DeviceInfo` field of the structure returned by `imaqhwinfo` now contain data.

```
dev = imaqhwinfo('mydeviceimaq')

dev =

       AdaptorDllName: 'D:\Work\mydeviceimaq\mydeviceimaq2.dll'
    AdaptorDllVersion: '1.9 (R14SP3)'
          AdaptorName: 'mydeviceimaq'
            DeviceIDs: {[1]}
           DeviceInfo: [1x1 struct]
```

To view detailed information about the device, view the structure in the `DeviceInfo` field. The `DeviceInfo` field is an array of structures, where each structure provides detailed information about a particular device.

```
dev_info = dev.DeviceInfo

dev_info =

          DefaultFormat: 'RS170'
    DeviceFileSupported: 0
             DeviceName: 'MyDevice'
               DeviceID: 1
      ObjectConstructor: 'videoinput('mydeviceimaq', 1)'
       SupportedFormats: {'PAL'  'RS170'}
```

The following table describes the information in each field, with references to other sections that provide more information.

Field	Description
DefaultFormat	Text string that specifies the default format used by the device. You define the default format when you add the IDeviceFormat object to the IDeviceInfo object; see "Specifying the Default Format" on page 3-14.
DeviceFileSupported	Boolean value that tells whether the device supports device configuration files (also known as camera files). You use the setDeviceFileSupport() member function of the IDeviceInfo object to set this value; see "Configuring Device Configuration File (Camera File) Support" on page 3-14.
DeviceName	Text string that identifies a particular device. You define this value when you create the IDeviceInfo object; see "Storing Device Information" on page 3-12.
DeviceID	Numeric value that uniquely identifies a particular device. You define this value when you create the IDeviceInfo object; see "Storing Device Information" on page 3-12.
ObjectConstructor	Text string that contains the videoinput function syntax required to create an object instance for this device. The toolbox engine creates this string.
SupportedFormats	Cell array of strings that identifies the formats this device supports. You define this value when you create the IDeviceFormat objects associated with a particular device; see "Storing Format Information" on page 3-13.

Defining Classes to Hold Device-Specific Information

You might want to store more information about a device or format than the `IDeviceInfo` and `IDeviceFormat` objects allow. One way to do this is to define a new class that contains this additional information. Then, in your adaptor, instantiate an object of this class and store it in the adaptor data of the `IDeviceInfo` or `IDeviceFormat` objects. Using adaptor data is a good way to pass important information around inside your adaptor because the `IDeviceInfo` and `IDeviceFormat` objects are passed to other adaptor functions.

Using adaptor data is a three-step process:

1. Define a class to hold the device or format information. See "Defining a Device or Format Information Class" on page 3-18 for more information.

2. Instantiate an object of this class in your adaptor. Use the constructor you define for your class.

3. Store the object in the adaptor data of the `IDeviceInfo` or `IDeviceFormat` object. See "Storing Adaptor Data" on page 3-18 for more information.

Defining a Device or Format Information Class

The class that you define to store additional device or format information must be derived from the `IMAQinterface` class. Subclassing the `IMAQInterface` class ensures that all memory deallocations for these classes are routed through the toolbox engine.

For an example of such a class, see the `DemoDeviceFormat` class in the demo adaptor, defined in the file `DemoDeviceFormat.h`.

Storing Adaptor Data

To store your device or format class in the adaptor data of an `IDeviceInfo` or `IDeviceFormat` object, use the `setAdaptorData()` member function of the object.

Note The objects you store in adaptor data are automatically destroyed when the IDeviceInfo and IDeviceFormat objects are destroyed. Once you store an object in adaptor data, do not try to destroy the objects yourself.

The demo adaptor provides an example, defining a class to hold additional format information. This class, named DemoDeviceFormat, stores format information such as width, height, and color space. The following example, taken from the demo adaptor, shows how to instantiate an object of this derived class, assign values to the data members of the class, and then store the object in the adaptor data of the IDeviceFormat object.

```
DemoDeviceFormat* rgbFormatInfo = new DemoDeviceFormat();

rgbFormatInfo->setFormatWidth(demo::RGB_FORMAT_WIDTH);
rgbFormatInfo->setFormatHeight(demo::RGB_FORMAT_HEIGHT);
rgbFormatInfo->setFormatNumBands(demo::RGB_FORMAT_BANDS);
rgbFormatInfo->setFormatColorSpace(imaqkit::colorspaces::RGB);

deviceFormat->setAdaptorData(rgbFormatInfo);
```

Accessing Adaptor Data

To access the adaptor data stored in an IDeviceInfo or IDeviceFormat object, use the getAdaptorData() member function of the object.

The following example, taken from the demo adaptor, shows how to retrieve the adaptor data from the IDeviceFormat object. In the example, selectedFormat is a DemoDeviceFormat object. Note that because the getAdaptorData() member function returns a handle to the IMAQInterface class, you must cast the returned object to your defined class.

```
dynamic_cast<DemoDeviceFormat*>(selectedFormat->getAdaptorData());
```

Unloading Your Adaptor DLL

Every adaptor must include an `uninitializeAdaptor()` function. The engine calls this function when a user resets the toolbox, by calling the `imaqreset` function, or exits MATLAB.

Your adaptor's implementation of this function depends upon the requirements of your hardware. Every adaptor must include the `uninitializeAdaptor()` function, even if it is an empty implementation.

Example

As defined by the adaptor kit, the `uninitializeAdaptor()` function accepts no arguments and does not return a value. The following example implements an empty `initializeAdaptor()` function.

```
void uninitializeAdaptor()
{
     // Perform any cleanup your hardware requires.
}
```

Returning Warnings and Errors to MATLAB

To return error or warning messages from your adaptor to the MATLAB command line, use the `adaptorError()` and `adaptorWarning()` functions. These functions implement an interface similar to the MATLAB `error` and `warning` functions. Using these functions, you can display a text message at the MATLAB command line.

You must also include a message ID in your message using the format

```
<component>[:<component>]:<mnemonic>
```

where `<component>` and `<mnemonic>` are alphanumeric strings (for example, `'MATLAB:divideByZero'`). The identifier can be used to enable or disable display of the identified warning. For more information, type `help warning` or `help error` at the MATLAB command line.

The following example outputs a warning message to the MATLAB command line.

```
imaqkit::adaptorWarn("MyDeviceAdaptor:constructor","In
constructor");
```

Defining Your Adaptor Class

This chapter describes how to define your adaptor class and instantiate an object of this class. Every adaptor must define a class that is a subclass of the adaptor kit `IAdaptor` abstract class. This abstract class defines several virtual functions that your adaptor class must implement. This chapter gets you started with an adaptor class implementation by creating a stub implementation. This stub implementation will enable you to create a video input object with your adaptor using the `videoinput` function. In subsequent chapters, you complete adaptor development by fleshing out the implementations of these virtual functions.

Overview (p. 4-2)	Provides an overview of the user tasks your adaptor implements in its adaptor class
Summary of IAdaptor Abstract Class Virtual Functions (p. 4-3)	Lists the pure virtual functions in the IAdaptor abstract class.
Creating a Stub Implementation of Your Adaptor Class (p. 4-5)	Provides a step-by-step approach to creating a stub implementation of your adaptor class
Identifying Video Sources (p. 4-10)	Describes how to identify a video source
Instantiating an Adaptor Object (p. 4-12)	Describes how to create your adaptor class constructor

Overview

When a user calls the `videoinput` function to create a video input object, the toolbox engine calls two of the exported functions in your adaptor:

- `getDeviceAttributes()`
- `createInstance()`

(To see a flow-of-control diagram that shows how these functions fit with the other required exported functions, see "Adaptor Exported Functions: An Overview" on page 3-3.)

The `getDeviceAttributes()` function defines which properties of the device that you want to expose to users. This function is described only briefly in this chapter (see "Identifying Video Sources" on page 4-10). For complete information about implementing this exported function, see

The toolbox engine calls your adaptor's `createInstance()` function to instantiate an object of the adaptor class. Every adaptor must define a class that is a subclass of the `IAdaptor` abstract class, providing implementations of the pure virtual functions defined in this abstract class.

This chapter describes how to create a stub implementation of your adaptor class (see "Creating a Stub Implementation of Your Adaptor Class" on page 4-5) and create the constructor and destructor for this class, see "Instantiating an Adaptor Object" on page 4-12. In Chapter 5, "Acquiring Image Data" you flesh out the implementation of these functions.

Note Because each instance of your adaptor class is associated with a specific format selected by the user, most of the information returned by these functions is static.

Summary of IAdaptor Abstract Class Virtual Functions

The following table lists the pure virtual functions defined by the IAdaptor abstract class, in alphabetical order.

Pure Virtual Function	Description with Declaration
`closeDevice()`	Terminates the connection to a device — see "Suggested Algorithm for closeDevice()" on page 5-13. `virtual bool closeDevice();`
`getDriverDescription()`	Returns a character string identifying the device driver used by the device — see "Specifying Device Driver Identification Information" on page 5-35. `virtual const char* getDriverDescription() const;`
`getDriverVersion()`	Returns a character string identifying the version number of the device driver used by the device — see "Specifying Device Driver Identification Information" on page 5-35. `virtual const char* getDriverVersion() const;`
`getFrameType()`	Returns the toolbox-defined frame type used to store the images provided by the device — see "Specifying Frame Type" on page 5-7. `imaqkit::frametypes::FRAMETYPE getFrameType() const;`
`getMaxHeight()`	Returns an integer specifying the maximum vertical resolution (the number of lines) of the image data — see "Specifying Image Dimensions" on page 5-5. `virtual int getMaxHeight() const;`
`getMaxWidth()`	Returns an integer specifying the maximum horizontal resolution (in pixels) of the image data — see "Specifying Image Dimensions" on page 5-5. `virtual int getMaxWidth() const;`
`getNumberOfBands()`	Returns the number of bands used in the returned image data — see "Specifying Image Dimensions" on page 5-5. `virtual int getNumberOfBands() const;`

Pure Virtual Function	Description with Declaration
`openDevice()`	Opens a connection with the device, preparing it for use — see "Opening and Closing a Connection with a Device" on page 5-10. `virtual bool openDevice();`
`startCapture()`	Starts retrieving frames from the device — see "Starting and Stopping Image Acquisition" on page 5-15. `virtual bool startCapture();`
`stopCapture()`	Stops retrieving frames from the device — see "Suggested Algorithm for stopCapture()" on page 5-17. `virtual bool stopCapture();`

Creating a Stub Implementation of Your Adaptor Class

To create a stub implementation of your adaptor class, follow this procedure:

1 Add a C++ header file to the adaptor C++ project. This header file will hold the definition of your adaptor class. You can give your class any name. This example uses the following naming convention:

```
vendor_name + adaptor
```

For this example, the header file that contains the adaptor class definition is named `MyDeviceAdaptor.h`.

2 Copy the following class definition into the header file. This adaptor class contains all the virtual functions defined by the `IAdaptor` abstract class.

```
#include "mwadaptorimaq.h" // required header

class MyDeviceAdaptor : public imaqkit::IAdaptor {

public:

   // Constructor and Destructor
   MyDeviceAdaptor(imaqkit::IEngine* engine,
                   imaqkit::IDeviceInfo* deviceInfo,
                   const char* formatName);

   virtual ~MyDeviceAdaptor();

   // Adaptor and Image Information Functions
   virtual const char* getDriverDescription() const;
   virtual const char* getDriverVersion() const;
   virtual int getMaxWidth() const;
   virtual int getMaxHeight() const;
   virtual int getNumberOfBands() const;
   virtual imaqkit::frametypes::FRAMETYPE getFrameType() const;

   // Image Acquisition Functions
   virtual bool openDevice();
   virtual bool closeDevice();
```

```
    virtual bool startCapture();
    virtual bool stopCapture();

};
```

3 Add a C++ source file to the adaptor project. You can give the source file any name. This example names the file `mydeviceadaptor.cpp`.

4 Copy the following stub implementations of all the adaptor virtual functions into the C++ source file.

```
#include "MyDeviceAdaptor.h"
#include "mwadaptorimaq.h"

// Class constructor
MyDeviceAdaptor::MyDeviceAdaptor(imaqkit::IEngine* engine,
                         imaqkit::IDeviceInfo* deviceInfo,
    const char* formatName):imaqkit::IAdaptor(engine){
}

// Class destructor
MyDeviceAdaptor::~MyDeviceAdaptor(){
}

// Device driver information functions
const char* MyDeviceAdaptor::getDriverDescription() const{
 return "MyDevice_Driver";
}
const char* MyDeviceAdaptor::getDriverVersion() const {
 return "1.0.0";
}

// Image data information functions
int MyDeviceAdaptor::getMaxWidth() const { return 640;}
int MyDeviceAdaptor::getMaxHeight() const { return 480;}
int MyDeviceAdaptor::getNumberOfBands() const { return 1;}

imaqkit::frametypes::FRAMETYPE MyDeviceAdaptor::getFrameType()
              const {
    return imaqkit::frametypes::MONO8;
```

```
}

// Image acquisition functions
bool MyDeviceAdaptor::openDevice() {return true;}
bool MyDeviceAdaptor::closeDevice(){return true;}
bool MyDeviceAdaptor::startCapture(){return true;}
bool MyDeviceAdaptor::stopCapture(){return true;}
```

5. After defining the adaptor class and creating stub implementations of all its member functions, you must make several edits to the file containing the exported functions, `mydevice_exported_fcns.cpp`, that you created in Chapter 3, "Providing Hardware Information".

