MICROSCAN

Visionscape® .NET Programmer's Manual

v7.0.0, April 2014

84-100023-02 Rev C

Microscan Systems, Inc.
Tel: +1.425.226.5700 / 800.762.1149
Fax: +1.425.226.8250

Disclaimer

The information and specifications described in this manual are subject to change without notice.

Latest Manual Version

For the latest version of this manual, see the Download Center on our web site at:
www.microscan.com.

Technical Support

For technical support, e-mail: **helpdesk@microscan.com**.

Warranty

For current warranty information, see: **www.microscan.com/warranty**.

Microscan Systems, Inc.

United States Corporate Headquarters
+1.425.226.5700 / 800.762.1149

United States Northeast Technology Center
+1.603.598.8400 / 800.468.9503

European Headquarters
+31.172.423360

Asia Pacific Headquarters
+65.6846.1214

Contents

PREFACE

Welcome

Purpose of This Manual

This manual contains detailed information about how to develop complete custom Visionscape applications using Visual Studio 2008, Visual Studio 2010 (SP1), or Visual Studio 2012 and the .NET framework.

Manual Conventions

The following typographical conventions are used throughout this manual.

- Items emphasizing important information are **bolded**.
- Menu selections, menu items and entries in screen images are indicated as: Run (triggered), Modify..., etc.

Preface

CHAPTER 1 Introduction

Visionscape Architecture

The Visionscape architecture is open, allowing multi-level access to a wide variety of users ranging from factory-floor operators who monitor vision operation, to engineers who set up, install, and/or modify vision applications, to system integrators and low-level software developers who develop custom vision applications.

After you install Visionscape, you have a library of components that allow programming access to the AVP files you've created in FrontRunner as well as to Visionscape hardware components. With these powerful components, you can develop complete custom applications using Visual Studio 2008, Visual Studio 2010 (SP1), or Visual Studio 2012 and the .NET framework.

As a Visionscape programmer, you may customize access to underlying components and provide specific end-user access, such as training and running pre-configured jobs. Programmers can also utilize the components to access specific features, such as live video, job creation, training, and inspection execution.

FIGURE 1–1. Layered Architecture

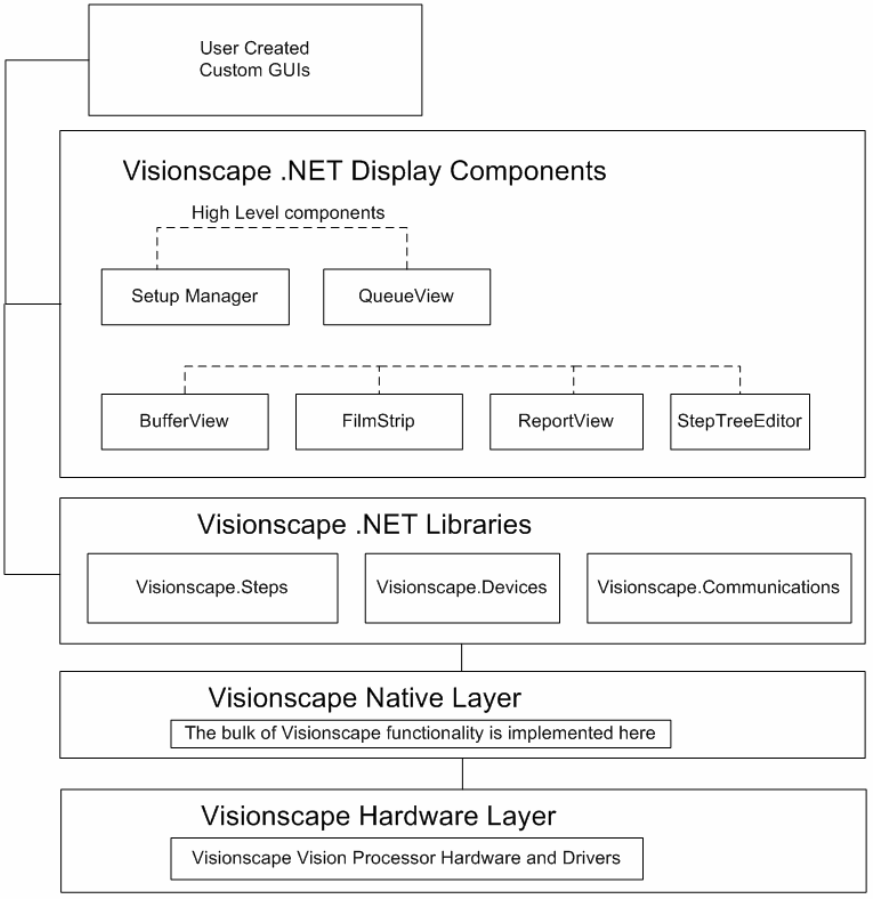


Figure 1–1 shows a basic diagram of the Visionscape component hierarchy. At the topmost level (User Created Custom GUIs) are the end-user applications running on a host PC using Windows®. These applications are written in either C# or Visual Basic.NET.

The next level down shows the Visionscape Display Components layer. These high level visual components can be dropped onto a Windows Form to provide high level functionality such as displaying images at runtime, providing setup capabilities for your vision job, or viewing uploaded vision results and report queue data.

Table 1–1 lists the DLLs that provide display capabilities and the components contained within each.

TABLE 1–1. .NET Display Components

Library	Component	Function	Reference
Visionscape.Display.Image.DLL		Provides components designed to display images.	Chapter 6
	BufferView	Displays a single image buffer.	
	FilmStrip	Displays multiple images in a film-strip-like fashion.	
Visionscape.Display.Devices.DLL		Provides components to make device discovery, display and selection easier.	Chapter 7
	DeviceDropdown	A dropdown combo box that provides a list of all available devices.	
	ToolStripDevice Dropdown	Identical to DeviceDropdown, only this version is for use in ToolStrip controls.	
Visionscape.Display.Reporting		Provides components that can be used to display uploaded inspection data as well as to display and manipulate IO.	Chapter 8
	StatsView	Displays inspection stats (cycle time, process time, etc.)	
	ResultsView	Displays a list of uploaded inspection result data in a grid format.	
	ReportView	Combines the StatsView and ResultsView controls and also displays the current inspection counts in a single control.	

TABLE 1–1. .NET Display Components

	IoView	Displays a list of IO buttons that correspond to a user defined range of IO points. These buttons will display the current state of each IO point, and can be clicked to toggle their state.	
	IoTriggerView	Provides the ability to generate virtual IO trigger pulses.	
Visionscape.Display.Runtime		Provides high-level controls that would typically be used at runtime.	Chapter 9
	QueueView	Allows you to easily display the contents of an Inspection's Part Queue.	
Visionscape.Display.Setup		Provides high-level controls that would typically be used to provide setup functionality.	Chapter 10
	SetupManager	Can be connected to the currently loaded job, and allows the user to acquire images, live video, adjust tool positions and parameters. Also allows the inspection tools to be run, tested and debugged.	
	StepTreeView	Provides a tree view of the currently loaded Job.	

The next level is the Visionscape .NET Libraries layer. These are software libraries that provide access to the core vision system functionality required to develop and deploy vision applications. They have no user interface associated with them. The following table lists the available .NET software libraries:

TABLE 1–2. .NET Software Libraries

Library
Visionscape.DLL
Visionscape.Steps.DLL
Visionscape.Communications.DLL
Visionscape.Devices.DLL

The .NET Libraries talk directly to the next layer down, which is the Visionscape Native layer. This layer includes the actual tools, such as image acquisition, image pre-processing, feature extraction, measurement computation, expression evaluation, control, and I/O. These tools can run either directly on AVP hardware (Vision HAWKs) or on the host PC with Visionscape GigE cameras. There is no need to programmatically interact directly with this layer, as the .NET library layer provides all the functionality you will need.

Finally, at the lowest level is the Hardware Driver layer. No direct programming access is required (or allowed) at this level. Our higher level libraries deal with the hardware for you, insulating you from the complexities of low level device access.

Visionscape Devices

Historically we have used the name “Visionscape Device” to refer to the piece of hardware that you purchased from us. In the past, this was either one of our frame grabber boards, or a smart camera. Starting with Visionscape 4.0, we also support GigE cameras. The Visionscape framework will create a special GigE “Device” to collect and provide access to all of the GigE cameras discovered on your network. This GigE system is a virtual device however, it does not represent a single piece of hardware, but instead represents a collection hardware (your GigE cameras). So consider a “Device” to be a Vision System, an object that can acquire and inspect images, sometimes from more than one camera.

Visionscape Devices fall into these two categories:

- GigE Devices – As mentioned above, a GigE Device is a virtual device that collects all discovered GigE cameras into a single object. The Visionscape framework and your user interface code will interact with the GigE cameras through this single Device.
- Smart Cameras — These devices are cameras with the Vision Processing smarts built right in. A network connection is made to the smart camera and this is used to download your AVP, as well as to upload images and results at runtime. Jobs will run on the smart camera independent of the PC. In fact, once you've downloaded a job to a smart camera and started it running, you can disconnect your PC and the smart camera will continue to run.

Programming Language Considerations

Visual Studio

The Visionscape .NET libraries and all samples described in this manual were created using Visual Studio. The .NET libraries are built using the .NET 3.5 framework.

The Big Picture

In the following chapters we will cover in depth all of the components that make up VsKit.NET.

In this section, we want to try and give you a very general overview of the key concepts involved with creating a typical user interface. This will hopefully give you the "Big Picture" before you continue on through the manual.

Jobs:

A Job is another name for your Vision program. Jobs are always saved to disk with the AVP extension. So we will often refer to Job and AVP interchangeably. A Job is made up of a series of "Steps", which provide vision and logic functionality. Jobs and Steps are covered in depth in Chapter 2.

Devices:

As mentioned above, the term “Device” is used to represent a piece of Visionscape hardware, or in the case of GigE, a collection of GigE cameras. Devices are represented by the VsDevice object, which is covered in depth in Chapter 3.

A Job Must be Connected to a Device Before it Can Run:

A typical user interface would do the following when it starts up:

1. Load a Job from Disk.

2. Get a reference to the Visionscape Device that you want to run on. You will need to wait for the Device to be discovered.

3. Connect the Job to the Device by either downloading to it, or by calling SelectSystem.

4. Start the inspections.

5. Connect ReportConnection object in order to receive images and results while running.

You Control your Running Inspections Through the Device:

Once you have connected your Job to your Device, you can start and stop the inspections through calls to the VsDevice object.

You Receive Images and Data from Running Inspections via Report Connections:

Once you have a Job running on a Device, you will typically want to watch the images and results from the running inspections. To do this, you must create a ReportConnection object, which can receive either images, results or both. An event will be sent to your application whenever a new report is available. Your event handler will receive an InspectionReport object which holds all cycle report data. ReportConnections are covered in depth in chapter 4.

You Display Images using the BufferView control:

As described above, your ReportConnection will send you an InspectionReport object that contains your images and results. The InspectonReport object has an Images collection, which is a collection of BufferDm object. Images are always represented by the BufferDm object. The BufferView control displays BufferDms, so it is very easy to take an image from an InspectionReport, and display it in a BufferView control. For a complete description of the BufferView object, refer to Chapter 6.

Smart Cameras May Be Handled Differently:

Smart cameras can run independently of the PC, and so they typically have a Job already loaded and running. If your UI will be dealing with smart cameras, it is likely that you will simply be “monitoring” them. This means that you will simply connect to the device, and display results and images, but not load a new Job to it. To monitor a smart camera, your startup scenario would look more like this:

- Wait for discovery of the smart camera.
- Upon discovery, connect your ReportConnection(s).
- Display images and results when NewReport event received.

Note: You should understand that you absolutely can download a new Job to your smart camera each time your UI starts up, if that’s the behavior you want.

I/O is handled through the IOConnection object:

If you need to get or set I/O values, or you need to be notified when certain I/O points change state, then you can use the IOConnection object. This is covered in depth in Chapter 5.

Setup Capabilities are Provided by the SetupManager Control:

If your user interface needs to provide “Setup Mode” capabilities where the user can adjust tool positions and parameters, then you will need to use the SetupManager component. This is covered in depth in Chapter 10.

2

CHAPTER 2 Jobs, Steps, and Datums

Introduction to the Visionscape.Steps namespace

In this chapter, we'll discuss Jobs, and the Steps and Datums that construct them. We'll explain how to load a Job from disk, how you can then access each of the Steps within that Job, and how you can get and set any of the parameters (refered to as "Datums") of those steps. We'll describe the Step and Datum interfaces, and how to use them to find Steps, add or remove Steps, and how to get and set Datum values within a Step.

Assembly Names: Visionscape.dll & Visionscape.Steps.dll
Namespace: Visionscape.Steps

Jobs and Job Files

We use the term "Job" to refer to any Visionscape vision inspection program that you have created using our FrontRunner application, or from code (more about that later). When saved to file, a Job will always have the AVP file extension. For that reason, we also refer to Job files as AVPs. A Job is essentially a collection of Steps in a tree structure.

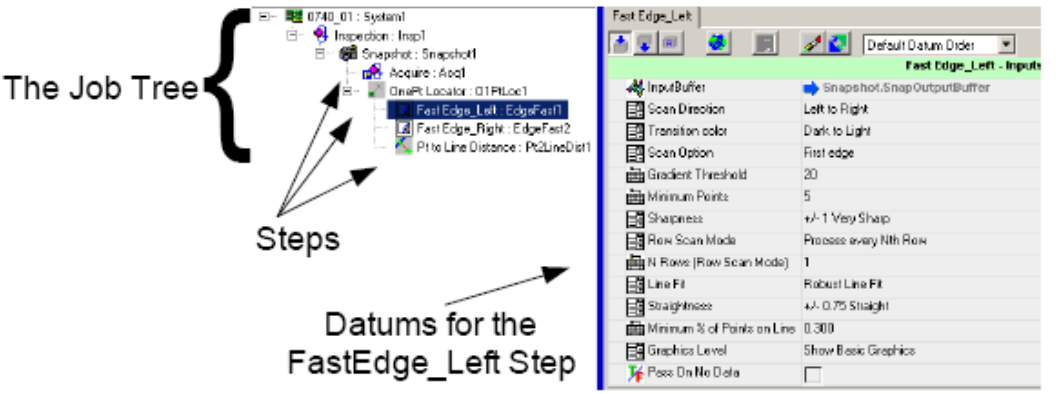
Steps

A Step is a single “tool” in a Vision Program. A user inserts Steps into the Job in order to add functionality. A Step may run a vision algorithm like the Blob Step or Fast Edge Step, it may perform measurements like the Pt to Line Distance Step, or it may perform logical operations like the IF Step or the VarAssign Step. Each Step contains a collection of Datums that configure its specific functionality.

Datums

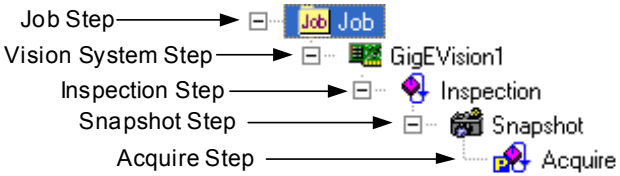
A Datum is a generic representation of a Step parameter. It encapsulates all types of data, such as integers, floating point values, arrays, etc. The “High Threshold” parameter of the Blob Step is an example of a Datum.

FIGURE 2–1. Example of a Job and the Steps and Datums Within It



Important Step Types

As shown above, a Job Tree is made up of a hierarchy of Steps. When writing a Visionscape user interface, you should understand the purpose of each of the Steps that are always at the top of the Hierarchy. The example here shows the Step hierarchy of a default Job created in FrontRunner.



JobStep:

The JobStep is the top-most Step in the Job hierarchy. It acts as the container for your Job, and therefore provides no direct functionality at runtime. You can only have one JobStep loaded at a time. A JobStep can hold the vision programs for multiple devices or just one.

VisionSystemStep:

The VisionSystemStep represents your hardware, and it holds the vision program for one device. So only one VisionSystemStep can run on a Device. A Job that contains multiple VisionSystem Steps is a Job that contains the vision programs for multiple Visionscape Devices. Each VisionSystemStep in your Job must be connected to a Device before it can run, this is done by either downloading it to the device, or calling the SelectSystem method (more on this in Chapter 3). When you build a Job in FrontRunner, you are only allowed to build a Job for a single Device, so these Jobs will only ever contain a single VisionSystem Step. Our I-PAK product can create Jobs that contain multiple VisionSystemSteps, and programmatically you can create a Job with multiple VisionSystemSteps. The example Job tree above shows a VisionSystem Step attached to the Device “GigEVision1”.

Inspection Step:

The Inspection Step contains all of the Steps that constitute an Inspection. It is this Step that manages running all of the Steps in your vision program. When you want to receive images and results from a running device, it is the Inspection Steps that will produce these reports. Each Inspection runs in a separate thread, independently from the others. The VisionSystemStep must contain at least one Inspection Step, but it can contain many.

Snapshot Step:

The Snapshot Step handles image acquisition in your inspection. You will generally insert one Snapshot Step for each camera you are using. You insert all of your vision tools inside of the Snapshot Step, as this Step produces an output buffer for your vision tools to run in. Although the Snapshot Step handles image acquisition, the parameters that govern image acquisition are not configured here, they are configured in the child Acquire Step.

Acquire Step:

Every Snapshot Step has a child Acquire Step. You can not add or delete this Step. The parameters that govern Image Acquisition are controlled via the datums of the Acquire Step. The selected camera, the I/O point to use as the Trigger, the image exposure time and many other parameters are all controlled by the datums of this Step. If you need to modify these parameters in your user interface, you will need to access the Acquire Steps in your Job.

Visionscape.Steps and The Step Object

Accessing Jobs and Steps in your Visual Studio program requires you to add a Reference to Visionscape.dll and Visionscape.Steps.dll. In your C# or VB.NET project, you go to the Solution Explorer, right-click on “References” in the project tree, and select “Add a Reference”. Under the .Net tab in the “Add a Reference” dialog, select the following items:

```
Visionscape
Visionscape.Steps
```

To provide easy access to the objects within this namespace, add the following statement to the top of your C# files (all sample code assumes this using statement is present):

```
using Visionscape.Steps;
```

JobStep

As we said above, the JobStep acts as the container for your vision programs, so the JobStep object will be used to load and save Jobs from disk. Here's a C# example of how you would load the sample "example_datamatrix.avp" file (installed with Visionscape) in your Form Load event:

```
private JobStep m_Job = new JobStep();

private void frmMain_Load(object sender, EventArgs e)
{
        //load the job file
        m_Job.Load("C:\Vscape\tutorials and Samples\Sample
Jobs\Data Matrix\example_datamatrix.avp");
}
```

In this example, we simply called the Load method of JobStep, and passed in the path to our AVP file. The AVP file is now loaded in memory, contained within our JobStep variable, m_Job. Using methods and properties of the JobStep object, we can access any of the Steps within the loaded Job, and any of their Datums. The JobStep object is a specialized form of the more generic Step object. To use Object Oriented Programming terminology, you would say that JobStep is derived from the Step Object. This means that the JobStep contains all of the methods and properties of the Step object, but also adds a few of its own, like the Load method shown in our example.

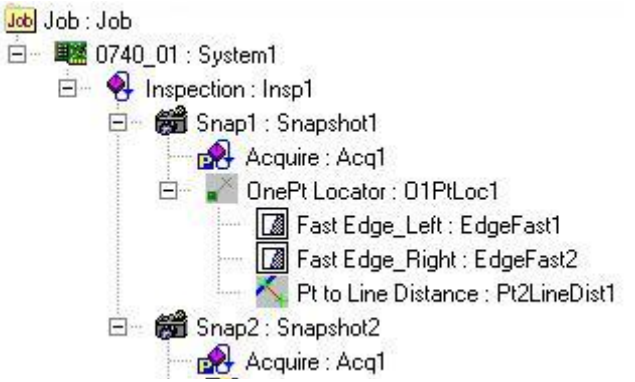
The Step Object

As we just described, the Step Object is the generic object (the base class if you will) upon which all Steps are built. You should understand the following key concepts about Steps.

Steps Are Collections

Each Step is also a collection that holds a list of the child steps that were inserted inside of it when the Job was built. So, you can enumerate the Steps of your Job just as you would the elements of any collection object. Consider the example Job tree in Figure 2–2, and the parent-child relationships of each step:

FIGURE 2–2. A Job Tree is a Collection of Collections



Parent Step	Child Count	Child Steps in the Collection
0740_01	1	Inspection:Insp1
Inspection	2	Snap1:Snapshot1 Snap2:Snapshot2
Snap1	1	OnePtLocator:O1PtLoc1
OnePtLocator	3	Fast Edge_Left:EdgeFast1 Fast Edge_Right:EdgeFast2 Pt to Line Distance:Pt2LineDist

So, let's assume that the Job we loaded from disk matches the Job tree shown in Figure 2–2. As you can see, the Job Step is always the top-most Step in the tree. The Job will always contain one or more VisionSystem steps in its collection. So, if we wanted to access the first VisionSystem step in the Job, we could simply do the following:

```
//Get the first Vision System Step in the Job
VisionSystemStep vs = (VisionSystemStep)m_Job[1];//1-based
collection
```

And, if we wanted to iterate through all the VisionSystem steps, we could do the following:

```
foreach(Step vs in m_Job)
{
     Console.WriteLine("The Name of this Step is " +
vs.Name);
}
```

The Step Object Provides Many Properties That Describe the Step

These include the Step's Name, Symbolic Name, the type of Step (Blob, Snapshot, Flaw, etc.), what category it falls under, is it trainable, is it trained, etc. For more information, see "The Major Properties That Describe A Step" on page 2-5 and "Step Object Properties" on page 2-35.

The Step Object Has Many Methods for Finding Child Steps

It's possible to find any step within a Job tree by simply using the collection methods of the Step Object, but this can sometimes require a fair amount of code. You may want to simply find all the Snapshots in your Job, or find the first Step named "My Fast Edge Tool". Fortunately, the Step Object provides several flexible methods that locate Steps quickly. We will cover these in more detail later in this chapter.

The Step Object can Add and Remove Steps From Your Job

The Step Object provides methods that allow you to create new steps and delete existing ones. It's actually possible to create an entire Job from code if you wish (though this is not generally recommended). Refer to "Adding and Removing Steps" on page 2-10 for more information.

Every Step Contains a Collection of Datums

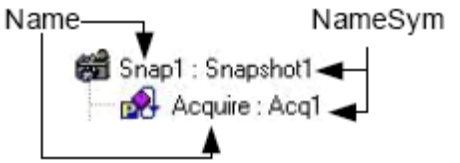
The Datums property of Step Object is a collection that contains a list of all of the Datums for that Step. You can get and set the values of any Step parameter via the Datum interface. More on that later. Now, let's cover each of these topics in more detail.

The Major Properties That Describe A Step

The Step Object has many properties and methods, but the following properties are the most commonly used, and the ones that provide the most valuable information to describe a Step.

- Name — Holds the name the user assigned to the step. You can change this name via code, or the user may change it while in FrontRunner.
- NameSym — The symbolic name that Visionscape assigned to the Step. This name is fixed and cannot be changed.

FIGURE 2–3. Name and Symbolic Name



- Trainable — Returns True if this Step is Trainable. Most Steps in Visionscape do not need to be trained and will return False for this property. Examples of Trainable steps are the Template Find Step, the OCV tools, DMR and IntelliFind®.
- Trained — Returns True for steps that are Trainable and are currently trained.
- Type — Returns a string that identifies the type of the Step. This will always come in the format:

```
Step.<type>.1
```

Where “<type>” would be replaced by the actual type of the Step. Some examples:

```
Snapshot Step = "Step.Snapshot.1"
Inspection Step = "Step.Inspection.1"
Fast Edge Step = "Step.Edgefast.1"
```

Refer to the StepBrowser.exe utility provided with Visionscape for a complete list of all Step types. You can use this utility to verify that a Step is of the proper type before you perform some specific operation. For example, perhaps you are looping through all the children of an Inspection step, looking for Snapshot Steps. You might write the following code:

```
//find the first Inspection step under the Job
Step insp = m_Job.FindByType("Step.Inspection")
```

```
//loop through all the children of the inspection step
foreach (Step child in insp)
{    //is this a Snapshot Step?
     if (child.Type == "Step.Snapshot.1")
     {
         Console.WriteLine("Found a Snapshot named " +
child.Name);
     }
}
```

- Category — Returns a value of type EnumAvpStepCategory that identifies the category of the step. The available categories are:
  - PostProc — This stands for Post Processing Step, and most Steps fall into this category. This means that the Step will run AFTER the processing of its parent. In other words, the Visionscape framework will run the parent first, then it will run this step.
  - PreProc — This stands for Pre Processing Step. A Step that is in this category will be run by the Visionscape framework before its parent step. An example of this would be the Acquire Step that is built into the Snapshot Step. The TwoPt Locator Step built into the OCV Fontless Step is another example. You may not delete Steps in this category, it's only deleted when its parent is deleted.
  - Private — This is a Step that was created by its parent Step, and is private to that Step. The owner of a Private Step is responsible for running it. You are not permitted to delete the step. Examples of this category are the AutoThreshold step in Blob, and the OutputValid step in the Inspection step.
  - Setup — A Step in this category was created by its Parent Step for the sole purpose of being used at Setup time. This category of Step does nothing at runtime. An example would be the Template Setup Step, which is built into the Template Find and One Pt Locator steps. This step provides you with an ROI to place around the template you wish to train on, but provides no functionality at runtime. This Step is only deleted when its parent is deleted.

– Part — This category designates Steps that are used for Calibration. Currently, this applies only to the Blob step that is added by the Calibration Manager when you attempt to Calibrate your Job. You may not delete a Part Step.

Finding Steps in the Step Tree

Several methods are provided in the Step object to make locating particular Steps or groups of Steps quick and easy.

Step FindBySymName(**string** *name*)

name: The symbolic name of the step you are searching for.

Calling this method causes the Step to search all of it's children for the child step with the given symbolic name. If successful, a reference to the located Step is returned, if not successful, an exception is thrown.

```
Step mystep = m_Job.FindBySymName("Snapshot1");
```

Step FindByName(**string** *name*)

name: The user assigned name of the step you are searching for.

Calling this method causes the Step to search all of it's children for the 1st child step with the given user name. If successful, a reference to the located Step is returned, if not successful, an exception is thrown.

```
Step mystep = m_Job.FindByName("My Snapshot");
```

Step FindByType(**string** *type*)

type: This string specifies the type of step to search for, in the form "Step.type".

- e.g. "Step.Snapshot", "Step.Inspection", "Step.EdgeFast" etc... The StepBrowser utility can be used to look up the type of any Step.

Calling this method causes the Step to search all of it's child steps for the 1st Step that matches the specified type. If successful, a reference to the located Step is returned, if not successful, an exception is thrown.

```
Step snap = m_Job.FindByType("Step.Inspection");
```

Composite Find(string *nameOrType*, **EnumAvpFindOption** *option*, **EnumAvpStepCategory** *whichCategory*)

- nameOrType — A string that specifies either the user name, symbolic name or step type that you are searching for.
- option — Specifies how you want to search, your options are...
  - FIND_BY_SYMNAME — Searches for the first Step with a Symbolic name that matches the string specified in the nameOrType parameter.
  - FIND_BY_TYPE — Searches for the first Step that matches the type specified in the nameOrType parameter.
  - FIND_BY_USERNAME — Searches for the first Step with a username that matches the string specified in the nameOrType parameter.
- whichCategory — Use this parameter when you want to search only for Steps within a given Step category. You will typically specify S_ALL.

Calling this method causes the Step to search only its child Steps for the first one that matches the search criteria. You can search for Steps by user name, symbolic name or by step Type, as specified using the Option parameter. You can also specify the Step category that you want searched using the whichCategory parameter. If successful, a reference to the located Step is returned. An exception is thrown if the Step cannot be found.

Examples:

```
Step insp, onept, blob;
//find the first Inspection step under the Job
insp = (Step)m_Job.Find("Step.Inspection",
 EnumAvpFindOption.FIND_BY_TYPE,
 EnumAvpStepCategory.S_ALL);
//find Step named "OnePt Locator" under the inspection
onept = (Step)insp.Find("OnePt Locator",
 EnumAvpFindOption.FIND_BY_USERNAME,
 EnumAvpStepCategory.S_ALL);
//find step with Symbolic Name "Blob1" under locator
blob = (Step)onept.Find("Blob1",
EnumAvpFindOption.FIND_BY_SYMNAME,
```

```
EnumAvpStepCategory.S_ALL);
```

Note: You may notice that when searching for the first Inspection step, we used the string “Step.Inspection” and not “Step.Inspection.1”. Either string will work, all of the find methods are smart enough to recognize when the “.1” is present or not.

StepList GetStepList(string *type*)

– type: A string that specifies the type of string to search for. This is in the form “Step.type”.

  • e.g. “Step.Snapshot”, “Step.Inspection”, “Step.EdgeFast” etc... The StepBrowser utility can be used to look up the type of any Step.

Searches the children of the Step for ALL steps that match the specified type. A reference to a StepList object is returned that contains all matches. Following is an example of finding all of the snapshot steps in a Job:

```
StepList allSnaps = m_Job.GetStepList("Step.Snapshot");
foreach(Step snap in allSnaps)
{
    Console.WriteLine("Snapshot Name = " + snap.Name);
}
```

IAvpCollection FindByType(string *stepType*, int *findInAllChildren*)

– stepType — A string that specifies the type of string to search for. This is in the form “Step.type”.

  • e.g. “Step.Snapshot”, “Step.Inspection”, “Step.EdgeFast” etc... The StepBrowser utility can be used to look up the type of any Step.

  • findInAllChildern – set to 1 to search ALL child steps, set to 0 to search immediate children only.

This version of FindByType is identical to GetStepList, the difference being that you can use the findInAllChildren parameter to specify that only immediate children should be searched, and not all levels of child Steps. This version also returns a reference to an IAvpCollection interface rather than a StepList. In general you should prefer GetStepList to this method. Following is an example of using FindByType to find all of the snapshots under a Job Step.

```
IAvpCollection snaps = m_Job.FindByType("Step.Snapshot", 1);
foreach (Step snap in snaps)
{
    Console.WriteLine("Snapshot Name = " + snap.Name);
}
```

Composite FindParent(**string** *stepType*)

- stepType — A string that specifies the type of Step you want to search for. This is in the form “Step.type” where “type” is the type of Step. You are not required to include the “.1” at the end of the find string.

This method walks up through the Job tree, searching the parents of the Step for the type specified in the stepType parameter. Typically, you would use this method when you want to find the parent Snapshot or Inspection of a given Step.

Examples:

```
//find the first fast edge step in the Job
Step fedge = m_Job.FindByType("Step.EdgeFast");
//find the parent Snapshot and Inspection steps of the
//FastEdge step
Step parentSnap = (Step)fedge.FindParent("Step.Snapshot");
Step parentInsp = (Step)
fedge.FindParent("Step.Inspection");
```

Composite ParentInspection { get; }

Composite ParentVisionSystem { get; }

Use these properties as a quick and easy way to access the parent Inspection Step or parent Vision System Step of a given Step object.

```
Step insp = (Step)mystep.ParentInspection;
Step vs = (Step)mystep.ParentVisionSystem;
```

Adding and Removing Steps

The Step object provides methods that allow you to add and remove Steps (with some limitations) from its collection of child steps. You use one of the versions of the AddStep method when you want to add a child step.

Step AddStep(string *type*)

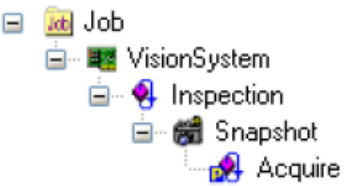
– type: The type of step that you wish to add. This is in the form "Step.Type"

  • e.g. "Step.Snapshot", "Step.EdgeFast", etc.

Adds a child step of the specified Step type to the calling Step's list of children. The new step will be added at the end of the child list. A reference to the newly added step is returned. You should understand that a Step can only add steps to it's own list of children. So if you want to add steps under one of the snapshot steps in your Job, then you must first locate that Snapshot Step, and call AddStep on it. You can not use the JobStep for instance to add steps to an Inspection Step. The following example illustrates this as we create the basic framework of a Job.

```
m_Job = new JobStep();
//add a Vision System step to our Job
VisionSystemStep vs =
(VisionSystemStep)m_Job.AddStep("Step.VisionSystem");
//now add an Inspection under the Vision System Step
Step insp = vs.AddStep("Step.Inspection");
//lastly, add a snapshot under the inspection
Step snap = insp.AddStep("Step.Snapshot");
```

The resulting Job would look like this (NOTE: The Acquire step is added automatically by the Snapshot Step):



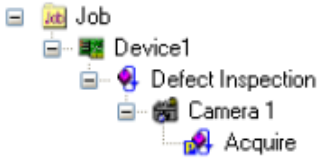
Step AddStep(string *type*, string *name*)

- type: The type of step that you wish to add. This is in the form "Step.Type"
  - e.g. "Step.Snapshot", "Step.EdgeFast", etc
- name: The name that should be assigned to the newly created Step.

This version of AddStep allows you to specify the user name of the Step, so the Steps can be created and named in one shot. Using our previous example, we could assign our own names to each of the Steps like this:

```
m_Job = new JobStep();
//add a Vision System step to our Job
VisionSystemStep vs =
(VisionSystemStep)m_Job.AddStep("Step.VisionSystem",
 "Device1");
//now add an Inspection under the Vision System Step
Step insp = vs.AddStep("Step.Inspection", "Defect
Inspection");
//lastly, add a snapshot under the inspection
Step snap = insp.AddStep("Step.Snapshot", "Camera 1");
```

Now the resulting Step tree would look like this:



Step AddStepAfter(string *type*, Step *relativeStep*)

- type : The type of step that you wish to add. This is in the form "Step.Type"
  - e.g. "Step.Snapshot", "Step.EdgeFast", etc
- relativeStep: This is the child Step that the new step will be added after.