6. Add a reference to your adaptor class header file. This is needed because the `createInstance()` exported function instantiates an object of this class.

```
#include "MyDeviceAdaptor.h"
```

7. Copy the following stub implementations of the `getDeviceAttributes()` and `createInstance()` functions into the exported functions source file, `mydevice_exported_fcns.cpp`. The `getDeviceAttributes()` function defines a video source. This is required to enable creation of a video input object. See "Identifying Video Sources" on page 4-10 for more information.

```
void getDeviceAttributes(const imaqkit::IDeviceInfo* deviceInfo,
           const char* sourceType,
         imaqkit::IPropFactory* devicePropFact,
       imaqkit::IVideoSourceInfo* sourceContainer,
         imaqkit::ITriggerInfo* hwTriggerInfo){

   // Create a video source
  sourceContainer->addAdaptorSource("MyDeviceSource", 1);
}

imaqkit::IAdaptor* createInstance(imaqkit::IEngine* engine,
        imaqkit::IDeviceInfo* deviceInfo,
           char* sourceType){

 imaqkit::IAdaptor* adaptor = new
                  MyDeviceAdaptor(engine,deviceInfo,sourceType);
```

```
        return  adaptor;
    }
```

8 Export the `getDeviceAttributes()` and `createInstance()` functions by adding them to the adaptor module definition file. The file should look like this:

```
LIBRARY mydeviceimaq
EXPORTS
    initializeAdaptor   @1
    getAvailHW          @2
    uninitializeAdaptor @3
    getDeviceAttributes @4
    createInstance      @5
```

9 Build the adaptor DLL. Select the **Build Solution** option on the **Build** menu.

10 Start MATLAB.

11 Call the `imaqhwinfo` function. Note how the adaptor, named `mydeviceimaq`, is included in the list of available adaptors returned by `imaqhwinfo`. If you have not previously registered your adaptor DLL, register your adaptor with the toolbox — see "Registering an Adaptor with the Toolbox" on page 1-11. To view more detailed information about your adaptor, call `imaqhwinfo` again with this syntax:

```
dev_info = imaqhwinfo('mydeviceimaq');
```

12 Create a video input object for the `mydeviceimaq` adaptor, using the `videoinput` function.

Note While you can create a video input object with your adaptor, you cannot use it to acquire video from a device. You must implement the adaptor class acquisition functions to do that. See Chapter 5, "Acquiring Image Data" for more information.

```
vid = videoinput('mydeviceimaq',1)
```

```
Summary of Video Input Object Using 'MyDevice'.

   Acquisition Source(s): MyDeviceSource is available.

  Acquisition Parameters: 'MyDeviceSource' is the current selected source.
                         10 frames per trigger using the selected source.
                           '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.
```

Identifying Video Sources

The toolbox defines a video source as one or more hardware inputs that are treated as a single entity. For example, an image acquisition device might support an RGB source that is made up of three physical connections. The toolbox would treat the three connections as a single video source. Read the documentation that came with your device to determine the video sources it supports.

When a user creates a video input object, the toolbox engine automatically creates a video source object for each source supported by an adaptor. The `Source` property of the video input object lists the available video sources. The video source object that is used to acquire data is called the currently selected video source. By default, the toolbox engine uses the first video source you define as the selected source, but users can switch the selected source by setting the value of the video input object's `SelectedSourceName` property.

Suggested Algorithm

Your adaptor's `getDeviceAttributes()` function must define all the properties and sources of video data you want to make available to users. This section only covers defining video sources, which means determining the text labels used to identify the available video sources. For information about making device properties available to users, see Chapter 6, "Defining Device-Specific Properties"

Note Every adaptor must specify at least one video source; otherwise, you cannot create a video input object if a video source has not been specified.

You use the `addAdaptorSource()` member function of the `IVideoSourceInfo` object that the toolbox engine passes to your adaptor's `getDeviceAttributes()` function to define a video source. You specify the following two arguments:

- Name you want to assign to the source
- ID you want to assign to the source

The name is visible to users. Choose a name that clearly identifies the source. If the device vendor assigns names to the sources, you can use the same names. For example, Matrox some devices identify video sources by the labels `ch0`, `ch1`, etc.

Because the ID is not exposed to users, you can specify any convenient value. For example, if the device's SDK uses numerical identifiers to indicate a video source, use these values for your source IDs.

For example, this code specifies a video source.

```
sourceContainer->addAdaptorSource("MyDeviceSource",1)
```

You can use `IVideoSourceInfo` member functions to perform many tasks, such as determining the currently selected source. For more information about this class, see the Image Acquisition Toolbox Adaptor Kit API Reference documentation.

Instantiating an Adaptor Object

Every adaptor must include a `createInstance()` function. The engine calls this function to instantiate an object of your adaptor's class. This section includes the following topics:

- "Suggested Algorithm" on page 4-12
- "Implementing Your Adaptor Class Constructor" on page 4-13
- "Implementing Your Adaptor Class Destructor" on page 4-14

Suggested Algorithm

The algorithm for the `createInstance()` function is simple: call the adaptor class constructor to instantiate an object of an adaptor class and return a handle to the object. The engine passes these arguments to your adaptor's `createInstance()` function. The `createInstance()` function accepts three arguments:

```
imaqkit::IAdaptor* createInstance(imaqkit::IEngine* engine,
                       imaqkit::DeviceInfo* deviceInfo,
                              const char* FormatName)
```

The following table describes these arguments. Your adaptor's `createInstance()` function must return a handle to an `IAdaptor` object.

Argument	Purpose
`engine`	Handle to an `IEngine` object that enables your adaptor to communicate with the engine.
`deviceInfo`	Handle to an `IDeviceInfo` object that represents the characteristics of a particular device. This object will be one of the `IDeviceInfo` objects you created in your `getAvailHW()` function.
`formatName`	A text string that specifies the name of a video format supported by the device or the full path of a device configuration file. If this specifies a format, it must be one of the formats represented by the `IDeviceFormat` objects you created in your `getAvailHW()` function.

Implementing Your Adaptor Class Constructor

Because you write the code that calls your adaptor class constructor, you can define the arguments accepted by your adaptor class constructor. At a minimum, adaptor constructors must accept a handle to an `IEngine` object that represents the connection between the engine and your adaptor. This is defined by the `IAdaptor` superclass. Your adaptor uses this handle to access engine functions for packaging image frames and returning them to the engine.

In addition to this required argument, many adaptors also accept two other arguments

- Handle to an `IDeviceInfo` object that specifies the device to which you want to connect
- Text string specifying the desired acquisition source format or the full path to a device configuration file (also known as a camera file)

These are the same arguments passed to your adaptor's `createInstance()` function.

Suggested Algorithm

The requirements of your image acquisition device will determine what your class constructor must do. Class constructors typically perform tasks that only need to be performed once by the class, such as

- Setting up listeners for all device-specific properties. Listeners notify the class when a user changes the value of a device-specific property. See "Setting Up Property Listeners" on page 6-11.
- Creating a critical section object. Your adaptor will use the critical section to protect data members that might be accessed from multiple threads. See "Using Critical Sections" on page 5-31.

Note Your class constructor should not perform any device initialization, because a user might want to create multiple video input objects. Device initialization occurs when a user has requested frames — see "Opening and Closing a Connection with a Device" on page 5-10.

Example

The following example shows a `createInstance()` function that instantiates an object of class `MyDeviceAdaptor`.

```
imaqkit::IAdaptor* createInstance(imaqkit::IEngine* engine,
                                  imaqkit::IDeviceInfo* deviceInfo,
                                                   char* formatName) {

// Instantiate an object of your IAdaptor-derived class

   imaqkit::IAdaptor* adaptor = new
                  MyDeviceAdaptor(engine,deviceInfo,formatName);

   return  adaptor;
}
```

Implementing Your Adaptor Class Destructor

This destructor is invoked whenever the associated video input object in MATLAB is deleted.

```
delete(vid);
```

A destructor for a class cannot take parameters or return a value. An adaptor class, as a derived class, must contain a destructor and the destructor must be declared as virtual.

```
virtual ~MyAdaptor();
```

Suggested Algorithm

The design of your adaptor class and the requirements of your image acquisition device will determine what tasks your class destructor must perform. Your class must contain a destructor even if it is an empty implementation. Some examples of tasks a destructor might perform include:

- Stopping the device, if it is currently acquiring frames — see "Suggested Algorithm for stopCapture()" on page 5-17.
- Closing the connection with the device — see "Suggested Algorithm for closeDevice()" on page 5-13.

- Deleting the critical section object — see "Using Critical Sections" on page 5-31.

Example

This example shows a skeletal implementation of a destructor. For a more complete example, see the demo adaptor class.

```
MyAdaptor::~MyAdaptor(){

}
```

5

Acquiring Image Data

This chapter describes how to implement the adaptor member functions to perform image acquisition. After completing the tasks outlined in this chapter, you will be able to create a video input object, start it, and trigger an acquisition.

Overview (p. 5-2)	Provides an overview of how your adaptor acquires image frames.
Specifying the Format of the Image Data (p. 5-5)	Describes how to specify the format of the acquired image data
Opening and Closing a Connection with a Device (p. 5-10)	Describes how to open a connection with a device
Starting and Stopping Image Acquisition (p. 5-15)	Describes how to start a video input object.
Implementing the Acquisition Thread Function (p. 5-19)	Describes how to implement the main image acquisition function
Supporting ROIs (p. 5-25)	Describes how to support the specification of a region-of-interest (ROI) in software and in hardware
Supporting Hardware Triggers (p. 5-28)	Describes how to support hardware triggers
Using Critical Sections (p. 5-31)	Describes how to create and use critical section objects
Specifying Device Driver Identification Information (p. 5-35)	Describes how to provide device driver information

Overview

After completing chapters 3 and 4, you can see your adaptor included in the list of adaptors returned by `imaqhwinfo` and you can create a video input object using the `videoinput` function. Now it's time to acquire data from your device. In this chapter, you flesh out the stub implementations of the adaptor class virtual functions that work together to acquire data.

User Scenario

The following example shows how a toolbox user initiates the acquisition of image frames. The example calls the `videoinput` function to create a video input object and then calls the `start` function to start the object. Note in the summary that ten image frames are acquired.

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

start(vid);
vid

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 source.
                           'RGB555_128x96' 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.
                           10 frames acquired since starting.
                           10 frames available for GETDATA.
```

Triggering

In the previous example, the `start` function opens the connection with the device but does not actually cause the acquisition of image data. The toolbox

uses triggers to control image acquisition. By default, video input objects are configured with an immediate trigger so, in the example, when you start the object, an immediate trigger fires.

The toolbox also supports two other types of triggers: manual and hardware. With a manual trigger, after starting a video input object, you must call the `trigger` function to acquire data. With hardware triggers, you start the object and it waits until it receives a signal from an external device to start acquiring data.

The toolbox handles immediate and manual triggering automatically; you do not have to include any special processing in your adaptor. Supporting hardware triggers, requires some adaptor development work. For more information, see "Supporting Hardware Triggers" on page 5-28.

Overview of Virtual Functions Used to Acquire Data

The pure virtual functions in your adaptor class that you must implement work together to acquire data. However, the four main steps are:

1. Specify the format of the video data in the `getMaxHeight()`, `getMaxWidth()`, `getNumberOfBands()`, and `getFrameType()` functions — see "Specifying the Format of the Image Data" on page 5-5.

2. Open a connection with your device in the `openDevice()` function — see "Opening and Closing a Connection with a Device" on page 5-10.

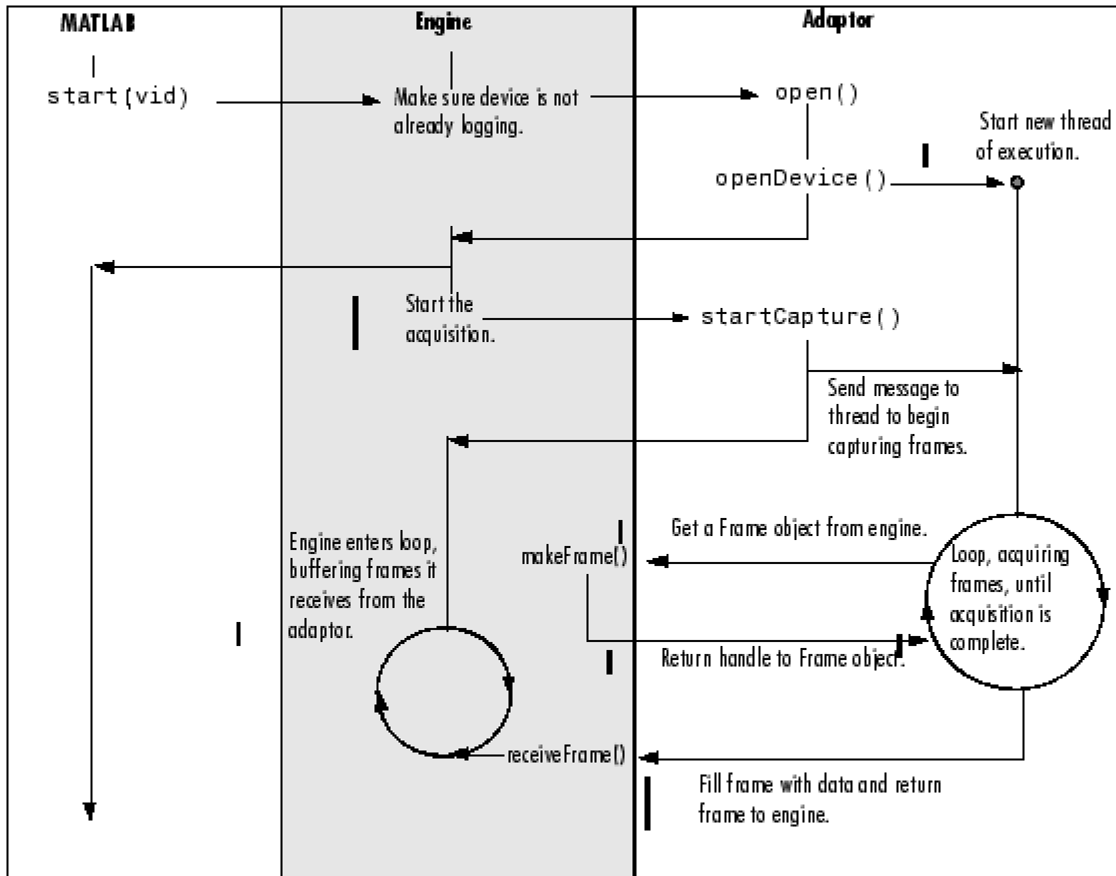
3. Start acquiring data in the `startCapture()` function — see "Starting and Stopping Image Acquisition" on page 5-15.

4. Stop acquiring data in the `stopCapture()` function — see "Starting and Stopping Image Acquisition" on page 5-15.

5. Close the connection with the device in the `closeDevice()` function — see "Opening and Closing a Connection with a Device" on page 5-10.

The following diagram shows this flow of control in graphical form. This diagram picks up where the diagram in Chapter 3 ends, after the object has been created — see "Overview" on page 5-2.

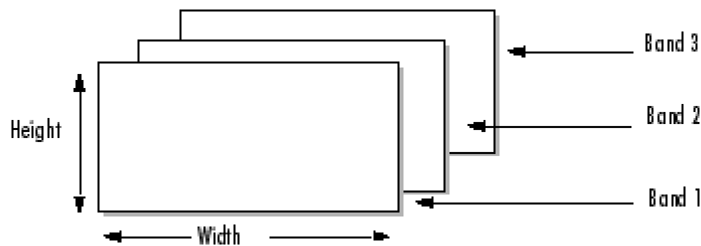
Note The diagrams do not show the calls your adaptor makes to the image acquisition device's SDK because these calls vary with each device's SDK.



Flow of Control among the Adaptor Acquisition Functions

Specifying the Format of the Image Data

Before you can acquire data from your device, you must tell the engine the format of the data it can expect to receive from your device. Without this information, the engine does not know how to interpret the data. For example, the engine needs to know the size of the bytes used to store image data, the length of each line and the total number of lines in each image frame, and the number of planes, or bands, in each image frame. (e.g. RGB data has three bands). The following figure illustrates this information.



In some cases, this format information is determined by external standards, such as the RS-170/NTSC standard. In other cases, device vendors define many different formats, described in the documentation that comes with the device. Adaptor writers decide which of these supported formats they want to make available to users of their adaptor in their `getAvailHW()` function, described in “Storing Format Information” on page 3-13.

This section describes how you specify format information in your adaptor after using the adaptor class virtual functions.

- “Specifying Image Dimensions” on page 5-5
- “Specifying Frame Type” on page 5-7

Specifying Image Dimensions

You specify the dimensions of the image data a device outputs using the following virtual functions.

- `getMaxHeight()` — Returns an integer that specifies the maximum height of the image data.

- `getMaxWidth()` — Returns an integer that specifies the maximum width of the image data.
- `getNumberOfBands()` — Returns an integer that specifies the number of dimensions in the data. For example, RGB formats use three bands.

The engine calls these functions in your adaptor to get the resolution information that it displays in the `VideoResolution` property of the video input object.

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

get(vid,'VideoResolution')

ans =

     640   480
```

Your adaptor also call these functions when it creates the `IAdaptorFrame` object to receive image data. See "Implementing the Acquisition Thread Function" on page 5-19 for more information.

Suggested Algorithm

The `getMaxHeight()`, `getMaxWidth()`, and `getNumberOfBands()` functions in an adaptor typically perform the following processing:

1. Determine the format specified by the user when they created the video input object. The engine passes this information as an argument to your adaptor's `createInstance()` function.

2. Based on the format chosen, return the appropriate values of the height, width, or number of bands. Your adaptor can accomplish this in many ways. One way, illustrated by the demo adaptor, is to determine these values in your `getAvailHW()` function and store the information in application data in the `IDeviceFormat` object — see "Defining Classes to Hold Device-Specific Information" on page 3-18. Then, the `getMaxHeight()`, `getMaxWidth()`, and `getNumberOfBands()` functions can retrieve this application data and read these values.

Example

The following implementations of the `getMaxHeight()` and `getMaxWidth()` functions determine the value based on the format specified by the user. The number of bands depends on whether the format is color or monochrome. For color formats, such as RGB and YUV, the number of bands is always 3. For monochrome (black and white) formats, the number of bands is always 1. The Image Acquisition Toolbox only supports image data with 1 or 3 bands.

Replace the stub implementations in the example adaptor with the following code C++ file, `mydevice.cpp`, created in Chapter 3. The values are appropriate for the format names specified in the example in “Specifying Device and Format Information” on page 3-9.

```
int MyDeviceAdaptor::getMaxHeight() const{
 if(strcmp(_formatName,"RS170"){
  return 480;
 } else {
  return 576;
}

int MyDeviceAdaptor::getMaxWidth() const {
 if(strcmp(_formatName,"RS170"){
  return 640;
 } else {
  return 768;
 }
}
int MyDeviceAdaptor::getNumberOfBands() const {

 return 1;
}
```

Specifying Frame Type

In addition to the image frame dimensions, you must provide the engine with information about the byte layout of the image data. Byte layout includes the number of bits used to represent pixel values, whether the data is signed or unsigned, the endianness of the data, and whether the device sends the bottom row first.

To specify this information, you must select one of the FRAMETYPE enumerations defined by the adaptor kit. The adaptor kit defines enumerations for many different frame types to represent the wide variety of formats supported by devices. For example, if your device is a monochrome (black and white) device that returns 8-bit data, you might choose the MONO8 frame type. If your device is a color device that returns 24-bit data, you might choose the RGB24 frame type. The following table summarizes the frame types that are available. To choose a specific format, view the list in the Image Acquisition Toolbox Adaptor Kit API Reference documentation or open the AdaptorFrameTypes.h file.

Format	Frame Types
Monochrome	8-, 10-, 12-, and 16-bit formats; both little-endian and big-endian; in regular and flip formats. (In flip formats, the device delivers the bottom line first.)
	Signed 16- and 32-bit formats; both little-endian and big-endian; in regular and flip formats.
	Floating-point and double formats; both little-endian and big-endian formats; in regular and flip formats.
Color	8-, 24-, 32-, and 48-bit RGB formats; both little-endian and big-endian; regular and flip; packed and planar (see "Understanding Packed and Planar Formats" on page 5-9).
	Frame types that specify the order of the bytes of color data (RGB or GBR) and specify where the blank byte is located (XRGB or XGBR).
	Formats that represent colors in 4-bits (4444), 5-bits (555), 5- or 6-bits (565), or 10-bits (101010).
	Formats that use the YUV color space.

Suggested Algorithm

Your adaptor's getFrameType() function must return the appropriate frame type that describes the data returned by your device for the specified format.

If your device supports multiple color formats, you do not need to expose all the formats to toolbox users. You can simply provide one color format and handle the low-level details in your adaptor with FRAMETYPE.

Example

The following example shows a skeletal implementation of the getFrameType() function. An actual implementation might select the frame type based on the format the user selected.

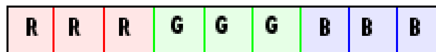
```
virtual imaqkit::frametypes::FRAMETYPE getFrameType() const {

 return imaqkit::frametypes::FRAMETYPE:MONO8;
}
```

Understanding Packed and Planar Formats

The adaptor kit IAdaptorFrame class defines many FRAMETYPE enumerations that cover the many possible types of image data devices can return. For example, some devices can return color images in packed or nonpacked (planar) formats. These formats describe how the bytes of red, green, and blue data are arranged in memory. In packed formats, the red, green, and blue triplets are grouped together. In nonpacked formats, all the red data is stored together, followed by all the green data, followed by all the blue data. The following figure illustrates this distinction.

Packed Format

Nonpacked (Planar) Format



Packed and Planar Formats

To get more information about video formats, go to the fourcc.org Web site.

Opening and Closing a Connection with a Device

Adaptors typically open a connection with the device in their `openDevice()` function and close the connection in their `closeDevice()` function. For most devices, opening a connection to the device reserves it for exclusive use. Closing the device releases the device.

Note The toolbox engine actually calls the `IAdaptor` class `open()` member function to open a connection with a device and the `close()` function to close a connection with a device. These function then call your adaptor's `openDevice()` and `closeDevice()` functions. If your adaptor needs to open or close a device, use the `open()` and `close()` functions, rather than calling `openDevice()` or `closeDevice()` directly.

Suggested Algorithm for openDevice()

The `openDevice()` function typically performs the following tasks.

1. Test whether the device is already open by calling the `IAdaptor` class `isOpen()` function. If the device is already open, your `openDevice()` function should return `true`. If the device is not already open, your `openDevice()` function should establish a connection to the device using device SDK calls.

2. Start the acquisition thread. See "Starting an Acquisition Thread" on page 5-11 for more information.

Note Starting a separate thread is only required if your adaptor uses a thread-based design. Adaptors can also use asynchronous interrupts (callbacks) to acquire frames, if the device supports this. In this scenario, adaptors receive notification asynchronously when data is available. For information about using this method, refer to the documentation for your device's SDK.

Starting an Acquisition Thread

To start an acquisition thread, use the Windows `CreateThread()` function. The `CreateThread()` function creates a thread that executes within the virtual address space of the calling process.

The `CreateThread()` function accepts these parameters.

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

For an adaptor, the following table lists the parameters you must set. For complete information about creating a thread, see the documentation at the Microsoft Developer Network Web site (`msdn.microsoft.com`).

Parameter	Description
`lpStartAddress`	Address of the acquisition thread procedure. Specify the name of the thread procedure declared in your adaptor class header file. See "Implementing the Acquisition Thread Function" on page 5-19.
`lpParameter`	Pointer to the object itself, i.e., the `this` pointer.
`lpThreadId`	Address of a variable in which the `CreateThread()` function returns the ID assigned to the newly created thread

After you call the `CreateThread()` function, applications typically call the `PostThreadMessage()` function to send a message to the new thread. This causes the system to create a message queue for the thread. Enter a loop to wait until the thread acknowledges the message was received to ensure that the thread queue has been created. Your adaptor terminates the thread in your adaptor's `closeDevice()` function — see "Suggested Algorithm for closeDevice()" on page 5-13.

Example: Opening a Connection

This example shows a skeletal implementation of an `openDevice()` function.

1. Replace the stub implementation of the `openDevice()` function in the `MyDevice` adaptor with this code.

```
bool MyDeviceAdaptor::openDevice()
{
    // If device is already open, return true.
    if (isOpen())
        return true;

    // Create the image acquistion thread.
    _acquireThread = CreateThread(NULL,
                                      0,
                            acquireThread,
                                   this,
                                      0,
                       &_acquireThreadID);
    if (_acquireThread == NULL) {
        closeDevice();
        return false;
    }

    // Wait for thread to create message queue.
    while(PostThreadMessage(_acquireThreadID,WM_USER+1,0,0) == 0)
        Sleep(1);

    return true;
}
```

2. To be able to compile and link your adaptor, you must create a stub implementation of your `acquireThread()` function and add it to your adaptor. You can fill in the complete implementation later — see “Implementing the Acquisition Thread Function” on page 5-19.

```
DWORD WINAPI MyDeviceAdaptor::acquireThread(void* param) {

   MSG msg;
   while (GetMessage(&msg,NULL,0,0) > 0) {
```

```
            switch (msg.message) {
               case WM_USER:

                  // The frame acquisition loop code goes here.
                  adaptor_warn('in acquire thread function\n');

            } // end switch
         } // end while

         return 0;
      } // end acquireThread
```

3 Add declarations of the `acquireThread()` function, the `acquireThread` variable, and the `acquireThreadID` variable as private data members of your adaptor class header file. In this example, `MyDeviceAdaptor.h`.

```
private:
// Declaration of acquisition thread function
static DWORD WINAPI acquireThread(void* param);

// Thread variable
HANDLE _acquireThread;

// Thread ID returned by Windows.
DWORD _acquireThreadID;
```

4 Compile and link your adaptor. You should be able to create a video input object. When you call the `start` function, verify that your adaptor successfully created the acquisition thread.

Suggested Algorithm for closeDevice()

The `closeDevice()` function typically performs the following tasks.

1 Test whether the device is already closed. If it is, exit.

2 Post a message to the acquisition thread to quit and wait until it returns before exiting, for adaptors with thread-based designs. For more information about posting a message to the thread, see “Sending a Message to the Acquisition Thread” on page 5-16.

3 Close the handle associated with the acquisition thread and reset the thread handle variable to NULL.

Example: Closing the Connection with a Device

The example shows a skeletal implementation of the closeDevice() function.

```
bool MyDeviceAdaptor::closeDevice(){

 // If the device is not open, return.
    if (!isOpen())
        return true;

    // Terminate and close the acquisition thread.
    if (_acquireThread) {
        // Send WM_QUIT message to thread.
        PostThreadMessage(_acquireThreadID, WM_QUIT, 0, 0);

        // Give the thread a chance to finish.
        WaitForSingleObject(_acquireThread, 10000);

        // Close thread handle.
        CloseHandle(_acquireThread);
        _acquireThread = NULL;
    }
 return true;
}
```

Starting and Stopping Image Acquisition

Once `openDevice()` returns successfully, the engine calls your adaptor's `startCapture()` function to start acquiring data.

The engine calls your adaptor's `stopCapture()` function when a user calls the `stop` or `closepreview` function on a video input object, or when the specified number of frames has been acquired and the acquisition is complete. For example,

```
vid = videoinput('winvideo',1);
set(vid,'FramesPerTrigger',1000); //
start(vid);
stop(vid);
```

Suggested Algorithm for startCapture()

The `startCapture()` function typically performs the following tasks.

1. Check whether an acquisition is already occurring, using the `IAdaptor` member function `isAcquiring()`. If it is, exit.

2. Send a message to the acquisition thread, using the Windows `PostThreadMessage()` function, telling it to begin acquiring image frames from the device. See "Sending a Message to the Acquisition Thread" on page 5-16 for more information.

Note Sending a start message to the acquisition thread is only required if your adaptor uses a thread-based design. Adaptors can also use asynchronous interrupts (callbacks) to acquire frames, if the device supports this. Refer to the documentation that came with your device's SDK for more information.