Adds a new Step of the specified type into the calling Step's child list, but instead of being added at the end, the new step is added immediately after the Step specified by the relativeStep parameter.

Step AddStepBefore(string *type*, Step *relativeStep*)

- type: The type of step that you wish to add. This is in the form “Step.Type”
  - e.g. “Step.Snapshot”, “Step.EdgeFast”, etc
- relativeStep: This is the child Step that the new step will be added before.

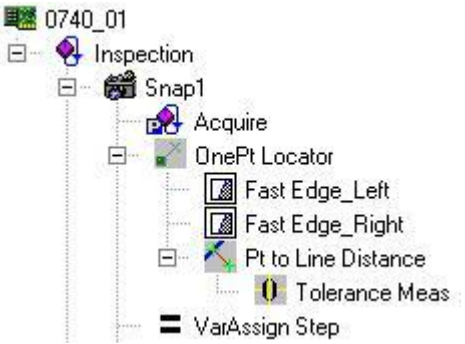
Adds a new Step of the specified type into the calling Step’s child list, but instead of being added at the end, the new step is added immediately before the Step specified by the relativeStep parameter.

Following are some additional examples of adding Steps to a Job.

Examples:

Assume we’ve loaded a Job that initially looks like this:

FIGURE 2–4. Initial Job

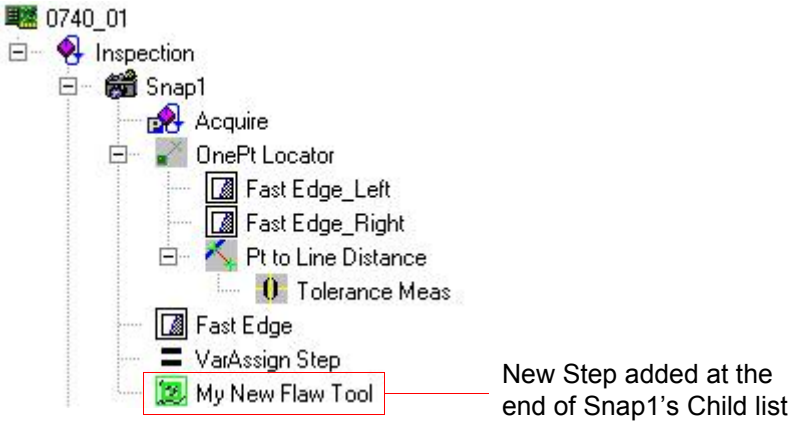


If we wanted to add a Flaw tool at the end of the Snapshot’s child list, we could run the following code:

```
//find the first snapshot step in the Job
 Step snap = m_Job.FindByType("Step.Snapshot");
 //add a new Flaw tool into the snapshot
 snap.AddStep("Step.FlawTool", "My New FlawTool");
```

Now, the Step Tree would look like this:

FIGURE 2–5. Job with Flaw Tool Added

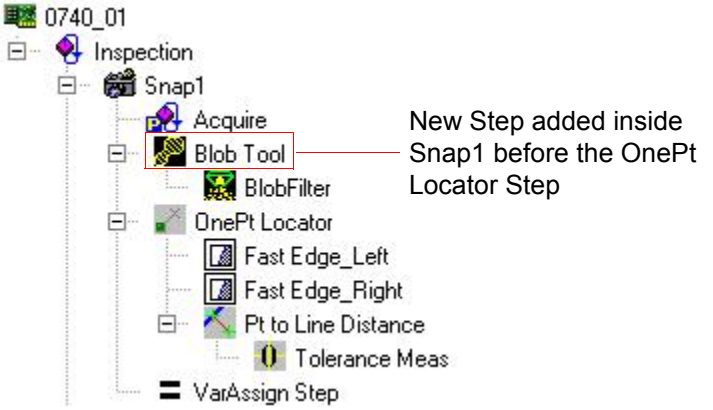


What if you wanted to add a new Blob tool under the Snapshot step, but you wanted it to be inserted before the One Pt Locator? Then, we could do the following:

```
//find the first snapshot step in the Job
 Step snap = m_Job.FindByType("Step.Snapshot");
 //find the One Pt locator under the snapshot
 Step onept = snap.FindByName("OnePt Locator");
 //add a new blob step into the snapshot, before the one
 //pt locator
 Step blob = snap.AddStepBefore("Step.Blob", onept);
 blob.Name = "My New Blob Tool";
```

Now, the Step Tree would look like this:

FIGURE 2–6. Job with Blob Tool Added

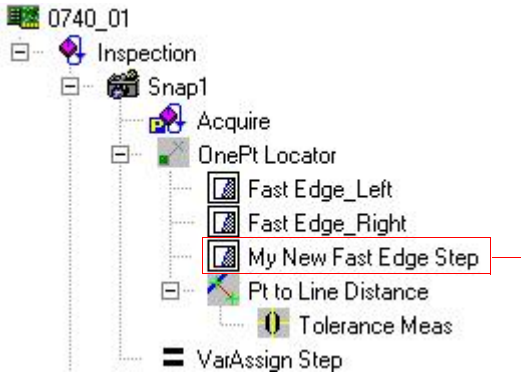


What if we wanted to add a new Fast Edge Step to the OnePt Locator step, but we wanted it to come immediately after the Fast Edge_Right Step? Then, we would do the following:

```
//find the 'Fast Edge_Right' step
Step fedge_right = onept.FindByName("Fast Edge_Right");
//add a new step into One Pt Locator,but After the Fast
Edge_Right step
Step fedge_new = onept.AddStepAfter("Step.EdgeFast",
fedge_right);
fedge_new.Name = "My New Fast Edge Step";
```

Now, the Step Tree would look like this:

FIGURE 2–7. Job with Fast Edge Added



Step AddStep(object *stepOrType*,
EnumAvpStepCategory *whichCategory*,
Step *relative*,
EAvpCAddOption *option*)

- stepOrType — A string that specifies the type of Step you want to add. This is in the form “Step.type” where “type” is the type of Step, like “Step.Blob”.
- whichCategory — Allows you to specify the category of the Step you are adding. In general you should specify S_POSTPROC. Refer to the description of the Category property for an explanation of the various step categories.
- relative — If you want your new Step to be added into the tree “relative” to some other Step, say just before or just after that Step, then you must use this parameter to pass in a reference to that “relative” Step. The value you specify in the option parameter determines where it’s inserted relative to this Step. Set to null for default behavior, which will insert the new Step at the end of the child list.
- option — Specifies where in the tree the new Step should be added. This parameter works together with the relative parameter. The available settings are:

- ADD_AFTER — The new Step will be added immediately after the Step specified in the relative parameter.
- ADD_BEFORE — The new Step will be added immediately before the Step specified in the relative parameter.

This version of AddStep provides the most options, but is also the most difficult to use. You would use this version when you want very precise control over how your new step will be added. Most situations can be handled with the previously documented versions of AddStep, but this version is still supported if needed. A reference to the newly added Step is returned if the function is successful. An exception will be thrown if unsuccessful. All of the examples shown previously can be accomplished with this version.

Step InsertStep(Step *insStep*)

– instep: The Step that is to be inserted into the child list.

This method can be used to insert an already existing Step into the calling Step's list of children. Assuming we had a VisionSystem Step variable named 'vs', we could add an Inspection Step to it like this:

```
InspectionStep insp2 = new InspectionStep();
vs.InsertStep((Step)insp2);
This is equivalent to this:
vs.AddStep("Step.Inspection");
```

To remove a Step, you use either the Remove or RemoveStep methods.

void Remove(int *Index*)

– index — This is the 1 based index of the Step you wish to remove from the Step's collection.

Example:

Continuing the example code from above, if we wanted to remove the Fast Edge Step we just added, we could simply do this:

```
onept.Remove(3); //remove the 3rd child step
```

If we wanted to remove the Step named "Pt to Line Distance" from our AddStep example, but didn't know what its index was, we could locate it, and then use its index property:

```
//Find the step named "Pt to Line Distance" (under onept)
Step ptlinedist = onept.FindByName("Pt to Line Distance");
//use the index from the step itself to remove it
onept.Remove(ptlinedist.Index);
```

void RemoveStep(int *Index*, **int** *delChildStep*)

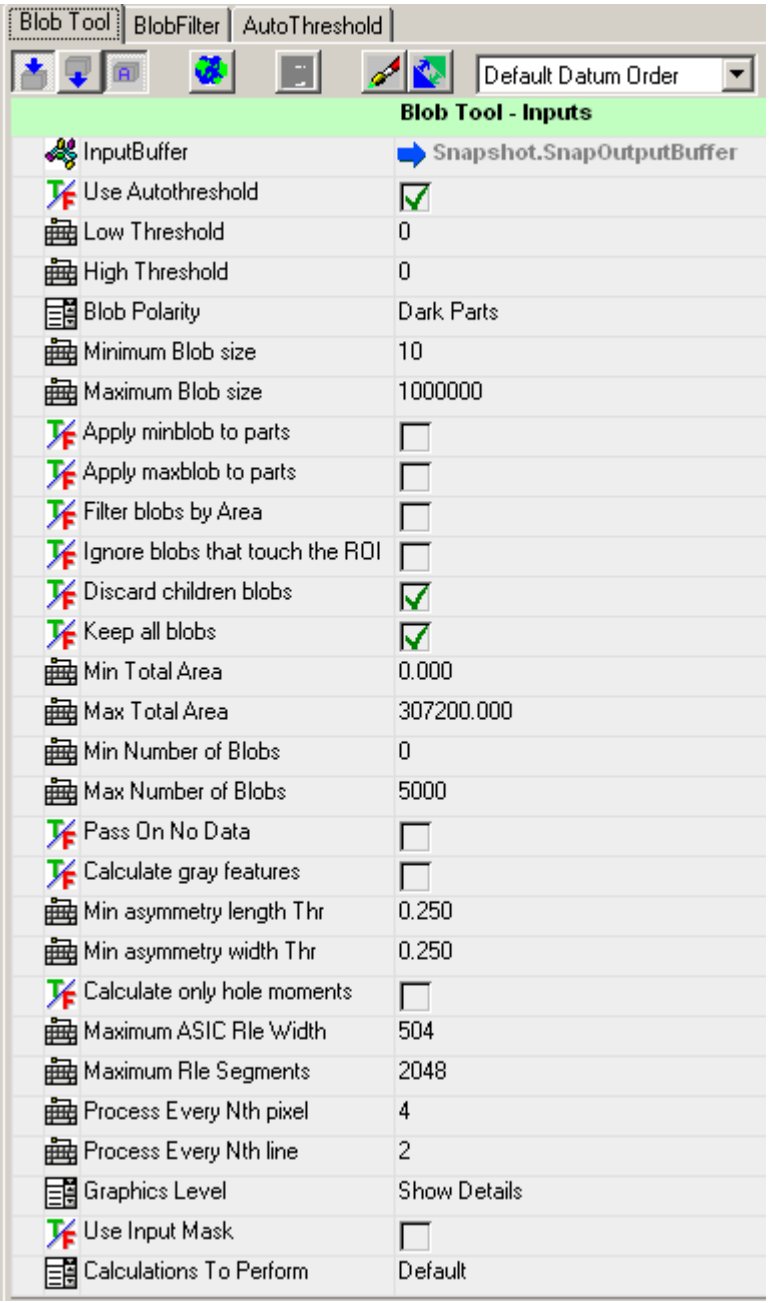
- index — This is the 1 based index of the Step you wish to remove from the Step's collection.
- delChildStep —
  - 1 — Remove the step from the collection AND delete it
  - 0 — The Step is removed from the parent step's collection but is NOT deleted from memory.

The only difference between Remove and RemoveStep is the delChildStep parameter of RemoveStep. This parameter allows you to only remove a Step from it's parent's child list, without actually destroying the object. Other than that they are functionally identical.

Accessing a Step's Datum Values

Every Step contains a collection of Datum objects. There is one Datum object for each of the Step's parameters, both input and output. Figure 2–8 shows the Datums for the Blob tool.

FIGURE 2–8. Blob Tool Datums



You can access a Step's Datums via the following properties and methods:

Datum Datum(string *datumSymName*)

– datumSymName: A string that represents the symbolic name of the Datum you want to access.

This method takes the symbolic name of the datum you want to access, and it returns a reference to that Datum object if it is found. An exception is thrown if not found. You would typically use this method when you want to read or write the value of an individual Datum within a Step. You can use the StepBrowser utility to look up the Symbolic Names of every Datum for every Step. For more information, see "Using StepBrowser to Look Up Symbolic Names" on page 2-30.

Example:

Continuing our AddStep example, we could take the newly added Blob Step and find its AutoTheshold and High Threshold Datums. We can then modify the values of each in order to turn the Auto Threshold capability off, and then set our own High Threshold.

```
//add a new blob step into the snapshot, before the one pt
//locator
    Step blob = snap.AddStepBefore("Step.Blob", onept);
    blob.Name = "My New Blob Tool";
    //find the Autothreshold Datum
    Datum AutoThresh = blob.Datum("UseAutoThr");
    //find the High Threshold Datum
    Datum HiThr = blob.Datum("HiThr");
    //now turn off AutoThreshold, and set High Threshold
    AutoThresh.Value = false;
    HiThr.Value = 150;
```

IAvpCollection Datums { get; }

This property returns a reference to the Step's collection of Datum objects. You would use this property whenever you want to iterate through all of a Step's Datums.

Example:

```
    //iterate through all of the datums in our blob step
    foreach(Datum d in blob.Datums)
    {
```

```
        Console.WriteLine("Datum Name = " + d.Name);
        Console.WriteLine("Datum Type = " + d.Type);
    }
```

IAvpCollection DatumList(EnumAvpDatumCategory *cat*)

Cat: Specifies a datum category. Available options are:

- D_INPUT: Return only input datums. These are Datums in which you set a reference to another datum.
- D_OUTPUT: Return only output datums
- D_RESOURCE: Return only Resource datums. These are Datums that have an editor and are directly modified by the user.
- D_ALL: Return all datums

This method allows you to specify a Datum category, and it will then only return a collection of the Datums that are within that category. Typically you would use this property when you wanted to analyze just a Step's Output or Input Datums.

Example:

```
IAvpCollection outs =
blob.DatumList(EnumAvpDatumCategory.D_OUTPUT);
foreach(Datum outDat in outs)
{
    Console.WriteLine("Datum Name = " + outDat.Name);
    Console.WriteLine("Datum Type = " + outDat.Type);
}
```

Modifying Datum Values

The Datum object has a value property that you use to both get and set its value. The value property will return an object, as the Datum object needs to wrap many different data types. Datums in Visionscape can hold integers, floating point values, strings, single dimension arrays like points (x,y, angle and scale), lines (a,b,c), and also two dimensional arrays like Point Lists, Blob Trees, etc.

Example of getting a Blob tool's High Threshold value:

```
//assume 'blob' is a reference to a blob step
//find the High Threshold Datum
Datum HiThrDatum = blob.Datum("HiThr");
//Get the high threshold value as an integer
int hiThresh = (int)HiThrDatum.Value;
```

Example of getting the ROI datum:

```
//get the blob's ROI datum
Datum ROI = blob.Datum("ROI");
//The ROI's value is an array of objects (doubles), verify
//this
if (ROI.Value.GetType().IsArray)
{   //Get the value of the ROI datum
    object[] roi = (object[])ROI.Value;
    //dump out the ROI's center x, y, width, height, and
    //angle
    Console.WriteLine("ROI Center X,Y: " + roi[0] + ", " +
roi[1] );
   Console.WriteLine("ROI Width, Height: " + roi[2] + ", " +
roi[3]);
    Console.WriteLine("ROI Angle: " + roi[4]);
}
```

Setting Datum values is just as easy. You simply assign your new value to the value property.

Example of setting the Blob tool's High Threshold value:

```
//find the High Threshold Datum
Datum HiThrDatum = blob.Datum("HiThr");
//set its value to 150
HiThrDatum.Value = 150;
```

Example of modifying the Blob tool's ROI Position:

```
//get the blob's ROI datum
 Datum ROI = blob.Datum("ROI");
 //get the current roi settings as an array
 object[] roi = (object[])ROI.Value;
 //move the roi left 20 pixels, and up 50 pixels
 roi[0] = (double)roi[0] + 20;
 roi[1] = (double) roi[1] - 50;
 //set the width and height to 80 x 200
 roi[2] = 80;
```

```
roi[3] = 200;
//now set the modified array back into the datum
ROI.Value = roi;
```

In the above example, we moved the Blob's ROI 20 pixels to the right and 50 pixels up, and we set the width to 80 and the height to 200 pixels. The easiest way to modify a Datum that takes an array is to get its current value, modify it, and set it back into the value property. This insures that the dimensions of your array will be correct.

As mentioned previously, the Datum object holds many different types of data. You can check the type of any Datum object by querying the read-only Type property. This returns a string with a similar format to the Step Object's Type property, in that it follows the format "Datum.Type.1". The following table lists the most common data types, and the corresponding Datum type that is used to represent it.