The `startCapture()` function also typically makes sure that the latest image acquisition object properties are used (see "Setting Up Property Listeners" on page 6-11), and configures hardware triggers, if supported and set (see "Supporting Hardware Triggers" on page 5-28).

Sending a Message to the Acquisition Thread

To send a message to a thread, use the Windows `PostThreadMessage()` function. The adaptor's acquisition thread function uses the Windows `GetMessage()` function to receive these messages — see "Example: Opening a Connection" on page 5-12.

The `PostThreadMessage()` function accepts these parameters:

```
BOOL PostThreadMessage( DWORD idThread,
                        UINT Msg,
                        WPARAM wParam,
                        LPARAM lParam
);
```

The following table describes how to set these parameters for an adaptor. For more information about sending thread messages, see the documentation at the Microsoft Developer Network Web site (`msdn.microsoft.com`).

Parameter	Description
`idThread`	Identifier of the thread to which the message is to be posted, returned by `CreateThread()`.
`Msg`	Message to be posted. Microsoft defines a range of values for user messages, beginning with the value `WM_USER`.
`wParam`	Additional message-specific information
`lParam`	Additional message-specific information

Example: Initiating Acquisition

This example illustrates a simple `startCapture()` function. This function takes no arguments and returns a Boolean value indicating whether the video input object is in `start` state.

1 Replace the stub implementation in the `MyDeviceAdaptor.cpp` file with this code and then rebuild your adaptor.

```
bool MyDeviceAdaptor::startCapture(){
    // Check if device is already acquiring frames.
```

```
        if (! isAcquiring())
            return false;

        // Send start message to acquisition thread
        PostThreadMessage(_acquireThreadID, WM_USER, 0, 0);

        return true;
    }
```

2 Start MATLAB and run your adaptor to verify that your acquisition thread gets the start message `from startCapture()`.

Suggested Algorithm for stopCapture()

The `stopcapture()` function typically performs these tasks.

1 Checks whether the adaptor is already stopped by calling the `isAcquiring()` function. If the device is not currently acquiring data, return `true`.

2 Stops the frame acquisition loop and stops the device, if necessary

---

Note It is important not to exit the `stopCapture()` function while the acquisition thread function is still acquiring frames. One way to do this is to try to acquire a critical section. When you are able to acquire the critical section, you can be sure that the frame acquisition loop has ended, giving up its critical section.

Example

The example shows the `stopCapture()` function from the demo adaptor. The demo adaptor provides one example of how to stop the frame acquisition loop. The adaptor defines a flag variable that it checks each time it enters the frame acquisition loop. To break out of the frame acquisition loop, set this flag variable to false. See the demo adaptor for more details.

This example illustrates a simple `stopCapture()` function. This function takes no arguments and returns a Boolean value indicating whether the video input object is in `stopped` state.

1 Replace the stub implementation in the `MyDeviceAdaptor.cpp` file with this code and then rebuild your adaptor.

```
bool MyDeviceAdaptor::stopCapture(){

// If the device is not acquiring data, return.
   if (!isAcquiring())
      return true;

//**********************************************************
// Insert calls to your device's SDK to stop the device, if
// necessary.
//**********************************************************

   return true;
}
```

Implementing the Acquisition Thread Function

This section describes how to implement your adaptor's acquisition thread function. In a threaded adaptor design, the acquisition thread function performs the actual acquisition of frames from the device. When you create the thread ("Opening and Closing a Connection with a Device" on page 5-10), you specify the name of this acquisition thread function as the starting address for the new thread.

User Scenario

The toolbox engine invokes the acquisition thread function indirectly when a user calls the `start`, `getsnapshot`, or `preview` function. Once called, the acquisition thread function acquires frames until the specified number of frames has been acquired or the user calls the `stop` function.

Suggested Algorithm

Note The design of the acquisition thread function can vary significantly between various adaptors, depending on the requirements of the device's SDK. This section does not describe device-dependent implementation details but rather highlights required tasks that are common to all implementations.

At its highest level, in a threaded design, an acquisition thread function typically contains two loops:

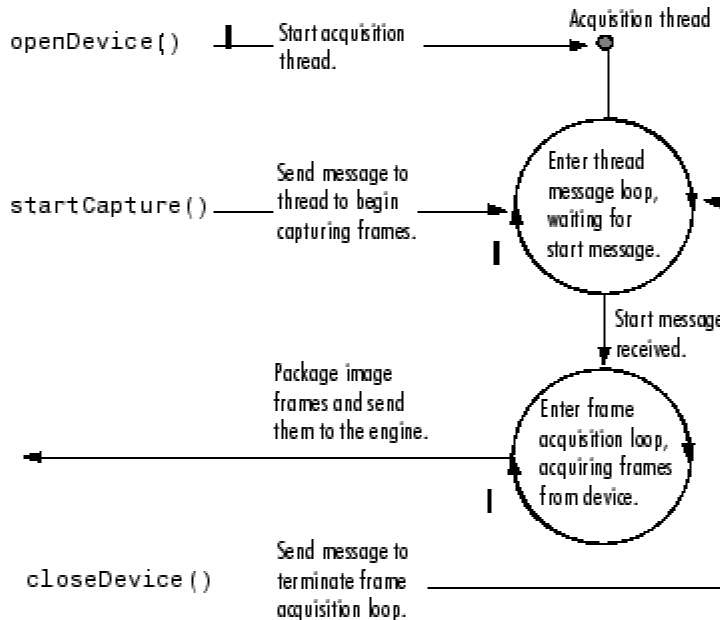
- "Thread Message Loop" on page 5-19
- "Frame Acquisition Loop" on page 5-20

Thread Message Loop

The thread message loop is the main processing loop in the acquisition thread function. When the thread is created, the function enters the thread message loop, waiting for the message to start acquiring frames. Your adaptor's `startCapture()` function sends the message to the acquisition thread, telling it to start acquiring frames. This example uses the `WM_USER` message to

indicate this state. See “Sending a Message to the Acquisition Thread” on page 5-16 for more information.

When it receives the appropriate message, the acquisition thread function enters the frame acquisition loop. The following figure illustrates this interaction between your adaptor functions and the acquisition thread. For information about the frame acquisition loop, see “Frame Acquisition Loop” on page 5-20.



Interaction of Adaptor Functions and Acquisition Thread

Frame Acquisition Loop

The frame acquisition loop is where your adaptor acquires frames from the device and sends them to the engine. This process involves the following steps:

1 Check whether the specified number of frames has been acquired. The frame acquisition loop acquires frames from the device until the specified number of frames has been acquired. Use the `IAdaptor` member function `isAcquisitionNotComplete()` to determine if more frames are needed.

2. If your adaptor supports hardware triggers, you would check whether a hardware trigger is configured here — "Supporting Hardware Triggers" on page 5-28.

3. Grab a frame from the device. This code is completely dependent on your device SDK's API. With many device SDKs, you allocate a buffer and the device fills it with image data. See your device's API documentation to learn how to get frames from your device.

4. Check whether you need to send the acquired frame to the engine, using the `IAdaptor` member function `isSendFrame()`. This is how the toolbox implements the `FrameGrabInterval` property, where users can specify that they only want to acquire every other frame, for example.

   If you need to send a frame to the engine, package the frame in an `IAdaptorFrame` object; otherwise, skip to step 5.

   a. Create a frame object, using the `IEngine` object `makeFrame()` member function. You must specify the image frame dimensions and frame type when you create the frame object.

   b. Put the acquired image data into the frame object, using the `IAdaptorFrame` object `setImage()` member function. You specify a pointer to the buffer that contains the image data, the frame width and height and any offsets from the upper left corner of the image.

   Note For information about specifying frame width, height, and offset with ROIs, see "Supporting ROIs" on page 5-25.

   ---

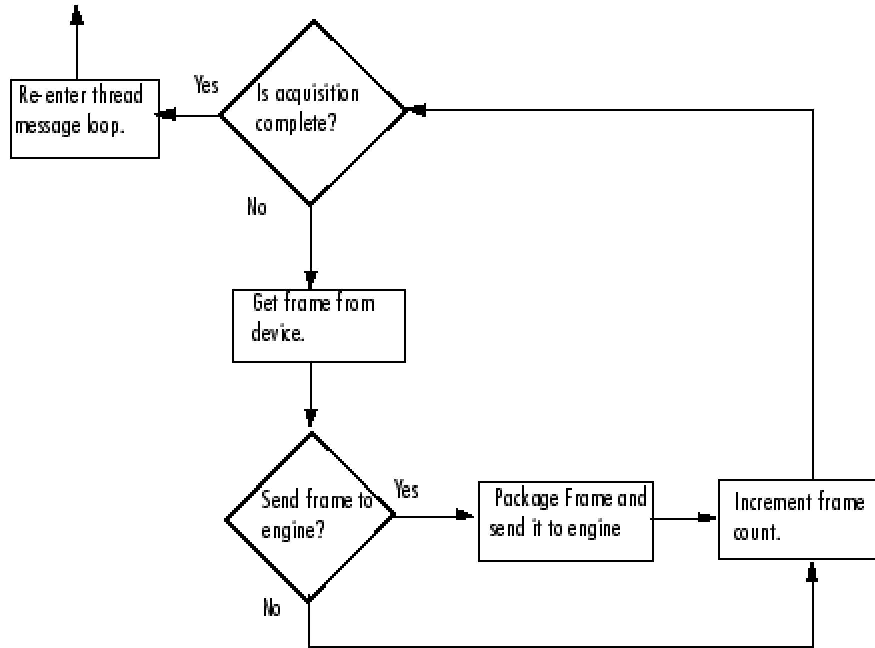
   c. Log the time of the acquisition in the frame object, using the `IAdaptorFrame` member function `setTime()`. Device SDKs sometimes provide access to time stamp information, but you can also use the adaptor kit `getCurrentTime()` function.

   d. Send the packaged frame to the engine, using the `IEngine` member function `receiveFrame()`.

5. Increment the frame count using the `IAdaptor` member function `incrementFrameCount()`. Whether you need to send a frame or not, you must always increment the frame count whenever you acquire a frame.

6 Return to the top of the frame acquisition loop.

The following figure illustrates the frame acquisition loop.



A Possible Algorithm for the Frame Acquisition Loop

Example

The following is a declaration of an acquisition thread function. You can give your acquisition thread procedure any name, such as `acquireThread()`.

```
DWORD WINAPI acquireThread(void* ThreadParam);
```

Your thread function must accept a single parameter, which is defined as a pointer to the object itself, i.e., the `this` pointer. The thread function returns a value that indicates success or failure. For more information, see the documentation at the Microsoft Developer Network Web site (`msdn.microsoft.com`).

The following is an acquisition thread function that you can use with the example `MyDeviceAdaptor`. Replace the skeletal implementation you used in "Starting an Acquisition Thread" on page 5-11 with this code.

```
DWORD WINAPI MyDeviceAdaptor::acquireThread(void* param) {

MyDeviceAdaptor* adaptor = reinterpret_cast<MyDeviceAdaptor*>(param);

MSG msg;
while (GetMessage(&msg,NULL,0,0) > 0) {
    switch (msg.message) {
        case WM_USER:
          // Check if a frame needs to be acquired.
            while(adaptor->isAcquisitionNotComplete()) {

            // Insert Device-specific code here to acquire frames
            // into a buffer.

                if (adaptor->isSendFrame()) {

                    // Get frame type & dimensions.
                    imaqkit::frametypes::FRAMETYPE frameType =
                                        adaptor->getFrameType();
                    int imWidth = adaptor->getMaxWidth();
                    int imHeight = adaptor->getMaxHeight();

                    // Create a frame object.
                    imaqkit::IAdaptorFrame* frame =
                            adaptor->getEngine()->makeFrame(frameType,
                                                             imWidth,
                                                            imHeight);

                    // Copy data from buffer into frame object.
                    frame->setImage(imBuffer,
                                     imWidth,
                                    imHeight,
                                           0, // X Offset from origin
                                           0); // Y Offset from origin

                    // Set image's timestamp.
```

```
                    frame->setTime(imaqkit::getCurrentTime());

                    // Send frame object to engine.
                    adaptor->getEngine()->receiveFrame(frame);
                } // if isSendFrame()

            // Increment the frame count.
            adaptor->incrementFrameCount();

            } // while(isAcquisitionNotComplete()

             break;
        } //switch-case WM_USER
    } //while message is not WM_QUIT

return 0;
}
```

Supporting ROIs

The toolbox supports the specification of regions of interest (ROIs) in both software and hardware.

When using a software ROI, a toolbox user sets the dimensions of the ROI in the `ROIPosition` property. The device returns the entire image frame. Your adaptor specifies the ROI dimensions when it creates the Frame object to package up the image data.

For hardware ROI, the user defines the ROI on the device. The device returns only the data in the specified ROI.

Implementing Software ROI

Users can set the value of the `ROIPosition` property to specify an ROI. Users specify the value as a four-element vector in the form:

```
[Xoffset Yoffset Width Height]
```

The *x*- and *y*-offsets define the position of the ROI in relation to the upper left corner of the image frame. For more information see the toolbox documentation.

Suggested Algorithm

To support software ROI, your adaptor must check the value of the `ROIposition` property before creating the frame object because you need to specify the ROI dimensions when you create the frame.

In your frame acquisition loop, insert the following call to the IAdaptor function `getROI()`. Then, use the ROI width and height values when you create the `IAdaptorFrame` object, rather than the full image height and width returned by the device.

Note You use the ROI width and height when you create the frame but you use the full image width and height when you copy the image data from the buffer into the frame object.

Example

The following is a version of the `isSendFrame()` loop in the acquisition thread function that checks the ROI. Note that you call the `getROI()` function to get the ROI values, and then use the width and height values in the call to `makeFrame()` and the offsets from the origin in the call to `setImage()`.

```
if (adaptor->isSendFrame()) {

    // Get ROI information.
    int roiOriginX, roiOriginY, roiWidth, roiHeight;
    adaptor->getROI(roiOriginX,
                    roiOriginY,
                      roiWidth,
                     roiHeight);

    // Get frame type & dimensions
    imaqkit::frametypes::FRAMETYPE frameType =
                                   adaptor->getFrameType();
    int imWidth = adaptor->getMaxWidth();
    int imHeight = adaptor->getMaxHeight();

    // Create a frame object
    imaqkit::IAdaptorFrame* frame =
    adaptor->getEngine()->makeFrame(frameType,
                                    roiWidth,   // ROI width
                                    roiHeight); // ROI height

    // Copy data from buffer into frame object
    frame->setImage(imBuffer,
                     imWidth, // Full image width
                    imHeight, // Full image height
                  roiOriginX, // ROI origin
                  roiOriginY); // ROI origin

    // Set image's timestamp
    frame->setTime(imaqkit::getCurrentTime());
    // Send frame object to engine.
    adaptor->getEngine()->receiveFrame(frame);

} // if isSendFrame()
```

Implementing Hardware ROI

To implement hardware ROI, you must overload the IAdaptor's `getROI()` and `setROI()` member functions in your implementation of your adaptor class. By default, if the `IAdaptor` object's `getROI()` member function is not overloaded, ROI configurations will be handled in software by `imaqkit::IEngine`.

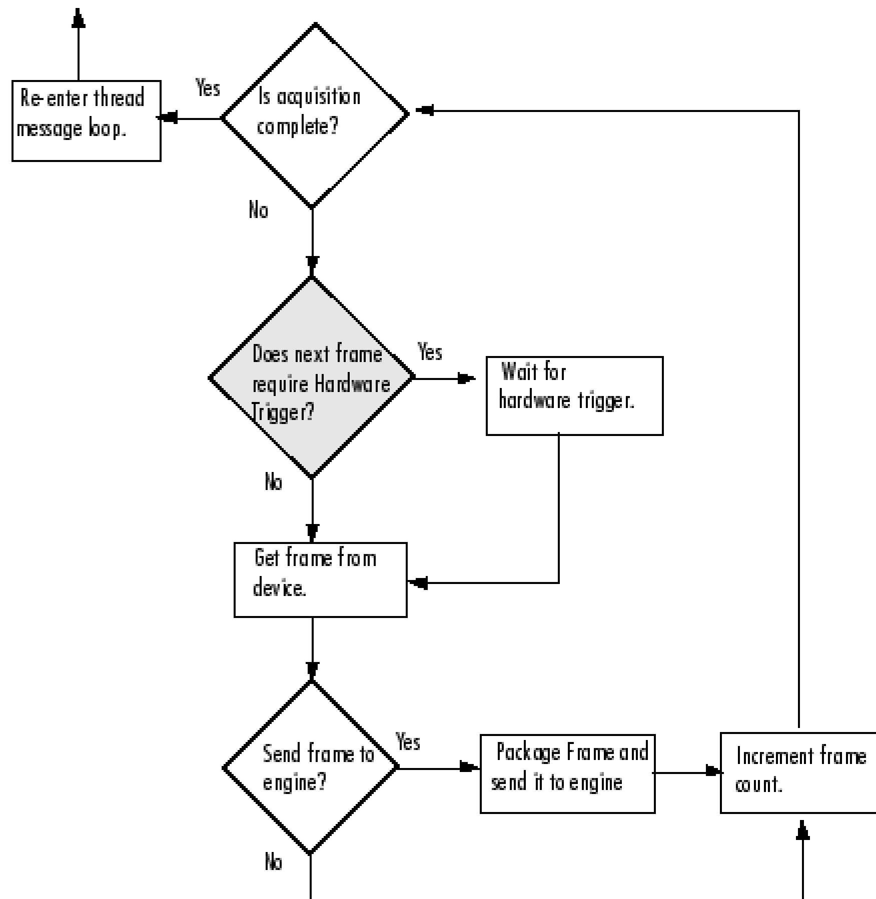
Supporting Hardware Triggers

The toolbox supports three types of triggers:

- Immediate — Trigger fires when video input object is started.
- Manual — Trigger fires when user calls `trigger` function.
- Hardware — Trigger fires when externally defined conditions are met

The engine provides automatic support for immediate and manual triggers. If you want your adaptor to support hardware triggers you must check to see if users have specified a hardware trigger in your acquisition thread function. Before you start acquiring frames from your device, insert a call to the `IAdaptor` member function `useHardwareTrigger()` to determine if the frame acquisition loop should wait for a hardware trigger to fire. If a hardware trigger is configured, insert device SDK calls required to wait for trigger.

The following figure illustrates the frame acquisition loop with the test for hardware trigger.



Main Acquisition Loop with Test for Hardware Trigger

Example

The following is an acquisition thread function that includes a call to check for hardware trigger.

```
while(adaptor->isAcquisitionNotComplete()) {

    // Check for hardware trigger
    if (adaptor->useHardwareTrigger()) {
```

```
                    // Add code here to configure the image
                    // acquisition device for hardware
                    // triggering.
                }

                if (adaptor->isSendFrame()) {

                 // see acquistion thread

                } // if isSendFrame()

            // Increment the frame count.
            adaptor->incrementFrameCount();

            } // while(isAcquisitionNotComplete()

             break;
        } //switch-case WM_USER
   } //while message is not WM_QUIT

return 0;
}
```

Using Critical Sections

This section describes how to use critical sections to protect portions of your adaptor code. The section describes the adaptor kit's main critical section class, `ICriticalSection`, and the ancillary class, `IAutoCriticalSection`, that you use to manage critical sections. Topics covered include

- "Understanding Critical Sections" on page 5-31
- "Example: Using a Critical Section" on page 5-32

Understanding Critical Sections

To prevent sections of code or resources from being accessed simultaneously by multiple threads, use critical section (`ICriticalSecton`) objects. The basic process for using a critical section has three-steps:

1. Create a critical section object, using the adaptor kit `createCriticalSection()` function.
2. At the point in your code that you want to protect, enter the critical section by calling the `ICriticalSection::enter()` member function.
3. At the end of the code that you want to protect, leave the critical section by calling the `ICriticalSection::leave()` member function.

While this process might appear simple, using a `ICriticalSection` object directly in this way can expose your adaptor to problems. For example, if an error occurs in the protected code, the call to the `leave()` function might never be executed. Entering a critical section and then never leaving it can cause unexpected results.

To make working with critical sections easier, the adaptor kit provides a second class, called `IAutoCriticalSection`, that can help you manage the critical sections you define.

You first create an `ICriticalSection` object and then pass this object to the `createAutoCriticalSection()` function when you create the `IAutoCriticalSection` object. When you create the object, you automatically enter the critical section without having to call the `enter()` function. When the protected code goes out of scope, the auto critical section automatically

leaves the critical section without your code having to call the `leave()` function.

The auto critical section object ensures that you always exit a critical section. However, you must also ensure that the auto critical section itself gets deleted. To do this, the adaptor kit recommends managing the handle to the `IAutoCriticalSection` object, returned by `createAutoCriticalSection()`, as an `auto_ptr` using the `std::auto_ptr<>` template class from the Standard Template Library. The `auto_ptr` helps ensure that the `IAutoCriticalSection` handle is deleted.

Example: Using a Critical Section

To define a section of code as a critical section, follow this procedure.

1. Create an `ICriticalSection` object, using the `createCriticalSection()` function. Adaptors typically create an `ICriticalSection` object in their constructors — see “Implementing Your Adaptor Class Constructor” on page 4-13.

   ```
   _mySection = imaqkit::createCriticalSection();
   ```

   The function returns a handle to an `ICriticalSection` object. `_mySection`, which is declared as a member variable in the adaptor class header file, as follows.

   ```
   imaqkit::ICriticalSection* _mySection;
   ```

2. At the point in your code that you want to protect, create an `IAutoCriticalSection` object. The `IAutoCriticalSection` class guarantees that the critical section objects are released when the protected code goes out of scope, or if an exception occurs. In an adaptor, you typically want to protect the frame acquisition loop in a critical section. Insert this code in the acquisition thread function, just before the frame acquisition loop — see “Implementing the Acquisition Thread Function” on page 5-19.

   ```
   std::auto_ptr<imaqkit::IAutoCriticalSection>
   myAutoSection(imaqkit::createAutoCriticalSection(adaptor->_mySection,
   true));
   ```

In this code, the variable myAutoSection is a handle to an IAutoCriticalSection object, that is managed as a Standard Template Library auto_ptr. The code passes a handle to an ICriticalSection object, _mySection, as an argument to the createAutoCriticalSection() function. The second argument to createAutoCriticalSection() specifies that the adaptor should enter the critical section automatically upon creation of the IAutoCriticalSection.

3 At the end of the code that you want to protect, leave the critical section. In an adaptor, you want to leave the critical section after the frame acquisition loop is done. Insert this code just before the acquisition thread function breaks out of the frame acquisition loop — see “Implementing the Acquisition Thread Function” on page 5-19.

You can use the IAutoCriticaSection::leave() function but this is not necessary. The IAutoCriticalSection leaves the critical section automatically when the code section goes out of scope. You might want to include explicit calls to the leave() function in your code to help document the extent of your critical section.

```
bool MyDeviceAdaptor::stopCapture(){

// If the device is not acquiring data, return.
   if (!isAcquiring())
      return true;

// Get the critical section.

   std::auto_ptr<imaqkit::IAutoCriticalSection>
   GrabSection(imaqkit::createAutoCriticalSection(_grabSection,
                                                      true));

   GrabSection->enter();

//***********************************************************
// Insert calls to your device's SDK to stop the device, if
// necessary.
//***********************************************************

// Leave the critical section.
```

```
    GrabSection->leave();

    return true;
}
```

Specifying Device Driver Identification Information

Two of the virtual functions you must implement return identification information about the device driver used to communicate with your device. This information can be useful for debugging purposes.

- `getDriverDescription()` — Returns a text string that identifies the device.
- `getDriverVersion()` — Returns a text string that specifies the version of the device driver.

Adaptors typically use an SDK function to query the device to get this information, if the SDK supports it, or obtain the information from the device documentation.

User Scenario

The identification text strings returned by `getDriverDescription()` and `getDriverVersion()` are visible to users if they call `imaqhwinfo`, specifying a video input object as an argument, as follows.

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

imaqhwinfo(vid)
ans =

               AdaptorName: 'mydeviceimaq'
                DeviceName: 'MyDevice'
                 MaxHeight: 280
                  MaxWidth: 120
              TotalSources: 1
   VendorDriverDescription: 'MyDevice_Driver'
       VendorDriverVersion: '1.0.0'
```

Example

The following example contains skeletal implementations of the `getDriverDescription()` and `getDriverVersion()` functions.

```
const char* MyDeviceAdaptor::getDriverDescription() const{
```

```
 return "MyDevice_Driver";
}

const char* MyDeviceAdaptor::getDriverVersion() const {
 return "1.0.0";
}
```

6

Defining Device-Specific Properties

This chapter describes how to define the properties that toolbox users can use to configure various attributes of a device. These properties can control aspects of the image acquired, such as brightness, behavior of the device, such as shutter speed, and other device-specific characteristics.

Overview (p. 6-2)	Provides an overview of the process of defining device-specific properties
Creating Device Properties (p. 6-6)	Describes how to create device-specific properties
Defining Hardware Trigger Configurations (p. 6-10)	Describes how to create hardware trigger configurations
Setting Up Property Listeners (p. 6-11)	Describes how to set up listeners to detect when users have changed the value of a property

Overview

You define which properties of your image acquisition device you want to expose to toolbox users. You make this determination by reading the device's SDK documentation, determining its capabilities, and deciding which capabilities toolbox users will be expected to configure. Once you decide to expose a property, you must define three characteristics of the property:

- Name
- Data type
- Range of valid values (optional)

Adaptor writers typically wait to define properties until after they are able to acquire data from the device because you need to acquire data to see the effect of some properties.

User Scenario

The properties that you define for your device appear to users as properties of the video source object associated with the video input object. The properties of the video input object, which represent general properties that are common to all image acquisition devices, are defined by the toolbox.

To view the device-specific properties you define, get a handle to the video source object and use the `get` function. To set the value of device-specific properties you define, get a handle to the video source object and use the `set` function. For example, this code creates a video input object, uses the `getselectedsource` function to get a handle to the currently selected video source object, and then views the properties of the video source object.