TABLE 2–1. Common Datum Types

Data Type	Corresponding Datum Type String
Angle	Datum.Angle.1
Area	Datum.Area.1
Boolean/Status	Datum.Status.1
Distance (A Double that can be calibrated)	Datum.Distance.1
Enumerated types (Datums displayed in a Combo Box)	Datum.Enum.1
Floating Point	Datum.Double.1
Integer	Datum.Int.1
Line	Datum.Line.1
Point	Datum.Point.1
ROI (region of interest)	Shape.Rect.1
String	Datum.String.1

In Table 2–2, we list all available Datum Types. For each type, we specify the format of the data returned by the value property, as well as the format expected when you try to set the value property.

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum Type	Get Value Returns	Set Value Takes
Datum.Angle	double	double or int
Datum.Area	double	double or int
Datum.Blob	Variant array of requested features for this blob. The returned array is of size (x), where x is the number of features requested. This is based on the value set in the “Calculations to Perform” Datum in your Blob tool. The possible values are: **Default** - Xcent, Ycent, Area, Color. **Basic** - Default results plus Angle, Nholes, Xmin, YatXmin, Xmax, YatXMax, YatYmin, Ymin, XatYmax, Ymax, Xdiff, Ydiff, Major, Minor, Arearatio, Minora, Minorb, Minorc, Majora, Majorb, Majorc. Area - Basic results plus Totarea, Holearea, Holeratio, Boxarea, Boxarearatio, Axratio. **All** - Area results plus PEround, Length, Width, Lenratio, Avgrad, Rmin, Rmax, Radratio, Rminang, Rmaxang, X3sign, Y3sign, Perimeter, Ppda, Rminx, Rminy, Rmaxx, Rmaxy.	Set not supported

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.BlobTree	Object array of requested features for a specific blob or all blobs. The returned array is of size (n,x), where n is the index of the blob, and x is the index of the feature requested. Refer to Datum.Blob for definition of each possible feature.	Set not supported
Datum.CalResult	Double array of size (3, 16). Contains both the forward and inverse linear transforms used for calibration as well as the calibration stats. (0,0) = Angle of cal target (0,1) (0,2) (0,3) (1,1) (1,2) (1,3) (2,1) (1,2) (2,3) = forward matrix (0,4) (0,5) (0,6) (1,4) (1,5) (1,6) (2,4) (1,5) (2,6) = inverse matrix (0,7) = Max Cal Residue (0,8) = Avg Cal Residue (0,9) = Pixels per unit X (0,10) = Pixels per unit Y (0,11) = Units per Pixel X (0,12) = Units per Pixel Y (0,13) = Camera angle (0,14) = Pix perspective error (0,15) = World perspective error	Double array of size (3,17). Array contents are the same as for Get Value. The 17th element should be set to 0.0.

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.CompList	Array of handles and names for all items in the list. Each item could be a Step or a Datum. Array is (x,2), where x is number of entries in the list. (x,0) = Handle to a Step. If the item in the list is a Step, the handle belongs to the Step. If the item is a Datum, the handle is to the owning Step of the datum. (x,1) = Symbolic name of the Step or Datum.	(x,2) array where x is the number of entries in this list (x,0) = Either the handle of the item (Step or Datum) or its complete symbolic name path (Step or Step.Datum) unique to the Datum.CompList search root (set by the owner of the datum) (x,1) = True or False to Add or Remove the entry from the list.
Datum.DblDmList	Array of double values	Set not supported
Datum.DblList	Array of double values	Array of double values
Datum.Distance	Double	Double

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.DMRResults	Array sized (n, 38), where n is the number of Matrices found + 1. The first row of the array is always populated with text labels that identify the data in each column, the actual matrix data then follows in each successive row. (n,0) = Decoded String (n,1) = Decoded? (boolean) (n,2) = Linked (boolean) (n,3) = found symbol type (int) (n,4) = num rows (n,5) = num cols (n,6) = ecc type (n,7) = format ID (n,8) = crc expected (n,9) = crc actual (n,10) = matrix angle (n,11) = error code (n,12) = total num linked (n,13) = Linked Position (n,14) = Pixels per Cell (n,15) = Symbol Height (n,16) = Symbol Width (n,17) = X1 (n,18) = Y1 (n,19) = X2 (n,20) = Y2 (n,21) = X3 (n,22) = Y3 (n,23) = X4 (n,24) = Y4 (n,25) = Locate Time (n,26) = Extent Time (n,27) = Size Time (n,28) = Warp Time	Set not supported

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.DMRResults (cont.)	(n,29) = Sample Time (n,30) = Decode Time (n,31) = Contrast (n,32) = Error Bits (n,33) = Damage % (n,34) = Border Match % (n,35) = Threshold Value (n,36) = Symbol Polarity (n,37) = Img Style	Set not supported
Datum.Double	Double	Double, Integer
Datum.Enum	Array containing the current selection in the first item (long) and the set of available selections (string) in the following items. Example: The Acquire Step's “CameraNumber” Datum, in which “Camera 2” was selected, would return an array that looks like this: (0) = 1 '0 based index of cur sel (1) = “Camera 1” (2) = “Camera 2” (3) = “Camera 3” (4) = “Camera 4”	Integer value containing index of current selection or the string identifying the new selection. Example: To change the CameraNumber selection to Camera 3, you could say: .value = 2 'select 3rd item OR .value = “Camera 3”.
Datum.Expression	Via the generic Datum interface, the Value property returns a Double that contains the last value of the evaluated expression. In order to retrieve the expression itself. You must use the ExpressionDm: Dim expr as ExpressionDm Dim strExpr as string 'get Inspection Step's 'criteria for inspection 'pass datum Set expr = insp.Datum(“PassCrit”) strExpr = expr.Expression	String value that contains the new expression.

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.FileSpec	String array containing a list of file names, or if the list is < 1, a single string	String value containing a file name or wildcard, or an array of strings containing files to add to the list. The value(s) you send are always added to the list, they do not replace the current list. To clear the list, pass an empty string ("").
Datum.FlexArray	Array sized (x,4) where x is the number of datums stored in the FlexArray plus 1. (0,0) contains the number of pages stored for each datum. Each of the remaining rows correspond to one variable: (x,0) is the handle of the datum, (x,1) is the symbolic name, (x,2) is the user name, and (x,3) is the category of the datum (output or resource).	Array sized (x,2) where x is the number of datums stored in the FlexArray plus 1. (0,0) contains the number of pages stored for each datum. Each of the remaining rows correspond to one variable: (x,0) is the handle of the datum and (x,1) is a boolean indicating Add or Remove.

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.InspectionResults	Array sized (x,3) where x is the number of ALL possible results, not just those selected for upload. (x,0) = Handle of the Datum that is available for upload (not the Step) (x,1) = Symbolic Name of the datum. (x,2) = True if the datum is to be uploaded, False if not. To determine which results are selected for upload, you must iterate through the array, checking for those entries where (x,2) = True.	Array sized (x,3). Where x is the number of results you are adding to the upload list. The contents are different then for Get: (x,0) = Handle of the Step (x,1) = Symbolic Name of the Datum (x,2) = True if you want to upload this datum, False if you are removing it from upload list.
Datum.Int	Int	Int

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.IOList	Integer, where upper word is the IO Type,
	Lower word is the 0 based IO index.
	Constants for the various IO types are:
	PHYSICAL = 1
	VIRTUAL = 2
	SENSOR = 3
	STROBE = 4
	ANALOGOUT = 5
	SLAVESENSOR = 6
	SERIAL TRIGGER = 11
	Note: These values are also represented
	by the enumerated type AvpIOType.
		Two options:
		1) Int, where upper word is IO type, lower word is index.
		NOTE: When setting a Serial Trigger, you will also need to specify the Trigger string. Use the IoListDm object, and call the SetTriggerString() method.
		`IOListDm trig = (IOListDm)acqstep.Datum("Trigger");`
		`//use Serial Trigger, 2nd //Port`
		`//type for serial trig = //11,`
		`//shift it to upper //word`
		`trig.Value = (11 << 16) | 1;`
		`//Trigger when "123" //received`
		`trig.SetTriggerString("123");`
		2) String that lists type and index. Supported only for sensor, physical and
		virtual IO. Format for each is:
		“Trigger 1”
		“Digital IO 3”
		“Virtual IO 22”

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.LayoutInfo	Array sized (x, 6) where x is the number of symbols in the layout, (x, 0) = symbol ID (x, 1) = x location of the symbol (x, 2) = y location of the symbol (x, 3) = correlation score for the symbol that was calculated when the layout was trained (x, 4) = width of the symbol in pixels (x, 5) = height of the symbol in pixels.	Same format as Get Value. The input data replaces the current LayoutData.
Datum.Line	Array of Doubles containing A,B,C. This is from the line equation Ax + By + C = 0	Array of Doubles containing A,B,C.
Datum.Matrix	Array of Doubles sized (x,y)	Array of Doubles sized (x,y)

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.OCVResults	Array of inspection result data sized (x, 18), where x is the number of symbols in the layout + 1. The first row of data contains text labels that identify the contents of each column. The actual data for each symbol starts in the 2nd row: (x, 0)-Passfail-Whether the symbol passed or failed the inspection (x, 1)-X location (x, 2)-Y location (x, 3)-X offset from the trained position (x, 4)-Y offset from the trained position (x, 5)-Correlation score (x, 6)-Sharpness value calculated (x, 7)-Sharpness value as a percentage of the trained sharpness value (x, 8)-Contrast value calculated (x, 9)-Contrast value as a percentage of the trained contrast value (x, 10)-Number of breaks found (x, 11)-Initial residue value (x, 12)-Initial residue value as a percentage of the symbol's trained area (x, 13)-Final residue value (x, 14)-Final residue value as a percentage of the symbol's trained area (x, 15)-The area of the largest blob found in the residue image	Set not supported

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.OCVResults (cont.)	(x, 16)-The area of the largest blob as a percentage of the symbol's trained area (x, 17)-The trained area	Set not supported
Datum.Point	Array of four floats. (0) = x (1) = y (2) = Angle in radians (3) = scale Note: Most tools will return data for just the X and Y values, and the angle and scale will default to 0 and 1 respectively.	Array of either 2 floats or 4 floats. Specify just X,Y if you want, or specify all 4 values. Format of the array should be the same as for Get Value.
Datum.PtList	Array of double point values (x,2) where x is the number of points, (x,0) is the point-x value, and (x,1) is the point-y value.	Array of double point values (x,2) where x is the number of points, (x,0) is the point-x value, and (x,1) is the point-y value.
Datum.Rect	Array of four Floats containing left, top, right, and bottom values.	Array of four Floats/Double/Integer values containing left, top, right, and bottom values.
Datum.Statistics	Datum.Statistics Single dimensional array with the following members: 0 - Owner Step Status (pass/fail) 1 - Measured value 2 - Nominal value 3 - Minimum value 4 - Average value 5 - Maximum value 6 - Standard Deviation 7 - Count	Single dimensional array with the following members: 0 - Owner Step Status (pass/fail) 1 - Measured value 2 - Nominal value 3 - Minimum value 4 - Average value 5 - Maximum value 6 - Standard Deviation 7 - Count
Datum.Status	Boolean	Boolean
Datum.Strelem	Array of size (x,y) containing a set of Boolean values.	Array of size (x,y) containing a set of Boolean values.

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.String	String	String
Datum.Struct	Array sized (x,2) where x is the number of datum elements in the struct. (x,0) is the error code associated with datum x. (x,1) contains the data from datum x and can be of any type.	Array sized (x,2) where x is the number of datums elements in the struct. (x,0) is the handle of the datum and (x,1) is a boolean indicating Add or Remove.
Datum.VerifyResults	A 1-dimensional array that provides the result data for either AIMDPM or ISO 15415 verification. (0) = Verification type (1) = Error code (2) = Decoded string (3) = Length of the decoded string (4) = Symbology type (5) = Aperture size (6) = Damage percentage (7) = Number of columns in the symbol (8) = Number of rows in the symbol (9) = Polarity (10) = Overall grade (11) = Reported grade (12) = Calibration status (13) = Cal target size 1 (14) = Cal target size 2 (15) = Cal target Rmin value (16) = Cal target Rmax value (17) = Error message (18) = Mean light value (19) = Process control on (20) = Global range active for process control parameters	Set not supported

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.VerifyResults (cont.)	(21) = Global good grade (22) = Global fair grade (23) = Contrast parameter active (process control) (24) = Contrast grade (25) = Contrast score (26) = Contrast grade required for “good” rating (27) = Contrast grade required for “fair” rating (28) = Modulation parameter active (process control) (29) = Modulation grade (30) = Modulation grade required for “good” rating (31) = Modulation grade required for “fair” rating (32) = Reflectance Margin active (process control) (33) = Reflectance Margin grade (34) = Reflectance Margin grade required for “good” (35) = Reflectance Margin grade require for “fair” (36) = Fixed Pattern Damage active (process control) (37) = Fixed Pattern Damager grade (38) = Fixed Pattern Damage grade required for “good” (39) = Fixed Pattern Damage grade required for “fair” (40) = Axial NonUniformity active (process control) (41) = Axial NonU grade (42) = Axial NonU score (43) = Axial NonU grade required for “good”	Set not supported (cont.)

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.VerifyResults (cont.)	(44) = Axial NonU grade required for “fair” (45) = Grid NonUniformity active (process control) (46) = Grid NonU grade (47) = Grid NonU score (48) = Grid NonU grade required for “good” (49) = Grid NonU grade required for “fair” (50) = Unused Error Correction active (process control) (51) = Unused Error Correction grade (52) = Unused Error Correction score (53) = Unused Error Correction grade required for “good” (54) = Unused Error Correction grade required for “fair” (55) = Minimum Reflectance active (process control) (56) = Min Reflectance grade (57) = Min Reflectance score (58) = Min Reflectance grade required for “good” (59) = Min Reflectance grade required for “fair” (60) = Print Growth active (process control) (61) = Print Growth grade (62) = Print Growth X (63) = Print Growth Y (64) = Print Growth grade required for “good” (65) = Print Growth grade require for “fair” (66) = Edge Determination active (process control) (67) = Edge Determination grade	Set not supported (cont.)

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.VerifyResults (cont.)	(68) = Edge Determination score	Set not supported (cont.)
	(69) = Quiet Zone active (process control)	
	(70) = Quiet Zone grade	
	(71) = Quiet Zone score	
	(72) = Quiet Zone grade required for “good”	
	(73) = Quiet Zone grade required for “fair”	
	(74) = Good Status (true if symbol was rated “good”)	
	(75) = Fair Status (true if symbol was rated “fair”)	
	(76) = Poor Status (true is symbol was rated “poor”)	
	(77) = Decodability active (process control)	
	(78) = Decodability grade	
	(79) = Decodability grade required for “good”	
	(80) = Decodability grade required for “fair”	
	(81) = Units	
	(82) = Cell size	
	(83) = Max exposure time used for calibration	
	(84) = Global range active for process control parameters	
	(85) = Global good grade	
	(86) = Global fair grade	
	(87) = Contrast parameter active (process control)	
	(88) = Contrast grade	
	(89) = Contrast score	
	(90) = Contrast grade required for “good” rating	

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.VerifyResults (cont.)	(91) = Contrast grade required for “fair” rating (92) = Modulation parameter active (process control) (93) = Modulation grade (94) = Modulation grade required for “good” rating (95) = Modulation grade required for “fair” rating (96) = Reflectance Margin active (process control) (97) = Reflectance Margin grade (98) = Reflectance Margin grade required for “good” (99) = Reflectance Margin grade require for “fair” (100) = Fixed Pattern Damage active (process control) (101) = Fixed Pattern Damager grade (102) = Fixed Pattern Damage grade required for “good” (103) = Fixed Pattern Damage grade required for “fair” (104) = Axial NonUniformity active (process control) (105) = Axial NonU grade (106) = Axial NonU score (107) = Axial NonU grade required for “good” (108) = Axial NonU grade required for “fair” (109) = Grid NonUniformity active (process control) (110) = Grid NonU grade (111) = Grid NonU score (112) = Grid NonU grade required for “good”	Set not supported (cont.)

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.VerifyResults (cont.)	(113) = Grid NonU grade required for “fair” (114) = Unused Error Correction active (process control) (115) = Unused Error Correction grade (116) = Unused Error Correction score (117) = Unused Error Correction grade required for “good” (118) = Unused Error Correction grade required for “fair” (119) = Minimum Reflectance active (process control) (120) = Min Reflectance grade (121) = Min Reflectance score (122) = Min Reflectance grade required for “good” (123) = Min Reflectance grade required for “fair” (124) = Print Growth active (process control) (125) = Print Growth grade (126) = Print Growth X (127) = Print Growth Y (128) = Print Growth grade required for “good” (129) = Print Growth grade require for “fair” (130) = Edge Determination active (process control) (131) = Edge Determination grade (132) = Edge Determination score (133) = Quiet Zone active (process control) (134) = Quiet Zone grade (135) = Quiet Zone score	Set not supported (cont.)