```
vid = videoinput('winvideo',1)
src = getselectedsource(vid);
get(vid)
General Settings:
    Parent = [1x1 videoinput]
    Selected = on
    SourceName = input1
    Tag =
    Type = videosource
```

```
Device Specific Properties:
  Brightness = -10
  Contrast = 266
  Exposure = 1024
  ExposureMode = auto
  Hue = 0
  Saturation = 340
  Sharpness = 40
```

Suggested Algorithm

When a user calls the `videoinput` function, the engine calls the `getDeviceAttributes()` function to set up any device-specific properties you might have defined for the device. The engine passes several arguments to your adaptor's `getDeviceAttributes()` function:

```
void getDeviceAttributes(const imaqkit::IDeviceInfo* deviceInfo,
                                    const char* acqFormat,
                         imaqkit::IPropFactory* devicePropFact,
                     imaqkit::IVideoSourceInfo* sourceContainer,
                         imaqkit::ITriggerInfo* hwTriggerInfo)
```

The following table describes these arguments.

Argument	Data Type	Description
`deviceInfo`	`IDeviceInfo` object	Specifies the image acquisition device
`acqformat`	Text string	Video format or path to device configuration file
`devicePropFact`	`IPropFactory` object	Provides member functions used to create properties

Argument	Data Type	Description
`sourceContainer`	`IVideoSourceInfo` object	Define the video sources available with this device, described in "Identifying Video Sources" on page 4-10
`hwTriggerInfo`	`ITriggerInfo` object	Specifies hardware triggers. The other two trigger types, immediate and manual, are handled automatically by the toolbox.

The algorithm for `getDeviceAttributes()` typically includes these steps:

1. Determine the device the user wants to establish a connection with. The user specifies the device ID when he creates the video input object.

2. Determine which format the user wants to use with the device. The user specifies the name of the format (or the path of a camera file) when he creates the video input object. To get format information, retrieve the `IDeviceFormat` object associated with the format from the `IDeviceInfo` object.

3. Create a property object appropriate to the data type of the property and store the property object in the device-specific property container — see "Creating Device Properties" on page 6-6.

4. Find all trigger configurations supported by the device and store the information in the `ITriggerInfo` object — see "Supporting Hardware Triggers" on page 5-28.

There are several ways to determine this property, source, and trigger information:

- By querying the device SDK at run-time
- By reading information from an imaging device file (IMDF). If you know the device information in advance, you can store it in an IMDF file using an XML-based markup language. This section describes how to read

information from an IMDF file. To learn how to create an IMDF file, see Chapter 7, “Storing Adaptor Information in an IMDF File”.

- A mixture of both methods.

Creating Device Properties

To define properties for a device, follow this procedure:

1. Create the property using the appropriate IPropFactory member function for the data type. For example, to create a property of type double, use the createDoubleProperty() function, specifying the property name and default value as arguments.

   ```
   hprop = devicePropFact->createDoubleProperty("Brightness",100)
   ```

   The IPropFactory class supports functions to create properties of various data types — see "Selecting the Property Creation Function" on page 6-6. You can also use the createPropFromIMDF() function to create a property from an IMDF file. See "Reading Properties from an IMDF File" on page 6-7 for more information.

2. Specify when the property can be modified, if ever, using the setPropReadOnly() function of the IPropFactory object. Use one of the following constants (defined in IEngine.h): READONLY_ALWAYS, READONLY_NEVER, and READONLY_WHILE_RUNNING. For example,

   ```
   devicePropFact->setPropReadOnly(hProp,
                         imaqkit::imaqengine::READONLY_WHILE_RUNNING);
   ```

3. Add the property to the engine's device-specific property container, using the addProperty() method of the IPropFactory object. For example,

   ```
   devicePropFact->addProperty(hProp);
   ```

   where hProp is a handle to the property you created in step 1.

Selecting the Property Creation Function

The IPropFactory() object supports functions that you can use to create properties of various data types, including:

- int
- double
- string

- Enumerated types

For example, use the `createDoubleProperty()` function to create a property whose value is of type `double`.

```
hprop = devicePropFact->createDoubleProperty("MyDoubleProp",2.5)
```

For the `int` and `double` types, you can also specify properties that have pairs of values or values within a defined range. For example, this code creates an integer property with upper and lower bounds.

```
hprop = devicePropFact->createIntProperty("MyBoundedIntProp",
                                           0,100,50)
```

To create a property with enumerated values, use `createEnumProperty()`, specifying the property name, and one enumeration, for example,

```
hprop = devicePropFact->createEnumProperty("MyEnum",
                                            "green",1)
```

You then add additional properties using `addEnumValue()`.

For more information about the `IPropFactory` class, see the *Image Acquisition Toolbox Adaptor Kit API Reference* documentation.

Reading Properties from an IMDF File

As an alternative to using IPropFactory member functions, you can also create properties by reading property information from an IMDF file using `createPropFromIMDF()` function.

This code fragment from the Demo adaptor creates a property from an IMDF file.

```
devicePropFact->createPropFromIMDF(demo::SHARPNESS_STR)
```

For more information about IMDF files, see Chapter 7, “Storing Adaptor Information in an IMDF File”

Creating Property Help

You can use IMDF files to define help text for the device-specific properties you create. For more information, see "Specifying Help in an IMDF File" on page 7-7.

Example

The following example presents a skeletal implementation of a `getDeviceAttributes()` function. The intent of this example is to show how to use adaptor kit objects to specify video sources and properties of various types.

This code fragment does not read source, property, or trigger information from an IMDF file. For information about this topic, see Chapter 7, "Storing Adaptor Information in an IMDF File"

1 Add the following code to the `getDeviceAttributes()` function in the adaptor. You created a skeletal version of this function in "Identifying Video Sources" on page 4-10. This code creates several properties of various types.

```
   void* hProp;  // Declare a handle to a property object.

// Create a property of type double with a default value
hProp = devicePropFact->createDoubleProperty("MyDoubleProp",2.5);

//  Specify when the property value can be modified.
devicePropFact->setPropReadOnly(hProp,
                              imaqkit::imaqengine::READONLY_ALWAYS);

// Add the property to the device-specific property container.
devicePropFact->addProperty(hProp);

// Create a bounded int property with maximum and minimum values
hProp = devicePropFact->createIntProperty("MyBoundedIntProp",
                                             0, 100, 50);

//  Specify when the property value can be modified.
devicePropFact->setPropReadOnly(hProp,
                               imaqkit::imaqengine::READONLY_NEVER);
```

```
// Add the property to the device-specific property container.
devicePropFact->addProperty(hProp);

// Create an enumerated property
hProp = devicePropFact->createEnumProperty("MyEnumeratedProp",
                                           "green", 1);

// Add additional enumerations
devicePropFact->addEnumValue(hProp, "blue", 2);
devicePropFact->addEnumValue(hProp, "red", 3);

// Specify when the property value can be modified.
  devicePropFact->setPropReadOnly(hProp,
                        imaqkit::imaqengine::READONLY_WHILE_RUNNING);

   // Add the property to the device-specific property container.
devicePropFact->addProperty(hProp);
```

2 Compile and link your adaptor to create the DLL.

3 Start MATLAB.

4 Create a video input object for your adaptor.

```
vid = videoinput('mydevice',1)
```

5 Get the selected source and view the device-specific properties you created.

```
src = getselectedsource(vid);
get(vid)
General Settings:
    Parent = [1x1 videoinput]
    Selected = on
    SourceName = input1
    Tag =
    Type = videosource

  Device Specific Properties:
    MyDoubleProp = 2.5
MyBoundedIntProp = 100
MyEnumeratedProp = green
```

Defining Hardware Trigger Configurations

To define hardware trigger configurations, use the `addConfiguration()` function of the `ITriggerInfo` object. The engine passes a handle to an `ITriggerInfo` object to your adaptor's `getDeviceAttributes()` function.

When you create a hardware trigger configuration, you specify:

- Name of the source of the trigger
- ID of the trigger source
- Name of the condition that triggers an acquisition
- ID of the trigger condition

For example,

```
hwTriggerInfo->addConfiguration("MyTriggerSource", 1,
                                 "MyTriggerCondition",2)
```

Setting Up Property Listeners

This section describes how to setup property listeners so that your adaptor can get notification when the value of a device-specific property changes. Adaptor writers typically set up property listeners in their adaptor class constructor — see "Implementing Your Adaptor Class Constructor" on page 4-13.

User Scenario

When a user changes the value of a video input object or a video source object property, using the `set` function, the engine changes the value of the property in its property containers. The engine maintains two property containers. One container, called the *engine property container*, stores the properties of the video input object. The other container, called the *adaptor property container*, stores the device-specific properties of the associated video source object. The properties that you can define in your adaptor appear as properties of the video source object.

When the value of a device-specific property changes, adaptors typically need to respond. For example, if a user changes the value of the `Brightness` property, your adaptor must communicate with the device to configure the new value.

Receiving Notification of Property Value Changes

To receive notification when the value of a property changes, your adaptor must associate a listener object with a particular property. When the value of that property changes, the engine calls the associated listener object. You define what the listener does in response to this notification, such as changing the actual configuration of the device.

To setup property listeners, follow this procedure. The sections that follow provide more detail about each step.

1. Define a property listener class, deriving it from the IPropPostSetListener abstract class — see "Defining a Listener Class" on page 6-12.

2. Implement the `notify()` virtual function in your listener class — see "Creating the notify() Function" on page 6-13.

3 Associate an object of your listener class with a property — see "Associating a Listener with a Property Container" on page 6-15.

Defining a Listener Class

To receive notification when a property value changes, you must define a property listener class that is derived from the abstract class `IPropPostSetListener`. (The name of the class includes the word `Post` because listeners are notified after the property value stored in the container is updated, i.e., post-set.)

The IPropPostSetListener class defines only one virtual function that you are required to implement: the `notify()` member function. In this function, you define what the listener does when it is notified of a change to a property value.

This example shows the definition of a listener class. The constructor for a listener class must accept a handle to an `IAdaptor` object that is their parent because listeners are established on a per instance basis. For more information about the `notify()` function, see "Creating the notify() Function" on page 6-13.

```
#include "mwadaptorimaq.h"
#include "MyDeviceImaq.h" // For this example

class MyDevicePropListener : public IPropPostSetListener
{
public:

   // Constructor/Destructor
 MyDevicePropListener(MyDeviceAdaptor* parent):
                              _parent(parent) {}

 virtual ~DemoPropListener() {};

   virtual void notify(imaqkit::IPropInfo* propertyInfo,
                       void* newValue);

private:

   // Declare handle to parent as member data
```

```
    MyDeviceAdaptor* _parent;

    // Property Information object.
    imaqkit::IPropInfo* _propInfo;

    // The new value for integer properties.
    int _lastIntValue;

    // The new value for double properties.
    double _lastDoubleValue;

    // The new value for string properties.
    char* _lastStrValue;

};
```

Creating the notify() Function

When a user changes the value of a property, the engine calls the notify() function of the listener class associated with the property.

Your listener class notify() function must accept two parameters:

```
void notify(IPropInfo* propertyInfo, void* newValue)
```

where

- propertyInfo is a handle to an IPropInfo object — The IPropInfo class is the interface that lets you get information about the property being configured. For example, using IPropInfo functions you can get the name of the property, its storage type and its default value.
- newValue is a pointer to the new property value — This value is provided as a void* and must be cast to the appropriate C++ data type. The following table tells which C++ data type to cast to for all of the property types supported by the adaptor kit.

imaqkit::PropertyTypes	C++ Data Type
STRING	char*
DOUBLE	double*

imaqkit::PropertyTypes	Case Data Type
INT	int*
DOUBLE_ARRAY	imaqkit::PropertyTypes::NDoubles*
INT_ARRAY	imaqkit::PropertyTypes::NInts*

Suggested Algorithm

The design of the `notify()` function can vary with the needs of your device and the facilities offered by its SDK. For example, you could create one property listener class that handles all value changes for all properties in a particular property container (general or device-specific). In this case, the `notify()` function would include a switch statement with cases that handle each individual property.

As an alternative, you could also define a separate listener class for each property or each property storage type. The engine would call the specific listener for the property specified.

You can also define listener classes that fit the way the SDK organizes property configuration. For example, if an SDK provides one function to configure all device properties, you can define a property listener class for these properties.

Example

This example shows an implementation of a `notify()` function for integer types.

```
void MyDevicePropListener::notify(IPropInfo* propertyInfo,
                                            void* newValue)
{

   // Get property name from the IPropInfo object.
   const char* propname = propertyInfo->getPropertyName();

   // Cast newValue to the proper type
   newVal = *reinterpret_cast<const int*>(newValue);
```

```
        // Insert calls to device SDK to apply value to hardware.

        // For debug purposes only.
        imaqkit::adaptorWarn("In listener. Property name is %s\n",propname);

    }
```

Associating a Listener with a Property Container

To set up a listener for a property, you associate it with a particular property in a particular property container. The following example shows how to add listeners for all the device-specific properties in the adaptor property container.

1 Get a handle to the appropriate property container object.

   The `IEngine` object has two member functions that return handles to property containers (`IPropContainer` objects). Because this example associates listeners with device-specific properties, it calls the `IEngine` class `getAdaptorPropContainer()` member function.

```
   imaqkit::IPropContainer* adaptorPropContainer =
                            getEngine()->getAdaptorPropContainer();
```

2 Add a listener to a property in the container, using the `IPropContainer` object's `addListener()` function. As arguments, specify the name of the property and a handle to listener object.

Note Because each instance of a listener object is deleted when the video input object is deleted, you must associate a new instance of a listener object with each property.

   The following example iterates through all the properties in the adaptor property container, associating a listener object with each one.

```
   void MyDeviceAdaptor::MyDeviceAdaptor()
   {
       // get a handle to the property container
       IPropContainer* propContainer =
```

```
                              getEngine()->getAdaptorPropContainer();

    // Determine the number of properties in the container.
    int numDeviceProps = propContainer->getNumberProps();

    // Retrieve the names of all the properties in the container
    const char **devicePropNames = new const
                              char*[numDeviceProps];
    propContainer->getPropNames(devicePropNames);

    // Create a variable to point to a property listener object.
    MyDevicePropListener* listener;

    // For each property in the container...
    for (int i = 0; i < numDeviceProps; i++){

        // Create a listener object...
        listener = new DemoPropListener(this);

        // and associate it with a specific property.
        propContainer->addListener(devicePropNames[i], listener);
    }

    // clean up the array of property names.

    delete [] devicePropNames;

}
```

7

Storing Adaptor Information in an IMDF File

This chapter describes how to store information about adaptor properties in an Image Device File (IMDF) in an XML based format.

Overview (p. 7-3)	Provides an overview of the IMDF property information mechanism
Creating an IMDF File: Toplevel Elements (p. 7-5)	Describes the basic elements in an IMDF file t
Specifying Help in an IMDF File (p. 7-7)	Describes how to create help text entries in an IMDF file
Specifying Device Information (p. 7-13)	Describes how to store device information in an IMDF file
Specifying Property Information (p. 7-16)	Describes how to store property information in an IMDF file
Specifying Format Information (p. 7-20)	Describes how to store video format information in an IMDF file
Specifying Hardware Trigger Information (p. 7-22)	Describes how to store hardware trigger information

Specifying Video Sources (p. 7-24)	Describes how to store video source information
Defining and Including Sections (p. 7-25)	Describes how to group IMDF elements together in collections called sections and reference sections from other elements

Overview

This chapter describes how to use an XML-based markup language to specify source, property, and hardware trigger information in an Imaging Device File (IMDF).

Note Creating an IMDF is optional. However, using an IMDF file can simplify the coding of your adaptor's `getDeviceAttributes()` function. In addition, it is the only convenient way to make help text available for the device-specific properties your adaptor creates.

User Scenario

When a user calls the `imaqhwinfo` function, the toolbox searches for adaptor DLLs. When it finds a DLL, it also looks for a matching IMDF file in the same directories. If found, the engine stores path information to the IMDF file. An IMDF file must reside in the same directory as your DLL and the `.imdf` file extension, such as `demoimaq.imdf`.

When a user calls the `videoinput` function to create a video input object, the engine reads and processes the IMDF file. When it reads the file, it processes the property, trigger, and source information specified at the top-level of the file. (To understand the hierarchical arrangement of an IMDF file, see "Elements of the IMDF Markup Language" on page 7-3.)

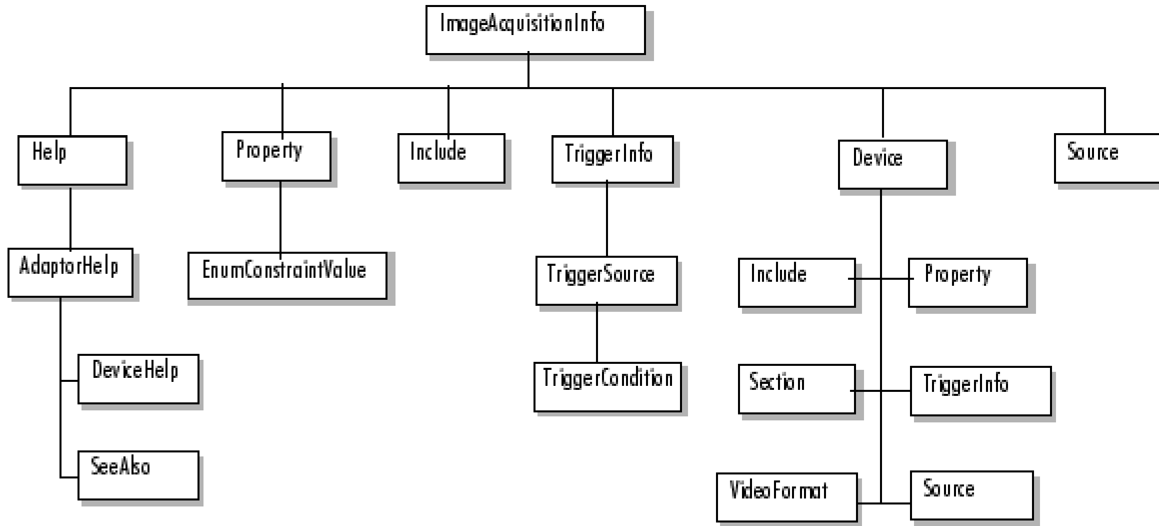
After reading all the top-level elements, the engine looks for a device element that matches the device the user specified in the `videoinput` function. If found, the engine then looks for video format element in the device element.

The engine then calls your adaptor's `getDeviceAttributes()` function, as described in "Setting Up Property Listeners" on page 6-11, to set up device properties.

Elements of the IMDF Markup Language

The following figure shows the hierarchical relationship of the elements of the XML-based markup language for IMDF files. The figure shows which elements can be children of other elements. Elements at the top-level of an

IMDF file (elements that are children of the `ImageAcquisitionInfo` element apply to an entire adaptor. Elements that are children of a `Device` element apply only to that device. To simplify the diagram, all possible subordinate elements are not always shown for elements below the top-level. When used in an IMDF file, elements are called *nodes*.



Hierarchy of IMDF Elements

Creating an IMDF File: Toplevel Elements

The `ImageAcquisitionInfo` element must be the root node of all IMDF files. Your IMDF file must begin with the `ImageAcquisitionInfo` node and end with the corresponding `ImageAcquisitionInfo` terminator, as in this example.

```
<ImageAcquisitionInfo>
.
.
.
</ImageAcquisitionInfo>
```

The following table lists the toplevel IMDF elements that you can specify as children of the `ImageAcquisitionInfo` element. The table lists the elements in the order they must appear in an IMDF file. Note that element tag names are case-sensitive.

Element	Description
`<Help>`	Defines the section of an IMDF file in which you specify help text for device-specific properties — see "Specifying Help in an IMDF File" on page 7-7. This is an optional element. If specified, an IMDF file can contain only one `Help` element.
`<Property>`	Defines properties of a device — see "Specifying Property Information" on page 7-16. This is an optional element. An IMDF file can contain multiple `Property` elements.
`<Include>`	Convenient way to specify another element or group of elements as children of another element — see "Defining and Including Sections" on page 7-25. This is an optional element. An IMDF file can contain multiple `Include` elements.
`<Source>`	Defines the source of video data — see "Specifying Video Sources" on page 7-24. This is an optional element. An IMDF file can contain multiple `Source` elements.

Element	Description
`<TriggerInfo>`	Defines hardware trigger information — see “Specifying Hardware Trigger Information” on page 7-22. This is an optional element. An IMDF file can contain only one `TriggerInfo` element.
`<Device>`	Specifies information about a device — see “Specifying Device Information” on page 7-13. This is an optional element. An IMDF file can contain multiple Device elements.

Specifying Help in an IMDF File

To define help text for adaptor properties in an IMDF file, use the `Help` element. You can include only one `Help` node in an IMDF file and it must be a child of the root node.

As children of the `Help` node, you create `AdaptorHelp` nodes that contain the help text for a particular property. You use the `name` attribute of the `AdaptorHelp` element to specify which property the help is associated with.

You can optionally include device-specific content in the help text. This text appears only when a particular device is selected. Use one or more `DeviceHelp` nodes to add device-specific help to an `AdaptorHelp` node. You use the `device` attribute of the `DeviceHelp` element to specify when the text should appear. You can also create see also references for your property help using `SeeAlso` nodes.

The following example outlines how to use these elements to create property help. The words in italics represent text that you must define.

```
<ImageAcquisitionInfo>
 <Help>
  <AdaptorHelp property=propertyname>
   Help text
   <DeviceHelp device=devicename>
    Device-specific help text
   </DeviceHelp>
   <SeeAlso>
    References to other properties
   </SeeAlso>
  </AdaptorHelp>
 .
 .
 .
 </Help>
</ImageAcquisitionInfo>
```

The following table summarizes the tags you can use to define help. For more information, see these topics.

- "User Scenario: Viewing Property Help" on page 7-8
- "Creating AdaptorHelp Nodes" on page 7-10

Element	Description	Attributes
<Help>	Defines the help section in an IMDF file. Must be child of the `ImageAcquisitionInfo` node.	None
<AdaptorHelp>	Defines the online help for a property. The Help node can contain one or more AdaptorHelp nodes.	`property=`*name*, where *name* is a text string specifying the property name
<DeviceHelp>	Specifies device-specific text in property help. This element is optional. An `AdaptorHelp` node can contain multiple `DeviceHelp` nodes.	`device=`*name*, where *name* is a text string that identifies a particular device
<SeeAlso>	Defines the see also line in property help. This element is optional. An `AdaptorHelp` node can contain multiple `SeeAlso` nodes.	None

User Scenario: Viewing Property Help

The purpose of using a `Help` element in an IMDF file is to create help text for device-specific properties. A user of your adaptor can display the help text at the command line using the `imaqhelp` command.

The following example shows how a user displays the help text for a device-specific property using the `imaqhelp` command. To see how to create this help in an IMDF file, see "Creating AdaptorHelp Nodes" on page 7-10.

```
    vid = videoinput('winvideo',1);
(1) src = getselectedsource(vid);
    get(src)
(2)   General Settings:
        Parent = [1x1 videoinput]
        Selected = on
        SourceName = input1
        Tag =
        Type = videosource

      Device Specific Properties:
        Brightness = -10
        Contrast = 266
        Exposure = 1024
        ExposureMode = auto
        Hue = 0
        Saturation = 340
        Sharpness = 40

(3) imaqhelp(src,'brightness')
(4) BRIGHTNESS  [-128  128]  (Read-only: whileRunning)

        Specify the brightness, also called the black level.
(5)
        Brightness describes the difference in intensity of light reflected from or
        transmitted through an image independent of its hue and saturation.

        For some devices, the value is expressed in IRE units * 100. For other
        devices, the units are arbitrary. Refer to the device's documentation for
        information.

        Depending on the acquisition device, this property may have an associated
        mode property allowing this value to be controlled automatically by the device,
        or for it to be manually configured.

(6)     See also BrightnessMode.
```

The items in this list correspond to the numbered elements above.

1 Device-specific properties are properties of the video source object. The example creates the video input object and then uses the `getselectedsource` function to get a handle to the video source object.

2 The example uses the `get` function to display a list of device-specific properties.

3 Use the `imaqhelp` function to display help for one of the properties of the video source object.

4 The first line of the help lists the name of the property with its constraints, such as range and permission.

5 The text of the help appears exactly as you enter it in the IMDF file. You include the text after the `AdaptorHelp` tag.

6 The **See Also** line is created by the `SeeAlso` node.

Creating AdaptorHelp Nodes

This section describes how to create help text for a property using the set of help tags defined by the IMDF DTD. The following example shows the IMDF entry for the `Brightness` property, displayed in "User Scenario: Viewing Property Help" on page 7-8. The example sets the `property` attribute of the `AdaptorHelp` tag to the name of a property.

Note Help text must start with a one-line summary. Make sure that each line of text is no longer than 80 characters.

```
<AdaptorHelp property="Brightness">
Specify the brightness, also called the black level.

Brightness describes the difference in intensity of light reflected from
or transmitted through an image independent of its hue and saturation.
For some devices, the value is expressed in IRE units * 100. For other
devices, the units are arbitrary. Refer to the device's documentation for
information.

Depending on the acquisition device, this property may have an associated
```

```
mode property allowing this value to be controlled automatically by the
device, or for it to be manually configured.

<SeeAlso>BrightnessMode.</SeeAlso>

</AdaptorHelp>
```

Including Device-Specific Help Text

To include help text that only appears for specific devices, use `DeviceHelp` elements.

In this example, the help text contains three device-specific sections. Note how the example sets the `device` attribute of the `DeviceHelp` property to the name of a device.

```
<AdaptorHelp property="StrobeEnable">
Enables the strobe output and its timer.

Upon enabling the strobe output, it will start detection
of triggers and generate output as appropriate.  Consult your
hardware manual for a detailed description of the strobe output.

<DeviceHelp device="PC2Vision">See also StrobeMode,
StrobeDuration, StrobeDelay, StrobePolarity,
StrobeAlignOnHs.
</DeviceHelp>

<DeviceHelp device="PCVisionPlus">See also StrobeMode,
StrobeDelay, StrobePolarity.
</DeviceHelp>

<DeviceHelp device="PCRGB">For the PC-RGB, StrobeEnable only
enables the timing circuitry.  The strobe output must still be
enabled with the StrobeOutputEnable property.

See also StrobeMode, StrobePolarity, StrobeDelay,
StrobeOutputEnable.
</DeviceHelp>
```

```
</AdaptorHelp>
```

Specifying Device Information

To specify information about a particular device in an IMDF file, use the `Device` element. You can include as many `Device` nodes in an IMDF file as you want but they must all be children of the root node.

In a `Device` node, you specify the name of the device as an attribute. The name is typically a text string defined by the device's SDK. Using other IMDF elements as children of the `Device` node, you can specify information about device-specific properties, video formats, and trigger information.

The following example outlines how to use these elements to create `Device` nodes. The words in italics represent text you define.

```
<ImageAcquisitionInfo>
 <Device device=devicename>
  <VideoFormat name=formatname>
  </VideoFormat>

  <Property constraint=constraint_value
     deviceSpecific=true_or_false
     name=property_name
     readOnly=always_never_or_whileRunning
     type=cell_double_int_or_string
     min=minimum_value
     max=maximum_value
     optional=on_or_off
     default=default_value>
  </Property>

  <TriggerInfo>
   <TriggerSource id=ID name=string>
    <TriggerCondition id=ID name=string/>
   </TriggerSource>
  </TriggerInfo
 </Device>
 .
 .
 .
</ImageAcquisitionInfo>
```

The following table summarizes the elements that can be children of a Device node, in the order they must be specified. For an example, see "Example: Device Node" on page 7-14.

Element	Description	Attributes
`<VideoFormat>`	Specifies information about a video format. This is an optional element. A `Device` node can contain multiple `VideoFormat` nodes.	`name=`*formatname*, where *formatname* is a text string that identifies a particular device
`<Include>`	Include a `Section` node in another node. This is an optional element. A `Device` node can contain multiple `Include` nodes.	`tag=`*sectionname*, where *sectionname* is a text string that identifies a particular `Section` node
`<Section>`	Groups a set of nodes into a `Section` node. This is an optional element. A `Device` node can contain multiple `Section` nodes.	`name=`*sectionname*, where *sectionname* is the name you want to assign to the group of nodes
`<Property>`	Describes the properties of a device. This is an optional element. A `Device` node can contain multiple `Property` nodes.	See "Specifying Property Information" on page 7-16.
`<Source>`	Defines the source of video data. This is an optional element.	See "Specifying Video Sources" on page 7-24
`<TriggerInfo>`	Provides information about hardware triggers, such as source and condition. This is an optional element. Note: A `Device` node can contain only one `TriggerInfo` node.	See "Specifying Hardware Trigger Information" on page 7-22.

Example: Device Node

The following example creates a `Device` node containing property and trigger information. For more information about the `Property` element, see

“Specifying Property Information” on page 7-16. For more information about the `TriggerInfo` element, see “Specifying Hardware Trigger Information” on page 7-22.