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.VerifyResults (cont.)	(136) = Quiet Zone grade required for “good” (137) = Quiet Zone grade required for “fair” (138) = Good Status (true if symbol was rated “good”) (139) = Fair Status (true if symbol was rated “fair”) (140) = Poor Status (true is symbol was rated “poor”) (141) = Decodability active (process control) (142) = Decodability grade (143) = Decodability grade required for “good” (144) = Decodability grade required for “fair” (145) = Units (146) = Cell size (147) = Max exposure time used for calibration	Set not supported (cont.)
Datum.Verify1D ResultsDm	A 1-Dimensional array that provides the result data for ISO15416 verification (1D symbols): (0) = Verification type (1) = Error code (2) = Decoded string (3) = Length of the decoded string (4) = Symbology type (5) = Aperture size (6) = Damage percentage (7) = Polarity (8) = Overall grade (9) = Reported grade (10) = Err message (11) = X Dimension	Set not supported

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.Verify1D ResultsDm (cont.)	(12) = Global threshold (13) = Decode Grade (14) = Decodability Grade (15) = Edge Determination Grade (16) = Symbol Contrast Grade (17) = Minimum Reflectance Grade (18) = Minimum Edge Contrast Grade (19) = Modulation Grade (20) = Defects Grade (21) = Quiet Zone Grade (22) = Calibration status (23) = Cal target size 1 (24) = Cal target size 2 (25) = Cal target Rmin value (26) = Cal target Rmax value (27) = Cal Max exposure (28) = Process control on (29) = Global range active for process control parameters (30) = Global good grade (31) = Global fair grade (32) = Edge Determination active (process control) (33) = Edge Determ grade required for “good” rating (34) = Edge Determ grade required for “fair” rating (35) = Decode active (process control) (36) = Decode grade required for “good” rating (37) = Decode grade required for “fair” rating	Set not supported (cont.)

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.Verify1D ResultsDm (cont.)	(38) = Symbol Contrast parameter active (process control) (39) = Sym Contrast grade required for “good” rating (40) = Sym Contrast grade required for “fair” rating (41) = Minimum Reflectance active (process control) (42) = Min Reflectance grade required for “good” (43) = Min Reflectance grade required for “fair” (44) = Min Edge Contrast active (process control) (45) = Min Edge Con grade required for “good” (46) = Min Edge Con grade required for “fair” (47) = Modulation parameter active (process control) (48) = Modulation grade required for “good” rating (49) = Modulation grade required for “fair” rating (50) = Defects active (process control) (51) = Defects grade required for “good” (52) = Defects grade require for “fair” (53) = Decodability active (process control) (54) = Decodability grade required for “good” (55) = Decodability grade required for “fair" (56) = Quiet Zone active (process control)	Set not supported (cont.)

TABLE 2–2. Datum Types, Get Values, and Set Values

Datum.Verify1D ResultsDm (cont.)	(57) = Quiet Zone grade required for “good” (58) = Quiet Zone grade required for “fair” (59) = Good Status (true if symbol was rated “good”) (60) = Fair Status (true if symbol was rated “fair”) (61) = Poor Status (true is symbol was rated “poor”) (62) = Units (63) = Bar Width	Set not supported (cont.)
Datum.Verify1D ScanResultsDm	A 2-Dimensional array that provides the grades and scores for each of the 10 scans performed by ISO15416 symbol verification. Size of array is (10,19). Where n = scan index: (n,0) = Final Grade for scan (n,1) = Edge Determination Grade for this scan (n,2) = Decode grade for this scan (n,3) = Symbol Contrast grade (n,4) = Modulation grade (n,5) = Minimum Edge Contrast grade (n,6) = Minimum Reflectance grade (n,7) = Defects grade (n,8) = Decodability grade (n,9) = Quiet Zone grade (n,10) = Symbol Contrast score (n,11) = Modulation score (n,12) = Min Edge Contrast score (n,13) = Min reflectance min score	Set not supported

TABLE 2–2. Datum Types, Get Values, and Set Values

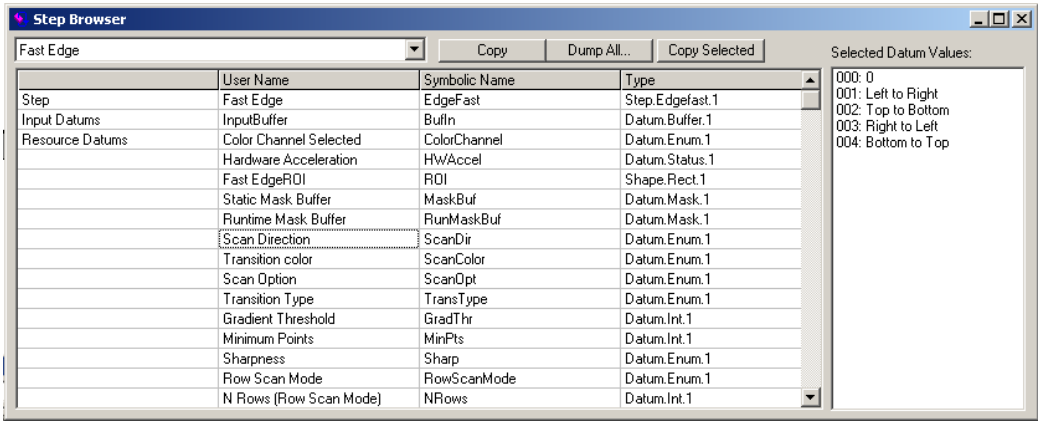
Datum.Verify1D ScanResultsDm (cont.)	(n,14) = Min reflectance max score (n,15) = Defects score (n,16) = Decodability score (n,17) = Quiet Zone 1 score (n,18) = Quiet Zone 2 score	Set not supported (cont.)
Shape.Rect (ROI)	Object Array of 13 items (0) = center X (1) = center Y (2) = width (3) = height (4) = angle (in radians) (5) = Pt1 X (top-left pt) (6) = Pt1 Y (top-left pt) (7) = Pt2 X (top-right) (8) = Pt2 Y (top-right) (9) = Pt3 X (bottom-right) (10) = Pt3 Y (bottom-right) (11) = Pt4 X (bottom-left) (12) = Pt4 Y (bottom-left) Note: 5-12 are expressed as offsets from the ROI's anchor point.	Object Array of 5 OR 13 items Array elements are the same as those listed to the left. You can pass a 5 element array if you just want to modify the location, width, height and angle. The full 13 element array is only required in the rare instances when you want to modify the control points as well. Although index 4 in the array takes an Angle, many Steps in Visionscape cannot be rotated. So, those Steps will ignore any angle you pass in.
Shape.Rhombus	Array of (4,2) double values containing the four points that describe the rhombus. (x,0) is the point-x and (x,1) is the point-y.	Array of (4,2) double values containing the four points that describe the rhombus. (x,0) is the point-x and (x,1) is the point-y.

Using StepBrowser to Look Up Symbolic Names

The StepBrowser utility is very useful for looking up information on the various Steps available in Visionscape. Specifically, it allows you to select any type of step from a drop-down list, and then all relevant programming information for that Step is displayed in the window. Specifically, it displays the strings that represent its Step.type and symbolic name, and most importantly, a list of all of its Datums. The Datum list includes the standard name, symbolic name and datum type of each. StepBrowser can be found in Start > Programs > Microscan Visionscape > Tools > Step Browser.

Once launched, StepBrowser looks like this:

FIGURE 2–9. StepBrowser



Using the combo box at the top left of the window, you can select any type of Visionscape Step. In the example in Figure 2–9, we've selected the Fast Edge Step. Once selected, StepBrowser shows you the default user name, the symbolic name, and the type in the first row of the grid for the selected Step. All subsequent rows contain a list of every Datum for the selected Step. This list also provides the default user name, symbolic name and Datum Type for each. This information is very useful when you are writing code to find and/or modify the values of many different Datums in many different Steps.

The JobStep Object

The JobStep derives from the Step object, and therefore it contains all of its properties and methods. However, it provides many additional capabilities as well, which we wil cover in this section. As our previous examples have already demonstrated, the JobStep loads an AVPfile, or “Job” from disk. The JobStep can also save the Job after you have modified it, and it also provides some useful utility functions:

void Load(string *fileName*)

– filename — A string that specifies the path to the AVP file you wish to load. All of the VisionSystemSteps in the specified AVP will be added to this JobStep. So, if you already have an AVP file loaded, and you now want to replace that AVP with a new one, you must remember to delete all the existing VisionSystemSteps in your Job (using the Remove method) before calling Load.

```
//loop until the Job Step has no children
 while(m_Job.Count > 0)
 {
     m_Job.Remove(1);
 }
 //now we can load our new job
 m_Job.Load(strAvpPath);
```

InspectionStep LoadInspection(VisionSystemStep *pSystem*, **string** *fileName*, **InspectionStep** *replaceObj*)

– pSystem — The VisionSystemStep into which the Inspection should be loaded

– filename — The file path to the InspectionStep AVP file.

– replaceObj — A reference to an existing InspectionStep that should be replaced by this inspection. Set to null if you want the Inspection to be added to the VisionSystemStep.

Loads an InspectionStep AVP file into the given VisionSystemStep. This is an AVP that contains an Inspection Step as it’s root step, meaning there is no VisionSystem step in the file. You can specify an existing InspectionStep that you want to replace with the newly loaded InspectionStep. This method returns a reference to the newly loaded InspectionStep.

void SaveAll(**string** *fileName*)

Saves the entire contents of the Job Step to disk. If your Job contains multiple Vision System Steps, they will all be saved in one file. The Job is saved to the file path specified by the filename parameter.

void SaveSystem(**string** *fileName*, **VisionSystemStep** *whichSystem*)

Saves only the specified Vision System Step to the file specified by fileName.

void SaveInspection(**InspectionStep** *pInsp*, **string** *fileName*)

Saves only the specified InspectionStep to the file specified by filename.

bool SelectSystem(**Visionscape.Devices.VsDevice** *dev*)

Connects the first VisionSystem Step in the Job to the specified Device. Refer to the VisionSystemStep documentation for more information on SelectSystem.

bool SelectSystem(**string** *sysName*)

Connects the first VisionSystemStep in the Job to the device with the name specified by the sysName parameter.

VisionSystemStep VisionSystemStep()

This method will return the first VisionSystemStep in the Job. This allows you to write code that is more easly read.

This:

```
VisionSystemStep vs = m_Job.VisionSystemStep();
```

Rather than this:

```
VisionSystemStep vs = (VisionSystemStep) m_Job[1];
```

VisionSystemStep VisionSystemStep(**int** *index*)

This overloaded version of the VisionSystemStep method takes a 0 based index that specifies the exact VisionSystemStep that you want. Returns null if the index is out of range.

object AVPFileInfoGet(**string** *fileName*)

Pass this function the name of an AVP file, and it will read the header of that file, and return you an 8 element object array with the following information:

The first 2 elements provide information on the version of Visionscape that this AVP was saved under. Typically, a Visionscape version would be presented as such:

4.1.1 build 34.

AvpInfo(0) = Integer.
Upper word = minor version number (2 in example)
Lower word = build number (34 in example above)

AvpInfo (1) = Integer.
Upper word = Major version number (3 in example)
Lower word = middle version number (7 in example)

AvpInfo (2) = Integer. Total objects contained in the file.

AvpInfo (3) = Integer. Total size of the file.

AvpInfo (4) = Integer. Total number of Steps in the Job.

AvpInfo (5) = Integer. Total Step size

AvpInfo (6) = String. Identifies the AVP type. There are 3 possible values.

– “SYSTEMSTEP” File contains just one VisionSystem Step

– “JOBSTEP” File contains multiple VisionSystem Steps

– “INSPSTEP” File contains a single Inspection Step

AvpInfo (7) = Integer. Digitizer type.

EnumPCPriority PCPriorityRuntime { set; get; }

This utility function provides an easy way to modify the process priority of your user interface. Options are:

– PP_CLASS_NORMAL Normal Process Priority

– PP_CLASS_HIGH High Process Priority

– PP_CLASS_REALTIME Realtime Process Priority (Default)

When you are running with Visionscape Host based System, you should always run your process at Realtime in order to prevent timing spikes. If, however, you are not concerned about timing spikes, and you would rather not run your user interface as a Realtime process, then use this property to lower the process priority to either High or Normal.

IComposite HandleToComposite(**int** *hnd*)

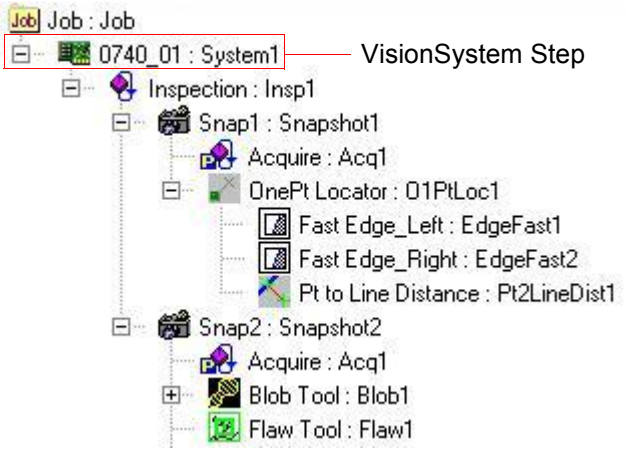
Converts a Datum handle or Step handle into an actual Datum object or Step object. The function returns a reference to a Composite object, which is the base class for both Step and Datum objects. Typically, the Legacy Setup Manager, Job Manager and Runtime Manager controls deal with Step and Datum handles, rather than actual Step and Datum objects. This very useful utility function can be used to convert those handles into actual objects when needed.

The VisionSystemStep Object

Just as the JobStep provides custom functionality on top of the standard Step properties and methods, so, too, does the VisionSystemStep object. Therefore, you should declare a variable of type VisionSystemStep when trying to access one:

```
VisionSystemStep vs = m_Job.VisionSystemStep();
```

FIGURE 2–10. VisionSystemStep



The VisionSystemStep represents your Visionscape hardware. When you load a Job into your JobStep, the immediate children of the JobStep will always be VisionSystemSteps. A Job constructed in FrontRunner will always contain one and only one VisionSystemStep. A Job constructed in I-PAK, or from code,could, however, contain more than one. After loading a Job, it's not ready to run until all of the VisionSystemSteps have been connected to a Visionscape Device (refer to "Visionscape Devices" on page 1-3 for more information on Devices). The VisionSystemStep provides special methods to connect to a Visionscape Device, and to query the current and past Device connection names. A list of these methods follows.

int SelectSystem(**string** *systemName*)

– systemName — A string that specifies the name of the Visionscape device that this VisionSystemStep should run on.

Refer to "Connecting Jobs to Visionscape Devices" on page 3-6 for examples of how to use the SelectSystem method. In general, you are better off using the Download method of the VsDevice object to connect your hardware, as this works for all devices. However, when using Visionscape Host based Systems, you may call SelectSystem directly if you wish.

string SelectedVisionSystem { get; }

This property allows you to query the name of the system/device that the VisionSystemStep is currently connected to.

string SystemLastSavedAs { get; }

This read-only property returns the name of the Device that was selected the last time this Job was saved. Typically, you would use this property to tell your application which Device it should connect to after you have loaded a Job from disk, since it's likely that your Job will always run on the same Device.

Step Object Properties

This section documents all of the properties of the Step Object.

BufferDm BufferInput { get; }

Read-only. Returns the input buffer Datum (if any) in the form of a BufferDm object.

BufferDm BufferOutput { get; }

Read-only. Returns the output buffer Datum (if any) in the form of a BufferDm object.

int CanRunUntrained { get; }

Read-only. Returns True if the Step can run untrained. For example, the Data Matrix step is a trainable step, but it can run untrained.

EnumAvpStepCategory Category { get; }

Read-only. Returns the category of this Step in relation to its Parent. Possible values are:

- S_POSTPROC— This stands for Post Processing Step, and most Steps fall into this category. This means that the Step will run AFTER the processing of its parent. In other words, the Visionscape framework will run the parent first, then it will run this step.
- S_PREPROC— This stands for Pre Processing Step. A Step that is in this category will be run by the Visionscape framework

before its parent step. An example of this would be the Acquire Step that is built into the Snapshot Step. The TwoPt Locator Step built into the OCV Fontless Step is another example. You may not delete Steps in this category, it's only deleted when its parent is deleted.

– S_PRIVATE— This is a Step that was created by its parent Step, and is private to that Step. The owner of a Private Step is responsible for running it. You are not permitted to delete the step. Examples of this category are the AutoThreshold step in Blob, and the OutputValid step in the Inspection step.

– S_SETUP— A Step in this category was created by its Parent Step for the sole purpose of being used at Setup time. This category of Step does nothing at runtime. An example would be the Template Setup Step, which is built into the Template Find and One Pt Locator steps. This step provides you with an ROI to place around the template you wish to train on, but provides no functionality at runtime. This Step is only deleted when its parent is deleted.

– S_PART— This category designates Steps that are used for Calibration. Currently, this applies only to the Blob step that is added by the Calibration Manager when you attempt to Calibrate your Job. You may not delete a Part Step.

string Cookie { set; get; }

Allows you to attach custom string data to a Step. This data is not saved with the Step when you save your Job to disk. Use the Tag property if you need your string to be saved to disk.

int Count { get; }

Returns the number of child steps under this step.

Datum Datum(**string** *datumSymName*)

Read-only. Returns the Datum with the specified symbolic name. An exception will be thrown if the name is invalid.

IAvpCollection DatumList(**EnumAvpDatumCategory** cat)

Read-only. Returns the list of Datums for this Step for a specific category.

IAvpCollection Datums { get; }

Read-only. Returns the list of all Datums for this Step. Refer to “Accessing a Step’s Datum Values” on page 2-15 for more information on the Datum, DatumList and Datums properties.

EnumEditabilityFlags Editability { set; get; }

Returns/sets the Editability Flags. You can combine the available options to get/set how the Step is handled in the Setup environment. This property is available for both the Step and Datum objects. Not all of the available flags apply to the Step object; the following are the only ones that do:

– EF_CANMASK — Step can be masked

– EF_CANROTATE — Step is rotatable

int Handle { get; }

Read-only. Returns the Handle of the Step. This is important when dealing with legacy controls (Runtime Manager, Setup Manager…) that work with Step handles, rather than references to Step objects.

IComposite HardwareDatum { get; }

Read-only. Returns the VisionDescDm from the parent VisionSystemStep. This datum contains information about the currently selected hardware.

int Index { get; }

Read-only. Returns the index of this Step in the collection of its Parent.

int LastError { get; }

Read-only. Returns the error code from the last time the Step ran. This will be 0 when there is no error.

string Name { set; get; }

Returns/sets the Name of the Step. This is also referred to as the user name, as the user is free to change this name at will.

string NameSym { get; }

Read-only. Returns the symbolic name of this Step. This name is assigned by the Visionscape Framework when the Step is created, and it will never change.

IStep Next { get; }

Read-only. Returns the next sibling in the currently selected Step category.

IComposite Parent { get; }

Read-only. Returns the Parent of this Step. Returns a Composite, but understand that this can be assigned to a Step object, as Composite is the base class of the Step and Datum classes.

IComposite ParentInspection { get; }

Read-only. Returns the parent Inspection Step of this Step.

IComposite ParentVisionSystem { get; }

Read-only. Returns the parent VisionSystem Step of this Step.

IStep Prev { get; }

Read-only. Returns the previous sibling in the current Step category.

int Rotatable { get; }

Read-only. Returns True if this Step can be rotated.

string Tag { set; get; }

Returns/sets custom string data for this Step. Similar to the Cookie property, only this data will be saved to disk with the Step when you save your Job.

int Trainable { get; }

Read-only. Returns True if the Step is trainable.

int Trained { get; }

Read-only. Returns True if the Step has been trained.

string TrainInfo { get; }

Read-only. Gets train information for this step.

string Type { get; }

Read-only. Returns the Type of Step. This will be in the form "Step.type.1".

Examples:

– Inspection step — "Step.Inspection.1"

– Snapshot step — "Step.Snapshot.1"

– Fast Edge Step — "Step.Edgefast.1"

int Untrainable { get; }

Read-only. Returns True if the Step is untrainable.

Step Object Methods

This section documents all of the methods of the Step Object:

Step AddStep(**string** *type*)
Step AddStep(**string** *type*, **string** *name*)
Step AddStep(**object** *stepOrType*,
EnumAvpStepCategory *whichCategory*,
Step *relative*,
EAvpCAddOption *option*)

Adds a Step of the specified type to the calling Step's child list. A reference to the newly added Step is returned.

Step AddStepBefore(**string** *type*, **Step** *relativeStep*)

Adds a Step of the specified type to the calling Step's child list, but adds it BEFORE the step specified by relativeStep. The relativeStep parameter must be a child of the calling Step. Refer to "Adding and Removing Steps" on page 2-10 for a more detailed description.

Step AddStepAfter(**string** *type*, **Step** *relativeStep*)

Adds a Step of the specified type to the calling Step's child list, but adds it immediately AFTER the step specified by relativeStep. The relativeStep parameter must be a child of the calling Step. Refer to "Adding and Removing Steps" on page 2-10 for a more detailed description

void ApplyChanges(**bool** *doingPaste*)

Call to apply the set of changed data to the Step. When you are modifying a Step's Datum values from code, it's a good idea to call this method after you are done. This method tells the Step that it should update it's internal data accordingly.

int CanBeContainedBy(**IStep** *thisStep*)

Returns True if the calling Step can be contained by the Step specified by the thisStep parameter.

int CanContain(**IStep** *thisStep*)

Returns True if the calling Step can contain the Step specified by the thisStep parameter.

int ChildCount(**EnumAvpStepCategory** *Category*)

Returns count of children in a specific category.

Step FindBySymName(**string** *name*)

Finds the first child Step with the specified symbolic name. Throws an exception if the Step does not exist. Please refer to "Finding Steps in the Step Tree" on page 2-8 for a more detailed description.

Step FindByName(**string** *name*)

Finds the first child Step with the specified user name. Throws an exception if the Step does not exist. Please refer to "Finding Steps in the Step Tree" on page 2-8 for a more detailed description.

Step FindByType(**string** *type*)

Finds the first child Step of the specified type. Type is in the form "Step.Inspection". Returns a reference to the Step if found, throws an exception if not found. Please refer to "Finding Steps in the Step Tree" on page 2-8 for a more detailed description.

IAvpCollection FindByType(string *stepType*, **int** *findInAllChildren*)

Returns an AvpCollection of all children that match the given type. Can choose to search all child steps or only immediate children using the findInAllChildren parameter. Refer to “Finding Steps in the Step Tree” on page 2-8 for a more detailed description.

Composite FindParent(string *stepType*)

Finds the Parent Step that matches the specified type. The stepType parameter is in the form “Step.Type”. Please refer to “Finding Steps in the Step Tree” on page 2-8 for a more detailed description.

Composite Find(string *nameOrType*, **EnumAvpFindOption** *option*, **EnumAvpStepCategory** *whichCategory*)

Finds the first child Step that matches the specified criteria. You can choose to search for child Steps based on their symbolic name, user name or type. Can also select the specific Step category to search. Refer to “Finding Steps in the Step Tree” on page 2-8 for a more detailed description

IStep FirstChild(EnumAvpStepCategory *Category*)

Returns first child Step in the specified category.

StepList GetStepList(string *type*)

Returns a list of all child Steps that are of the specified type. Prefer this method over the FindByType method.

IAvpCollection GetChildList(EnumAvpStepCategory *cat*)

Creates an AvpCollection of child Steps that are in the specified category.

int IsTimingEnabled()

Returns True if timing is enabled.

void Remove(int *Index*)

Removes the child Step at the specified index from the calling Step’s collection. The removed Step is always deleted. Please refer to “Adding and Removing Steps” on page 2-10 for more information.

void RemoveStep(int *Index*, **int** *delChildStep*)

Removes a Step from the set of children, optionally delete it. Refer to “Adding and Removing Steps” on page 2-10 for more information.

int RunWithPreRun(IDatum *pChangedDatum*)

When calling this method the Step will first run it’s PreRun function (it’s initialization routine), then it’s UIAction function (where it responds to changes to specific Datum values), and then Runs the step. This function returns true if the Step executed successfully (not necessarily passed). Typically, you would call this method when you have changed a datum value, and then want to run the Step to see the effect. Pass in a reference to the modified datum, or pass in null if you simply want to run the Step.

int TagForUpload(string *datumName*, **int** *bAdd*)

Either adds or removes the specified datum to/from the parent Inspection’s “Results to Upload” list. If bAdd = 1, the Datum is added to the list, if bAdd = 0, the datum is removed from the list.

int Train()

Trains the step, returns True if passed.

void UIAction_Apply(IDatum *pChangedDatum*)

Sends UIAction 'Apply' notification to the step for the given datum. If you are changing Datums in code, it is good practice to call this method because the Step may perform special initialization based on the value of the Datum you are modifying.

int Untrain()

Untrains the Step.

Datum Object Properties

This section documents all of the properties of the Datum Object:

EnumAvpDatumCategory Category { get; }

Read-only. Returns the category of this Datum. Typically, this is used to check whether a datum is an output an input or a resource datum. Categories are defined by the EnumAvpDatumCategory constants.

– D_All — Signifies all datum types.

– D_INPUT — An input datum that references another datum.

– D_OUTPUT — Indicates an output datum.

– D_PRIVATE — Indicates a private datum.

– D_RESOURCE — An input datum that requires user input to provide the value.

string Cookie { set; get; }

Allows you to store custom string data in this datum. This data will NOT be saved to disk. Use the Tag property if you want to save custom data along with the datum or Step.

EnumEditabilityFlags Editability { set; get; }

You will query the bits of this property to determine the editability settings for the datum. The values defined by EnumEditabilityFlags indicate the meaning of each bit. The following options are available for datums:

– EF_ALWAYSUPLOAD — When set, this datum's value will always be included in the list of uploaded results.

– EF_CAN_MODIFY_AT_RUNTIME — When set, this flag indicates that this datum can be modified at runtime. YOU SHOULD NEVER CHANGE THE STATE OF THIS FLAG.

– EF_ASK_MODIFY_AT_RUNTIME — When set, this flag indicates that the datum will ask its parent if its OK to modify its value at runtime. YOU SHOULD NEVER CHANGE THE STATE OF THIS FLAG.

– EF_CANMOVE

– EF_CANROTATE

– EF_CANSCALE

– EF_CANSIZE

– EF_CANSTRETCH

These options all apply to shapes only (e.g. the ROI), and indicate whether the shape can be moved, rotated, scaled, sized or stretched. YOU SHOULD NEVER CHANGE THE STATE OF THIS FLAG.

– EF_ENABLEREFEDIT — Indicates that this datum uses a reference editor, and that it's enabled.

– EF_NOUSERUPLOAD — When set, indicates that the user cannot select this datum for upload, but it WILL be included automatically in the list of uploaded results.

– EF_REFDATUM — Indicates that this datum can be set to reference other datums.

int Handle { get; }

Read-only. Returns the Handle of the Datum.

int IsReference { get; }

Read-only. Returns True if the Datum is referencing another Datum.

int LastError { get; }

Read-only. Returns the error code from the last time the parent step ran.

string Name { set; get; }

Returns/sets the Name of this object.

string NameSym { get; }

Read-only. Returns the symbolic name of this object.

int NumScalars { get; }

Read-only. If Datum holds array data, returns size of array. Returns 1 for simple types(int, double, string, etc.).

IComposite Owner { get; }

Read-only. Returns the owner of this object.

IComposite Parent { get; }

Read-only. Returns the Parent of this object.

IComposite ParentInspection { get; }

Read-only. Returns the parent Inspection Step.

IComposite ParentVisionSystem { get; }

Read-only. Returns the parent VisionSystemStep.

Datum ReferenceGet()

Valid for INPUT datums only, returns the Datum that is referenced by this Datum.

void ReferenceSet(IDatum refDatum)

Valid for INPUT Datums only, causes this Datum to point to, or reference, the Datum specified by the refDatum parameter.

int TaggedForUpload { get; }

Returns True if this Datum is tagged for upload.

string Type { get; }

Read-only. Returns the Type of this Datum. For example, an integer Datum would return the string “Datum.Int.1”, a Distance Datum would return the string “Datum.Distance.1”.

object Value { set; get; }

Gets/Sets the value of this datum. Refer to “Modifying Datum Values” on page 2-18 for a detailed description.

string ValueAsString { set; get; }

Allows you to get/set the value of the datum as a string. Some of the complex datum types do not implement this property and may return an empty string.

object ValueCalibrated { set; get; }

Use this property when you want to retrieve a Datum's value in calibrated units. The Job must have been calibrated first in order for this to have any effect. If the Job has not been calibrated, then this property returns the same data as the Value property.

EnumVisiblityFlags Visibility { set; get; }

Returns/sets Visibility Flags. Use this datum to hide/show datums. There are several valid options defined by the EnumVisibilityFlags type, but the only ones that really matter are those that follow:

– VF_NEVER — The datum is not visible. This means that the datum will not show up in the Datum page of the Setup Manager, or in the Datum Grid control. When applied to ROIs, this means that the ROI will not show up in the buffer and, therefore, cannot be moved by the user.

– VF_ALWAYS — The datum is always visible.

– VF_ADVANCED — The datum is advanced, and will only be shown in the DatumGrid control if the "Advanced" button on the toolbar is pressed.

int Visible { set; get; }

Allows easy manipulation of the Visibility flag when you simply want to show or hide a Datum. Set to 0 to hide a Datum, 1 to show it. Returns 0 if the Datum is not visible, non 0 if it is Visible.

Datum Object Methods

This section documents all of the methods of the Datum Object:

IComposite FindParent(**string** *stepType*)

Finds the Parent Step that matches the specified type. The stepType parameter is in the form "Step.Type". Please refer to "Finding Steps in the Step Tree" on page 2-8 for a more detailed description.

void Regen(**int** bDoPicture)

Regenerates the datum, and optionally takes a picture while doing so. "Regenerate" means to PreRun and then Run every Step that this Datum

is dependent upon. For example, a Point to Line Distance Step is dependent upon the Steps that provide the input point and input line, so those Steps would also be PreRun and Run.

int TagForUpload(int *bAdd*)

If bAdd is true, this Datum is added to the parent Inspection Step's list of Result to Upload. If bAdd is false, it is removed.

Step Handles: Converting to Step Objects

A Step Handle is a long integer that represents the address in memory (a pointer) of an underlying Step object that the Visionscape framework knows how to deal with. Several of the legacy Visionscape components deal primarily with Step Handles, rather than with Step Objects. Those components are the Runtime Manager, the original Setup Manager, Job Manager and Datum Manager. Each of these components have properties and methods that take Step Handles as inputs, and that will also return Step Handles to you. This leads to two questions:

- How do I pass a step handle to a method or property?

The Step Object provides a Handle property, so simply reference this property whenever you need to pass a Step Handle.

- How do I work with a step handle that is returned to me by a property or method?

The Visionscape.Steps namespace provides the StepHelper static class that can be used to convert Step Handles to Step Objects. In the following example, we demonstrate how the step handle returned by the original Setup Manager's GetCurrentStep function could be converted to a Step Object using the StepHelper.StepFromHandle() method:

```
//this legacy Setup manager function returns a step handle
int hStep = ctlSetup.GetCurrentStep();
if(hStep != 0)
{
//convert the handle to a Step Object using the StepHelper class
Step myStep = StepHelper.StepFromHandle(hStep);
Console.WriteLine("Selected Step is " + myStep.Name);
}
```

3

CHAPTER 3

Talking to Visionscape Hardware: VsCoordinator and VsDevice

Introduction to the Visionscape.Devices Namespace

In this chapter, we will talk about the objects provided by the Visionscape.Devices namespace. You will use these objects to query your PC and/or your network for Visionscape Devices.

Assembly Names: Visionscape.dll & Visionscape.Devices.dll

Namespace: Visionscape.Devices

To access this namespace, add the following .NET references to your project:

```
Visionscape
Visionscape.Devices
```

Add the following statement to the top of your C# files in order to make access to this namespace easier:

```
using Visionscape.Devices;
```

The two most important objects in this namespace are VsCoordinator, and VsDevice.

VsCoordinator

The primary purpose of the VsCoordinator object is to discover the Visionscape hardware that is available to your PC, and to collect that hardware into “Devices”. As we explained in Chapter 1, a “Device” represents a single smart camera or a collection of GigE cameras. VsCoordinator will discover all available smart cameras on your network, and create a VsDevice object to represent each. It will also discover all available GigE cameras and (by default) create one VsDevice object to talk to all of them. The discovered Devices are maintained in a list, and you will need to access this list in your code whenever you wish to communicate with a particular device. Access is provided via the Devices property, which is a collection of VsDevice objects.

You should also understand that VsCoordinator is a “Singleton” object, which means that there will only ever be one instance of it in a given process (EXE), regardless of how many times you instantiate the object in your project. For example, if you create instances of the VsCoordinator object in two different forms, both will actually be referencing the same instance of the object.

VsDevice

The VsDevice object is a programmer’s interface to any Visionscape device. Its API includes methods to start, stop, upload, download, take control, query status, and configure a device. It also contains properties that describe the device, such as the device type (GigE device, smart camera, etc), the digitizer type, etc. Therefore, the VsDevice object is central to writing your own Visionscape UI.

VsCoordinator and Device Discovery

When you instantiate a VsCoordinator object for the first time, it will immediately begin trying to detect what Visionscape Devices are present in your PC and on your network. This includes any Software Systems that you may have created. Because smart cameras and GigE cameras live on the network, they can not be instantly discovered, it may take from 2 to 10 seconds for all of your Devices to be discovered. Shortly we will discuss how to wait for Device discovery when your application starts up. As devices are discovered, GigE and Software Systems will always be

added to the Devices collection first, and smart cameras will always be added at the end of the list. The following example illustrates iterating through the VsCoordinator's Devices collection, and outputing the name of each Device to the output window:

```
VsCoordinator coord = new VsCoordinator();
foreach (VsDevice dev in coord.Devices)
{
    Console.WriteLine(dev.Name);
}
```

To find a particular device in the list by name, you can use the FindDeviceByName method. For example, the first GigE system discovered in your PC will always be assigned the name "GigeVision1", so you can find this device with the following code:

```
VsCoordinator coord = new VsCoordinator();
VsDevice gigeDev = coord.FindDeviceByName("GigeVision1");
```

Note: To see the names of the GigE systems and Software Systems in your PC, double-click on the Visionscape Backplane icon in the Windows System Tray.