```
<Device name="PCVision">
   <Property optional="on"
             constraint="enum"
             deviceSpecific="true"
             name="SyncSource"
             readOnly="whileRunning"
             type="string">
      <EnumConstraintValue id="1" name="strippedSync" />
      <EnumConstraintValue id="2" name="separateSync" />
      <EnumConstraintValue id="3" name="compositeSync" />
      <EnumConstraintValue id="4" name="variableScan" />
   </Property>

   <Property optional="on"
             constraint="enum"
             deviceSpecific="true"
             name="FieldStart"
             readOnly="whileRunning"
             type="string">
      <EnumConstraintValue id="0" name="evenField" />
      <EnumConstraintValue id="1" name="oddField" />
   </Property>

   <TriggerInfo>
      <TriggerSource id="1" name="extTrig">
         <TriggerCondition id="0" name="risingEdge"/>
         <TriggerCondition id="1" name="fallingEdge"/>
      </TriggerSource>
   </TriggerInfo>
</Device>
```

Specifying Property Information

To specify property information in an IMDF file, use the Property element. You can include as many Property nodes in an IMDF file as you want. Property nodes can be children of the root node, a Device node, or a Videoformat node. Property nodes can also be children of Section nodes.

Note Property nodes that are children of the root node affect all devices accessed through the adaptor. Property nodes that are children of a Device or VideoFormat node affect only that device or video format.

You use attributes of the Property element to specify characteristics of the property, such as its name, type, and constraints. For more information about Property attributes, see "Specifying Property Element Attributes" on page 7-17.

The following example outlines how to use these elements to specify property information. The example shows the Property node as a child of the root node but you use it the same way as a child of a Device or VideoFormat node. The words in italics represent text you define.

```
<ImageAcquisitionInfo>
 <Property constraint=constraint_value
    deviceSpecific=true_or_false
    name=property_name
    readOnly=always_never_or_whileRunning
    type=cell_double_int_or_string
    min=minimum_value
    max=maximum_value
    optional=on_or_off
    default=default_value>
 </Property>
 .
 .
 .
</ImageAcquisitionInfo>
```

Specifying Property Element Attributes

The following table lists the attributes of a Property node in alphabetical order. The table gives a brief description of the property and lists which properties are required and which are optional.

Attribute	Description	Required
`constraint`	Specifies the constraints on the property — see "Specifying Values for the Constraint Attribute" on page 7-17.	Required
`default`	Default value for the property.	Optional
`deviceSpecific`	Boolean value. `True` if property is vendor-specific; otherwise `false`.	Required
`min`	Minimum allowable value	Optional
`max`	Maximum allowable value	Optional
`name`	Name of property	Required
`optional`	If set to `off`, the property is created automatically and added to the object when the IMDF file is processed. If `on`, the adaptor must explicitly create the property. The default is `off`.	Optional
`readOnly`	Read-only status of property: `always`, `never`, or `whileRunning`.	Required
`type`	Data type of the property: `cell`, `double`, `int` or `string`.	Required

Specifying Values for the Constraint Attribute

Constraints specify information about what are valid values for a property. For example, to specify that a property only accepts positive values, use the positive constraint value, as follows:

```
constraint=positive
```

The following table lists all the possible values for the `constraint` attribute in alphabetical order.

Constraint Value	Description
`bounded`	Property has both a minimum and maximum value. If you set the `constraint` attribute to `bounded`, you must assign values to the `min` and `max` attributes.
`enum`	Property is an enumerated value. If set, the `Property` node must contain one or more `EnumConstraintValue` nodes. See “Specifying Enumerated Values” on page 7-18.
`inforpositive`	Value must be positive or infinite
`none`	No constraints
`positive`	Value must be positive
`zeroinforpositive`	Value must be greater than zero or infinite
`zeroorpositive`	Value must be greater than zero

Specifying Enumerated Values

If your property uses enumerated values, you must set the value of the `constraint` attribute to `enum`, the type attribute to `string`, and create `EnumConstraintValue` elements for each enumeration. The `EnumConstraintValue` nodes are children of the `Property` node.

When you create the EnumConstraintValue nodes, you specify two attributes:

- Value ID
- Value name

This example defines the property `StrobeEnable`. The constraint attribute is set to `enum`. The name attribute of the `EnumConstraintValue` nodes defines the possible values of this enumeration: `on` and `off`.

```
<Property optional="on"
          constraint="enum"
```

```
          deviceSpecific="true"
          name="StrobeEnable"
          readOnly="whileRunning"
          type="string">
   <EnumConstraintValue id="0" name="off" />
   <EnumConstraintValue id="1" name="on" />
</Property>
```

Specifying Format Information

To specify the video formats supported by a particular device in an IMDF file, use the `VideoFormat` element. `VideoFormat` nodes must be children of `Device` nodes. In the `VideoFormat` node, you specify the name of the format as the value of an attribute of the element.

You can also specify format-specific property and trigger information, if necessary. A `VideoFormat` node can have `Property` and `TriggerInfo` nodes as children. (`VideoFormat` nodes can also have a `Section` node as a child — see "Defining and Including Sections" on page 7-25.)

The following example outlines how to use the `VideoFormat` node. The words in italics represent text that you define.

```
<ImageAcquisitionInfo>
 <Device device=devicename>
  <VideoFormat name=formatname>
   <Property constraint=constraint_value
      deviceSpecific=true_or_false
      name=property_name
      readOnly=always_never_or_whileRunning
      type=cell_double_int_or_string
      min=minimum_value
      max=maximum_value
      optional=on_or_off
      default=default_value>
   </Property>

   <TriggerInfo>
   </TriggerInfo>
  </VideoFormat>
 </Device>
 .
 .
 .
</ImageAcquisitionInfo>
```

The following table lists the tags used to specify video format information.

Element	Description	Attributes
`<Include>`	Include one or more nodes grouped into a `Section` node. This is an optional element. A `VideoFormat` node can contain multiple `Include` nodes.	`tag=`*sectionname*, where *sectionname* is a text string that identifies a particular `Section` node
`<Section>`	Groups one or more nodes into a `Section` node. This is an optional element. A `VideoFormat` node can contain multiple `Section` nodes.	`name=`*sectionname*, where *sectionname* is the name you want to assign to a particular `Section` node
`<Property>`	Describes the properties of a video format. This is an optional element. A `VideoFormat` node can contain multiple `Property` nodes.	See "Specifying Property Information" on page 7-16.
`<Source>`	Defines the source of video data. This is an optional element.	See "Specifying Video Sources" on page 7-24
`<TriggerInfo>`	Trigger information specific to a particular video format. This is an optional element. A `VideoFormat` node can only contain one `TriggerInfo` node.	See "Specifying Hardware Trigger Information" on page 7-22.

Specifying Hardware Trigger Information

To specify hardware trigger information in an IMDF file, use the `TriggerInfo` node. A `TriggerInfo` node can be the child of the `ImageAcquisitionInfo`, `Device`, `VideoFormat`, and `Section` nodes.

You specify the source of the hardware trigger in a `TriggerSource` node that is the child of the `TriggerInfo` node. You specify the conditions under which trigger fires in one or more `TriggerCondition` nodes, which are children of the `TriggerSource` node.

The following example outlines how to use these elements to specify trigger information. The words in italics represent text you define.

```
<ImageAcquisitionInfo>
 <Device device=devicename>

  <TriggerInfo>
   <TriggerSource id=ID name=triggername>
    <TriggerCondition id=ID name=conditionname>
  </TriggerInfo>

 </Device>
 .
 .
 .
</ImageAcquisitionInfo>
```

The following table lists the elements used to specify hardware trigger information.

Element	Description	Attributes
`<TriggerInfo>`	Defines information about a hardware trigger.	None

Element	Description	Attributes
<TriggerSource>	Defines the source of the hardware trigger. A Triggerinfo node must contain or more TriggerSource nodes.	See "Specifying Trigger Sources" on page 7-23.
<TriggerCondition>	Defines a condition that must be met before a hardware trigger fires. A TriggerSource node can contain zero or more TriggerCondition nodes.	See "Specifying Trigger Conditions" on page 7-23.

Specifying Trigger Sources

When you define a hardware trigger, you must define the source (or sources) of the hardware trigger in one or more TriggerSource nodes. In a TriggerSource node, you specify values for two attributes: name and id. The value of the name attribute is visible to users of the toolbox in the display returned by the toolbox triggerinfo function. It is typically set to some value that is recognized by the device's SDK.

```
<TriggerSource id="1" name="extTrig">
</TriggerSource>
```

Specifying Trigger Conditions

When you define a hardware trigger, you must define the conditions that must be met before the trigger fires. The parent TriggerSource node specifies the trigger. In a TriggerCondition node, you specify values for two attributes: name and id. The value of the name attribute is visible to users of the toolbox in the display returned by the toolbox triggerinfo function. It is typically set to some value that is recognized by the device's SDK.

```
<TriggerCondition id="1" name="risingEdge">
</TriggerCondition>
```

Specifying Video Sources

To specify the video source in an IMDF file, use the `Source` element. A `Source` node can only be the child of the IMDF root element and it cannot have any child nodes of its own.

When you create a `Source` node, you must specify values for two attributes: `name` and `id`. In the `name` attribute, you specify the name of the source as it appears in the video source object's `Name` property. The `id` is typically set to some value that is recognized by the vendor's SDK. The `id` is only used by the adaptor and needs only to be unique between sources.

The following example outlines how to create a `Source` node. The words in italics represent text you define.

```
<ImageAcquisitionInfo>

 <Source id=ID name=sourcename>
 </Source>
 .
 .
 .
</ImageAcquisitionInfo>
```

Defining and Including Sections

You can gather one or more `Property` or `TriggernInfo` nodes into a group by using the `Section` element. A `Section` node can contain one or more `Property` nodes or a single `TriggerInfo` node or another `Section` node. A `Section` node can be the child of a `Device`, or `VideoFormat` node. Using the `Include` element, a `Section` node can be indirectly be a child of the root node, `Device`, `VideoFormat`, `Section`, or `TriggerInfo nodes`.

`Section` nodes can simplify an XML file. You can reuse node definitions without repeating the XML statements. For example, you can define common elements, such as video formats, that can be shared by several `Device` nodes in the XML file.

The following example outlines how to create a `Section` node and use it in an IMDF file. The words in italics represent text you define.

```
<ImageAcquisitionInfo>
 <Device device=devicename1>
  <Section name=sectionname>
   <Property>
   </Property>

   <TriggerInfo>
   </TriggerInfo
  </Section>
  <Property>
  </Property>
 </Device>
 <Device device=devicename2>
  <Include tag=sectionname/>
 </Device>
 .
 .
 .
</ImageAcquisitionInfo>
```

Index

A

acquiring frames
  determining when it is done 5-20
acquisition thread
  algorithm 5-19
adaptor data
  storing device-specific information 3-18
adaptor property container 6-11
`AdaptorHelp` elements
  creating 7-10
adaptors
  defining device and format information 3-9
  defining device attributes 6-2
  registering 1-11
  unloading DLL 3-20

C

`closeDevice()` virtual function
  implementing 5-10
closing connection to device 5-10
constraint attribute
  specifying 7-17

D

data types
  frame types 5-7
  property types mapped to C++ data types 6-13
device drivers
  specifying name and version 5-35
`Device` element
  example 7-14
  overview 7-13
device information
  classes 3-9
  specifying 3-12
  storing in IMDF file 7-13
devices
  storing device-specific information 3-18

E

engine property container 6-11
enumerated values
  specifying in IMDF file 7-18
error messages
  displaying 3-21
exporting adaptor functions 2-18 3-6

F

formats
  specifying video format in IMDF file 7-20
frame count
  incrementing in frame acquisition loop 5-21
frame types
  packed and planar 5-9

G

`getAvailHW()` function
  implementing 3-9
`getDeviceAttributes()` function
  implementing 6-2

H

hardware triggers
  determining if configured 5-28
  specifying in IMDF file 7-22
`Help` element
  overview 7-7
help text
  for properties 7-10
  format guidelines 7-10

I

`IDeviceFormat` class
  creating 3-13
  storing device format information 3-9
`IDeviceInfo` class
  storing device information 3-9
`IDeviceInfo` objects
  creating 3-12
`IHardwareInfo` class
  storing device and format information 3-9
image frames
  determining when to send to engine 5-21
  frame types 5-7
  sending to engine 5-21
image resolution
  specifying in adaptor class 5-5
`ImageAcquisitionInfo` element
  overview 7-5
Imaging Device Files, *see* IMDF files
`imaqregister`
  using 1-11
IMDF files
  defining sections 7-25
  including sections 7-25
  toplevel elements 7-5
`IPropPostSetListener` class
  deriving from 6-12

L

listeners
  See property listeners 6-11

M

module definition files
  exporting adaptor functions 2-18 3-6
  specifying in Visual Studio 2-19

N

`notify()`
  design considerations 6-14
  implementing the listener class notify function 6-13

P

packed frame types 5-9
planar frame types 5-9
preferences
  registering an adaptor 1-11
properties
  specifying attributes in IMDF file 7-17
  specifying constraints in IMDF file 7-17
  storing in IMDF file 7-16
property containers
  two types 6-11
property data types
  mapped to C++ data types 6-13
`Property` element
  attributes 7-17
  overview 7-16
  specifying constraints in IMDF file 7-17
property help text
  creating 7-10
property listeners
  defining listener class 6-12
  setting up 4-13
  setting up listeners 6-11

R

registering adaptors
  using `imaqregister` 1-11

S

`Source` element
  overview 7-24

`startCapture()` virtual function
    implementing 5-10 5-15
starting the acquisition of frames 5-10 5-15
`stopCapture()` virtual function
    implementing 5-15
stopping the acquisition of frames 5-15

T

thread message loop
    algorithm 5-19
timestamp
    logging with image data 5-21
trigger conditions
    specifying in IMDF file 7-23
trigger sources
    specifying in IMDF file 7-23
`TriggerInfo` element
    overview 7-22

U

`uninitializeAdaptor()` function
    implementing 3-20

V

video formats
    specifying in IMDF file 7-20
video sources
    specifying in IMDF file 7-24
`VideoFormat` element
    overview 7-20

W

warning messages
    displaying 3-21