This will open the AvpBackplane window, which will provide you with a list of all available GigE and Software Systems. But NOT smart cameras. To see a list of all discovered smart cameras, use the Visionscape Network Browser utility.

How Devices are Discovered

As mentioned previously, GigE cameras live on a network, and must be discovered before they can be accessed. The first time you instantiate your VsCoordinator object, it will in turn cause the Visionscape Backplane process to be started. You can see that the Visionscape Backplane has been started by looking for its icon in the Windows System Tray:

When the Visionscape Backplane process first starts it will try to detect what devices are present. This is handled as follows:

GigE Systems:

A command is sent out across your network connection asking all GigE cameras to announce themselves. If any GigE cameras are present, they will respond to this request. When the Visionscape Backplane process receives a response, it knows that it has GigE cameras, and will therefore go ahead and create a GigE System that will own each GigE camera. This system will be added to the front of the Devices Collection.

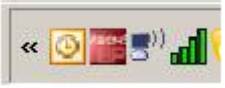
Software Systems:

Software Systems are not added to the Devices Collection until after the GigE System(s) have all been discovered. Software systems are always added after GigE systems, but before smart cameras.

Smart Cameras:

Cameras periodically announce their presence on the network. This is accomplished by broadcasting a short network message approximately every five seconds. This message is sent using UDP, which is a network protocol similar to TCP but much lighter weight and without the extensive error checking. The Visionscape Backplane process discovers smart cameras by listening for these UDP packets. When ever a UDP packet arrives from a smart camera, it checks to see if it is already in its list of Devices, and if not, it is added. So a smart camera could be discovered immediately, or it could take up to 10 seconds, it depends on when that first UDP packet arrives.

If you have previously instantiated a VsCoordinator in your application, then it's very likely that all Visionscape Devices will have already been discovered, meaning you won't have to wait for device discovery every time you create a VsCoordinator. If the Visionscape Backplane process is already running before you launch your application, then you also will not have to wait for Devices to be discovered:



Waiting for Device Discovery by Using “Device Focus”

So as we just discussed, when your application first starts, there may be a delay before all devices have been discovered and placed in the Devices collection of VsCoordinator. This means that your application will need to wait for its chosen Device to be discovered before it tries to access it. This is done by using the DeviceFocusSetOnDiscovery method of VsCoordinator. You simply pass in the name of the device you are looking for, and then the OnDeviceFocus event will be fired when that device is discovered. The following example demonstrates how we would wait for the discovery of the smart camera named “MyHawkEye_1600T”:

```
VsCoordinator m_coord;
private void frmMain_Load(object sender, EventArgs e)
{
m_coord = new VsCoordinator();
//wire up our event handler to the OnDeviceFocus event
m_coord.OnDeviceFocus += OnDeviceFocusEventHandler;
//tell coordinator to fire the OnDeviceFocus event when
//the device MyHawkEye_1600T is discovered
m_coord.DeviceFocusSetOnDiscovery("MyHawkEye_1600T", -1);
}
private void OnDeviceFocusEventHandler(VsDevice objDevice)
{
    Console.WriteLine("Our Device has been Discovered and is
Ready to use");
       //Continue your UI initialization here…
}
```

So as you can see, when our application starts up, we call the DeviceFocusSetOnDiscovery method in the Form_Load event, and then we wait for the OnDeviceFocus event to be fired. When OnDeviceFocusEventHandler is called, we know that the device “MyHawkEye_1600T” has been discovered and is ready to use. At that point, you would continue with whatever application initialization you need to perform, such as downloading a job, establishing report connections, etc.

Connecting Jobs to Visionscape Devices

At this point, you should understand that the VsDevice object represents a Vision System, which is one piece of, or a collection of, Visionscape

Hardware. You should also recognize that the VsCoordinator is intended to provide you with a list of all available VsDevices as soon as they have been discovered. In the previous chapter, we explained Jobs and Steps, and how to load AVP files from disk. Once a Job is loaded however, it can not function until it has been connected to a Device. In the following example, we demonstrate how to use the SelectSystem method of VisionSystemStep to connect the VisionSystem Step to the first GigE Device in the PC.

```
private JobStep _job = new JobStep();
private VsDevice _device;
private VsCoordinator _coordinator;
private void frmMain_Load(object sender, EventArgs e)
{
    //instantiate our coordinator object
    _coordinator = new VsCoordinator();
    //wire up our event handler to the OnDeviceFocus event
    _coordinator.OnDeviceFocus += OnDeviceFocusEventHandler;
    //tell coordinator to fire the OnDeviceFocus event when
    //the device GigEVision1 is discovered
    _coordinator.DeviceFocusSetOnDiscovery("GigEVision1", -1);
}
//This event will be received when GigEVision1 has been
//discovered
private void OnDeviceFocusEventHandler(VsDevice objDevice)
{
    _device = objDevice;
    //Load our job
    _job.Load("C:\Vscape\Jobs\_GigETest.avp");
    //Get the first VisionSystemStep
    VisionSystemStep vs = _job.VisionSystemStep();
    //Connect the VisionSystemStep to our device
    vs.SelectSystem(_device.Name);
    //Now we can run the inspections
    _device.StartAll();
}
//when running on PC based Devices,
//you must stop all inspections before exiting
private void frmMain_FormClosing(object sender,
FormClosingEventArgs e)
{
    _device.StopAll();
}
```

The previous example instructs the VsCoordinator to call our OnDeviceFocusEventHandler() function when the Device named "GigEVision1" is discovered. Once the Device is discovered, we load a Job from disk, connect the VisionSystemStep to the Device, and start all of the inspections running. We also make sure to call the Devices' StopAll method in the Form_Unload event to stop all inspections before the application exits. This is important when running with Host based Systems. A "Host based System" is one in which your PC acts as the primary vision processor, such as with GigE and Software Systems, and previously our framegrabber boards. On a Host based System, your application may crash if it attempts to shut down while inspections are still running. You do not need to worry about stopping inspections on exit when dealing with smart cameras. So, the previous example works well for Host based Systems.

But what about smart cameras? For these devices, it's not enough to simply select the hardware; you must also download your Job to the Device. In that case, the following code could be used. Assume we want to load a Job onto a smart camera named "MyHawkEye_1600T".

```
private JobStep m_Job = new JobStep();
private VsDevice m_dev;
private VsCoordinator m_coord;
//The Load event of our main form
private void frmMain_Load(object sender, EventArgs e)
{
    m_coord = new VsCoordinator();
    //wire up our event handler to the OnDeviceFocus event
    m_coord.OnDeviceFocus += OnDeviceFocusEventHandler;
    //tell coordinator to fire the OnDeviceFocus event when
    //the device MyHawkEye_1600T is discovered
    m_coord.DeviceFocusSetOnDiscovery("MyHawkEye_1600T", -1);
}

//This event handler will be called when our device is discovered
private void OnDeviceFocusEventHandler(VsDevice objDevice)
{
    //We've discovered our Device....
    m_dev = objDevice;
    //So load the job file
    m_Job.Load("C:\Vscape\Jobs\SampleJob.avp");
    try
    {   //download our job to the device, wait until it's done
        TransferStatus stat = m_dev.Download(m_Job, true);
```

```
            //check status of the download
            if(stat == TransferStatus.TransferOK)
           {   //download was successful, so start the inspections
                m_dev.StartAll();
            }
        }
        catch (Exception e)
        {
            Console.WriteLine("Download Failed for the following
Reason: " + e.Message);
        }
}
```

So in this example we waited for our smart camera to be discovered, and when it was, we loaded a Job, and then called the VsDevice.Download() method, passing in the Job step, and specifying 'true' for the bWait parameter, which means don't return until the download is complete. Next we simply check the return value to see if the download was successful, and if so, we start all inspections running on the device. We also added a try{} catch{} block to handle any exceptions thrown during the download. This is a simple and straightforward approach to downloading jobs to a smart camera.

So in looking at our previous two examples, we have lead you to believe that you'll need to run the first example when using a Host based Systems, and the second example when using smart cameras. However, this is not true, the second example works for all devices, including GigE and Software Systems. Although nothing needs to actually be downloaded to a Host based System (since the job is running locally on the PC), this call:

```
dev.Download(m_job, true);
```

is functionally equivalent to this call when using Host based Systems:

```
vs.SelectSystem dev.Name
```

In other words, the Download method will call the SelectSystem method for you. Thus, if you are trying to write a generic UI that will work with all Visionscape Devices, we recommend that you use the call to Download rather than SelectSystem.

One last point, you must remember that you need to stop all inspections from running when you exit your application if you are running on a Host System.

The previous examples showed, with a small amount of code, how to get a Visionscape Job loaded to a particular Device and running. We haven't showed you how to view the images and results yet, that's coming in the next chapter. But, how about an even easier way to get a Job loaded and running? Consider this modification to our previous example:

```
private VsDevice m_dev;
private VsCoordinator m_coord;
//The Load event of our main form
private void frmMain_Load(object sender, EventArgs e)
{
    m_coord = new VsCoordinator();
    //wire up our event handler to the OnDeviceFocus event
    m_coord.OnDeviceFocus += OnDeviceFocusEventHandler;
    //tell coordinator to fire the OnDeviceFocus event when
    //  the device MyHawkEye_1600T is discovered
    m_coord.DeviceFocusSetOnDiscovery("MyHawkEye_1600T", -
1);
}
//This event handler will be called when our device is
discovered
private void OnDeviceFocusEventHandler(VsDevice objDevice)
{
    //We've discovered our Device....
    m_dev = objDevice;
    try
    {   //Load the AVP and download it in one step, wait
until its done
        TransferStatus stat =
 m_dev.DownloadAVP("C:\Vscape\Jobs\SampleJob.avp",
true);
        //check status of the download
        if(stat == TransferStatus.TransferOK)
        {   //download was successful, so start the
inspections
            m_dev.StartAll();
        }
    }
    catch (Exception e)
    {
       Console.WriteLine("Download Failed for the following
Reason: " + e.Message);
    }
}
```

In this example, we eliminated the JobStep, and used the VsDevice object's DownloadAVP method to directly load and then download our AVP file in one shot. This method works with all Visionscape Devices. If the goal of your UI is to simply load an AVP from disk and get it running, the above sample is the simplest way to do it. If your goal is to write a more complex UI that provides the user with access to the Steps and Datums of the AVP, or if you need to scan the Steps of the Job to gather information about it before you start running, then you will need to load the Job into a JobStep first, as demonstrated in our previous example. The choice is yours.

What Else Can I Do With Device Objects?

Table 3–1 Acts as a quick reference of example code for common situations. For important additional details, please refer to the detailed documentation later on in this chapter.

TABLE 3–1. How Do I...

How do I...	Example

Enumerate Devices	1. Declare and create a VsCoordinator
	`VsCoordinator coord = new VsCoordinator();`
	2. Iterate over the Devices collection
	`VsCoordinator coord = new VsCoordinator();`
	`foreach (VsDevice dev in coord.Devices)`
	`{`
	`  Console.WriteLine(dev.Name);`
	`}`

Discover a smart camera when my application first starts	1. Declare member variables for VsCoordinator and VsDevice:
	`private VsCoordinator m_coord = new VsCoordinator();`
	`private VsDevice m_dev;`
	2. In your Form_Load event, connect an event handler to the OnDeviceFocus event, and call the DeviceFocusSetOnDiscovery method, passing in the name of your smart camera:
	`m_coord.OnDeviceFocus += OnDeviceFocusHandler;`
	`m_coord.DeviceFocusSetOnDiscovery("MyCam",-1);`
	3. Handle the OnDeviceFocus event, and add your initialization code to the event handler
	`private void OnDeviceFocusEvent(`
	`VsDevice objDevice)`
	`{`
	`  //finish initializing your app here…`
	`}`

Take Control of a Device	1. Call the TakeControl method of VsDevice:
	`bool InControl = m_dev.TakeControl("UserID","Pwd");`
	2. Check the return value to see if it succeeded
	`if(InControl)`
	`{`
	`  //safe to download, start, stop, etc.`
	`}`

Start/Stop Inspections	`dev.StartAll();//Starts all inspections`
	`dev.StartInspection(0);//starts 1st inspection`
	`//start 2nd insp, run for 3 cycles`

TABLE 3–1. How Do I...

Download an AVP file	`dev.StartInspection(1, 3);`
	`dev.StopAll();//stop all inspections`
	`dev.StopInspection(0); //stop inspection 1`

TABLE 3–1. How Do I...

Download a Job Step that is already loaded	1. Get a VsDevice reference for the Visionscape Device you wish to download to.
	2. Use the DownloadAVP method of VsDevice
	`TransferStatus stat = m_dev.DownloadAVP("SampleJob.avp", false);`
	3. Check the transfer status in a loop while calling DoEvents to allow other UI events to continue while waiting for the download to finish
	`while(m_dev.CheckXferStatus(20) == tagXFERSTATUS.XFER_IN_PROGRESS)`
	`{`
	`  Application.DoEvents();`
	`}`
	4. Step 3 can be skipped by passing 'true' as the 2nd parameter of Download. For more information about downloading a Job, see "Downloading a Job" on page 3-14

Get information on the running Job	1. Tell Device to update its Namespace information by calling QueryNamespace.
	`m_dev.QueryNamespace();`
	2. Access the NameSpace property, which holds a VsNameNode object with child VsNameNode objects arranged in a tree structure that is identical to the Loaded Job.
	`VsNameNode nnJob = m_dev.Namespace;`

Getting information on a Device's IO capabilities	1. Tell device to update its IO information by calling the QueryIOCaps method.
	`m_dev.QueryIOCaps();`
	2. Access the IOCaps property. This is a VsIOCaps object, the properties of which describe the IO capabilities of the device.
	`VsIOCaps iocaps = m_dev.IOCaps;`
	`Console.WriteLine("Num Physcial IO Pts: " + iocaps.CountPhysical);`

TABLE 3–1. How Do I...

Determine a Device's Type	1. Query the DeviceClass property.
	2. If you wanted to make sure a VsDevice object was referencing a smart camera, do the following:
	`if(m_dev.DeviceClass == tagDEVCLASS.DEVCLASS_SMART_CAMERA)`
	`{`
	`//perform a Smart-Camera-only action`
	`}`
	3. If you wanted to make sure a VsDevice object was referencing a GigE system…
	`if(m_dev.DeviceClass == tagDEVCLASS.DEVCLASS_GIGE_VISION_SYSTEM)`
	`{`
	`//perform a GigE-only action`
	`}`

A Detailed Look at VsDevice

VsDevice is the primary interface when communicating to any Visionscape device. The primary feature of this object is to provide a common programming interface, regardless of whether the target device is a smart camera, a GigE System, a vision board, or software emulated. You should never create a “new” VsDevice object; instead, you should retrieve a reference to a VsDevice through one of the VsCoordinator methods as described previously. Following is a more detailed description of the various capabilities provided by VsDevice.

Device Control Functions

VsDevice provides several methods to control the state of your device:

Taking Control / Releasing Control of a Smart Camera

When dealing with smart cameras, your user interface should “Take Control” of these devices before you perform any operation that may affect their behavior. Although it’s not required that you take control of a device before downloading to it or starting and stopping inspections, we strongly recommended that you do. This insures that your application does not conflict with another user on another PC who may currently be connected to the same smart camera (via FrontRunner or AppRunner, perhaps). The TakeControl method requires a user id and password:

(NOTE: The TakeControl method is relevant only to smart cameras, and does not apply to other Device types).

```
bool InControl = m_dev.TakeControl("UserID", "Pwd");
if(InControl)
{
    //safe to download, start, stop, etc
}
```

You can check whether you already have control by using the VsDevice HaveControl property:

```
if(m_dev.HaveControl)
{
       //we already have control
}
```

Release Control by using the ReleaseControl method of VsDevice:

```
m_dev.ReleaseControl();
```

Start / Stop Inspections

It's easy to start and stop all of the inspections on a given Device, you simply call the StartAll and StopAll methods. To start a specific inspection, you call the StartInspection and StopInspection methods:

```
dev.StartAll();//Starts all inspections
dev.StartInspection(0);//starts 1st inspection
//start 2nd insp, run for 3 cycles
dev.StartInspection(1, 3);
dev.StopAll();//stop all inspections
dev.StopInspection(0); //stop 1st inspection
```

Downloading a Job

You must download a VisionSystemStep to a device, as the VisionSystemStep represents the vision program for one device. To download a vision job to a device, you can either call Download, or you can call DownloadAVP. The Download method will download the VisionSystemStep that you pass to it. You can either pass it a VisionSystemStep directly, or you can pass it a reference to your JobStep, in which case it will find the first VisionSystemStep for you and download it. The DownloadAVP method takes the file name of an .AVP file, opens it for you, and then downloads. For each method you can choose to wait until the download is complete by passing true as the second parameter:

```
//Download the first VisionSystemStep in the Job like this
TransferStatus stat = m_dev.Download(m_Job, true);
//Download the second VisionSystemStep in the Job like this:
_device.Download(_job.VisionSystemStep(1), true);
//Download an AVP file like this
TransferStatus stat =
m_dev.DownloadAVP("SampleJob.avp",true);
```

Each method returns a TransferStatus enum value which will notify you of the success or failure of the download. The TransferStatus values are as follows:

```
TransferStatus.TransferOK  //Successful download
TransferStatus.TransferNotStarted //Failed to start the
//download
TransferStatus.TransferError//An error occurred //during
transfer
```

It is also possible for an exception to be thrown during download, so you should always enclose your Download calls in try{}catch{} blocks.

Typical reasons for download exceptions:

- The version of the firmware on your smart camera does not match the version on your PC. Download is not allowed between mismatched versions.
- Your Job fails PreRun, and is not ready to run. This often happens when trying to load a Job that was created on a different type of device. You should load the Job into FrontRunner and be sure that it runs there before trying to load it from your application.
- Trying to download a Job that contains IntelliFind® Steps to a smart camera that does not have an IntelliFind® License.

If you wish to perform an asynchronous download, in which the download is started and then control is returned to you immediately, then pass false as the second parameter. The following example demonstrates how you would start an asynchronous download of the first VisionSystemStep in the JobStep named m_Job, and then wait for the download to complete:

```
TransferStatus stat = m_dev.Download(m_Job, false);
while (m_dev.CheckXferStatus(20) ==
tagXFERSTATUS.XFER_IN_PROGRESS)
{
```

```
        Application.DoEvents();
}
```

The parameter to CheckXferStatus is the number of milliseconds to sleep (and, therefore, not tie up CPU resources) while waiting to see if the download is complete. The possible return values for CheckXferStatus are:

- XFER_IN_PROGRESS — Still transferring the job
- XFER_DONE — Finished with the transfer
- XFER_ERROR — Transfer was unsuccessful

As the transfer progresses, the OnXferProgress event is raised by the VsDevice object to report progress messages and state or error information. You can use this information to either update a progress bar, or you can simply display the messages for the user. Following is an example of how to respond to the OnXferProgress event.

```
//Wire up the event handler and start the download
    m_dev.OnXferProgress += m_dev_OnXferProgress;
    TransferStatus stat = m_dev.Download(m_Job, false);
}
//this event will be fired as the download progresses
void m_dev_OnXferProgress(int nState, int nStatus, string
msg)
{
    Console.WriteLine("Download Progress:");
    Console.WriteLine("   State: " + nState);
    Console.WriteLine("   Status: " + nStatus);
    Console.WriteLine("   Message: " + msg);
}
```

The parameters passed along with the OnXferProgress event are:

- nState — This Long value represents the overall state of the transfer. The following values are possible:
  - -1 — Transfer error
  - 0 — Transfer started
  - 1 — Transfer complete
  - 3 — Transfer message

- nStatus — Provides data relative to the current nState value. If nState indicates an error condition, then this Long value holds the error code. For all other states, nStatus holds a value from 1 to 100 that indicates the current completion percentage of the download. So, this value can be used to update a progress bar.
- msg — This string value describes the current status of the download. You may prefer to simply printout these messages for your user rather than implementing a progress bar.

Uploading a Job

Using the Upload method of VsDevice is slightly more involved, as we will be receiving a new VisionSystemStep, and must prepare the job to receive it. You will get the new VisionSystemStep from the OnUploadComplete event, as shown in the following example:

```
//declare a global VisionSystemStep because we will receive
//this Step
//in an event and need to stash it somewhere
private VisionSystemStep m_UploadedVSStep;
private void UploadJob()
{
    //delete the contents of the existing job 1st
    while(m_Job.Count > 0)
    {
        m_Job.Remove(1);
    }
    //wire up our event handler,
    //  to notify us when the upload is complete
    m_dev.OnUploadComplete += m_dev_OnUploadComplete;
    //start the asynchronous upload
    m_dev.Upload(m_Job);
    //waiting for an upload to complete is the same as for a
    //download
    while(m_dev.CheckXferStatus(20) ==
tagXFERSTATUS.XFER_IN_PROGRESS)
    {
        Application.DoEvents();
    }
    //upload is complete, do we have a Vision System Step?
    if(m_UploadedVSStep != null)
    {
        //Yes, we successfully received the job
    }
```

```
}
//This event is fired when the Upload is complete, and it
//will return
//the uploaded VisionSystem step to you
void m_dev_OnUploadComplete(int nStatus, VisionSystemStep
pVS)
{
    if(nStatus == 0)
    {    m_UploadedVSStep = pVS;
    }
    else
        m_UploadedVSStep = null;
}
```

Obtaining Device Information

VsDevice has many properties and methods that provide valuable information about the current device.

Basic Device Information

VsDevice provides a wealth of information about a given Vision System. For example, this code lists basic information from each of the devices in the VsCoordinator Devices list:

```
foreach(VsDevice dev in m_coord.Devices)
{
    Console.WriteLine("Name = " + dev.Name);
    Console.WriteLine("Device Class = " + dev.DeviceClass);
    Console.WriteLine("Model = " + dev.DeviceModel);
    Console.WriteLine("Digitizer = " + dev.DigitizerModel);
    Console.WriteLine("IP Address = " + dev.IPAddress);
    Console.WriteLine("MAC Address = " + dev.MACAddress);
    Console.WriteLine("Software Rev = " +
dev.SoftwareVersion);
    Console.WriteLine("Controlled By " +
dev.NameOfController);
    Console.WriteLine("Num Inspections = " +
dev.NumInspections);
}
```

DeviceClass Property

The DeviceClass field identifies what type of Device you are dealing with. It returns an enumeration value of type tagDEVCLASS. The values of the enumeration are as follows:

```
DEVCLASS_UNKNOWN=0
DEVCLASS_SOFTWARE_EMULATED=1
DEVCLASS_HOST_BOARD=2
DEVCLASS_PROCESSOR_BOARD=3
DEVCLASS_SMART_CAMERA=4
DEVCLASS_SMART_CAMERA_OLDER=5
DEVCLASS_SMART_CAMERA_UNREACHABLE=6
DEVCLASS_CAMERA=7
DEVCLASS_GIGE_VISION_SYSTEM= 8
```

The most important values are:

- DEVCLASS_SMART_CAMERA — The Vision HAWK Smart Camera, for example.
- DEVCLASS_GIGE_VISION_SYSTEM --- A GigE Vision system.
- DEVCLASS_SOFTWARE_EMULATED — A software system with no hardware.
- DEVCLASS_SMART_CAMERA_UNREACHABLE — A smart camera that, due to network topology, cannot be connected to via a TCP connection. Perhaps it's on a different subnet or has an incompatible IP address.

IsHostBased Property

You can use the IsHostBased property to determine if there is no target processor for the device; in other words, if the device is a GigE System, host based board or software emulated.

Determining if any Inspections are Running

```
if(m_dev.IsAnyInspectionRunning)
{
    m_dev.StopAll();
}
```

Determining if a Particular Inspection is Running

This is easily accomplished via the IsInspectionRunning method.

```
bool IsInspectionRunning(nInsp As Long)
```

nInsp — 0 based index of the inspection. If you pass in -1, the function check all inspections to see if ANY are running.

Device States

It is often valuable to find out what state a device is in. This information is obtained via the DeviceState property of VsDevice. This property returns an enumerated value of type tagDEVSTATE. The enumeration values are:

```
DEVSTATE_UNKNOWN=0
DEVSTATE_RUNNING=1
DEVSTATE_STOPPED=2
DEVSTATE_NOJOB=3
DEVSTATE_NOCOMM=4
DEVSTATE_ERROR=100
DEVSTATE_FILE_XFER=101
DEVSTATE_TRYOUT=102
DEVSTATE_EDIT=103
DEVSTATE_LIVE=104
DEVSTATE_FUNCTION=105
DEVSTATE_ACQUIRE=106
```

The most important of these are:

- DEVSTATE_NOJOB — No job has yet been loaded on the device
- DEVSTATE_RUNNING — If any inspection is running
- DEVSTATE_STOPPED — All inspections are stopped
- DEVSTATE_NOCOMM — UDP info has not been received for a long time, so it is assumed communications have been lost.

Special Device States

The other states are not commonly used in most user applications, but may be useful in certain situations. There are some shorthand properties of VsDevice that reflect states that may be useful:

```
dev.IsInLive - the device is acquiring live images
dev.IsInTryout - the program associated with the device is
in
tryout mode on the PC
dev.IsInAcquire - the device is acquiring a single image
```

Whenever the device state changes, the OnDeviceStateChanged event is raised by VsDevice. Normally, you would handle this event to be notified if the device has been stopped/started or entered an error condition. There is also the OnDeviceStateChanging event that is raised before the device state is actually changed.

Determining the I/O Capabilities of a Device

You can determine the I/O capabilities of a device by calling the QueryIOCaps method, which returns an object of type VsioCaps. The properties of VsioCaps describe the hardware configuration:

```
VsioCaps iocaps = m_dev.QueryioCaps();
if(iocaps != null)
{
    Console.WriteLine("# Physical = " +
iocaps.CountPhysical);
    Console.WriteLine( "# AnalogOut = " +
iocaps.CountAnalogOut);
    Console.WriteLine( "# PhysicalIn = " +
iocaps.CountPhysicalIn);
    Console.WriteLine( "# PhysicalOut = " +
iocaps.CountPhysicalOut);
    Console.WriteLine( "# RS422Input = " +
iocaps.CountRS422Input);
    Console.WriteLine( "# RS422Output = " +
iocaps.CountRS422Output);
    Console.WriteLine( "# Sensor = " + iocaps.CountSensor);
    Console.WriteLine( "# SlaveSensor = " +
iocaps.CountSlaveSensor);
    Console.WriteLine( "# Strobe = " + iocaps.CountStrobe);
    Console.WriteLine( "# TTLInput = " +
iocaps.CountTTLInput);
    Console.WriteLine( "# TTLOutput = " +
iocaps.CountTTLOutput);
    Console.WriteLine( "# Virtual = " + iocaps.CountVirtual);
    Console.WriteLine( "Mask Current= " +
iocaps.MaskCurrent);
    Console.WriteLine( "Mask GPIO = " + iocaps.MaskGPIO);
```

```
    Console.WriteLine("MAX NER Axis = " +
iocaps.MaxNERLightAxis);
}
```

UDPInfo Available for Networked Devices

For networked devices such as a smart camera, we recommend that you first read the section on UDPInfo (see “Networked Device Discovery and UDPInfo” on page 3-3). As mentioned, networked devices transmit a packet via the UDP network protocol about every five seconds. This packet contains much useful information and can be used to great advantage by a programmer. Its information is accessed via the UDPInfo property of VsDevice.

Because UDPInfo is only available for networked devices, it’s a good idea to always check if the object exists before using it:

```
Visionscape.Internals.VsUDPInfo udp;
udp = m_dev.UDPInfo;
if(udp != null)
{
    Console.WriteLine("Seconds Since Last Updated " +
                           m_dev.TimeSinceLastRefresh);
    Console.WriteLine( "Cycle Count 1st inspection " +
                                        udp.CountCycles1);
    Console.WriteLine( "Cycle Count 2st inspection " +
                                        udp.CountCycles2);
    Console.WriteLine( "Passed Count 1st inspection " +
                                        udp.CountPassed1);
    Console.WriteLine( "Passed Count 2st inspection " +
                                        udp.CountPassed2);
    Console.WriteLine( "Name of First Inspection " +
                              udp.FirstInspectionName);
    Console.WriteLine( "Use DHCP " + udp.NetDHCP);
    Console.WriteLine( "Net Mask " + udp.NetMask);
    Console.WriteLine( "Controlling PC " +
udp.IPAddressOfController);
    Console.WriteLine( "AVP Name of loaded program " +
                                        udp.ProgramName);
    Console.WriteLine( "Software Version " +
udp.SoftwareVersion);
    Console.WriteLine( "Number of Inspections " +
udp.NumInspections);
    Console.WriteLine("Number of Network Connections " +
udp.NumConnections);
```

```
}
```

In the preceding example, notice that the first Debug.Print statement prints the number of seconds since this data has last been updated with new UDP packet info from the device. This is done using the VsDevice TimeSinceLastRefresh property. You can also handle the OnUDPInfoChanged event, which will be raised whenever a new UDP packet arrives.

Retrieving Basic information on the Loaded Job

The following example shows the use of a few VsDevice properties in order to dump out some simple information about the job that is currently loaded:

```
Console.WriteLine("Number of inspections loaded" +
dev.NumInspections);
Console.WriteLine("Are any inspections running?" +
dev.IsAnyInspectionRunning);
Console.WriteLine("Is 1st Inspection running? " +
dev.IsInspectionRunning(0));
```

These methods should not be called unnecessarily, since they could involve a network transaction with the device and, therefore, may impact performance. This is not the case with the information obtained via UDPInfo described above.

Namespace Information

VsDevice allows you to query the namespace of a device in order to extract information about the Job that it's currently running. This is particularly useful when dealing with smart cameras in a user interface where you do not have the Job loaded locally. By calling the QueryNamespace method of VsDevice, you will cause a command to be sent to the device, which will cause it to construct a description of every Step in the Job. This description will be sent back to the VsDevice object, and it will construct a tree of VsNameNode objects that mimics the Job on the Device.

```
dev.QueryNamespace();
```

After this call, you can use various methods and properties to access the namespace data. As we mentioned above, this functionality is primarily used in applications that will deal with smart cameras, but it works with

ALL devices. If you already have the Job loaded in memory (in a JobStep), it's more efficient to analyze the actual Job rather than dealing with Namespaces. The following are the methods and properties used to access the Namespace.

VsNameNodeCollection ListInspections()

Provides you with a list of the running inspections on the device. This is in the form of a VsNameNodeCollection object, which is a collection of VsNameNodes. There will be one VsNameNode for each inspection step in the Job. The VsNameNodes will provide you with useful information that describes the Inspection Steps. Refer to the following section for more info on the VsNameNode object.

VsNameNodeCollection ListSnapshots(VsNameNode *nnInsp*)

Provides a list of Snapshots that live under the specified Inspection. You must pass in a VsNameNode object that represents an Inspection Step, and it will return you a list of VsNameNodes, one for every Snapshot Step under the specified Inspection. The following is an example of how you might walk through all of the Inspections and Snapshots of a remote device:

```
private void AnalyzeNamespace()
{
    //tell the device to send us updated namespace
    //information
    m_dev.QueryNamespace();
    //if no namespace information, just exit
    if(m_dev.Namespace == null)
        return;
    //get a list of namenodes for each inspection step
    VsNameNodeCollection nnAllInsp =
m_dev.ListInspections();
    //iterate through the inspection namenodes
    foreach(VsNameNode nnInsp in nnAllInsp)
    {   //dump some information on the inspection step
        Console.WriteLine("Inspection Info: " +
nnInsp.UserName + ", "
 + nnInsp.NameType);
        //get a list of all snapshots under this inspection
        VsNameNodeCollection nnAllSnaps =
m_dev.ListSnapshots(nnInsp);
        foreach (VsNameNode nnSnap in nnAllSnaps)
        {
```

```
            //snapshot step's user name
            Console.WriteLine("Snapshot Name: " +
nnSnap.UserName);
            //symbolic name
            Console.WriteLine("Symbolic Name: " +
nnSnap.SymbolicName);
            //step type
           Console.WriteLine("Step Type: " + nnSnap.ProgID);
        }
    }
}
```

VsNameNode Namespace { get; }

This read-only property returns you a reference to the VsNameNode that represents the VisionSystem Step that is running on the Device. A VsNameNode object, just like a Step object, is a collection. Each VsNameNode holds a list of all the child Steps and Datums that live under the actual Step that it represents. This means that you can walk through the children of the VsNameNode returned by the Namespace property, and analyze it in much the same way you would an actual loaded Job. Refer to the following section on the VsNameNode object for more information.

VsNameNode

The VsNameNode object represents either a Step or Datum object. It's the object used to provide Namespace information for a Device, typically a remote Device like a smart camera. When we say Namespace information, we are primarily talking about the Job that is loaded on the Device. You will access a Device's Namespace via the VsDevice object's Namespace property, or the ListInspections and ListSnapshots functions (refer to the previous section for more information). The following example shows how you would access the Namespace of a given Device:

```
m_dev.QueryNamespace();
VsNameNode nnJob = m_dev.Namespace;
```

VsDevice's Namespace property returns a VsNameNode object that represents the VisionSystem Step of the Job on the Device. When we say it "represents" the Step, we mean it's an object that contains information that describes that Step, it's not the actual Step. Every VsNameNode object is also a collection. It holds a collection of VsNameNode objects that describe all of the child Steps AND Datums of the Step that it

represents. This means that you can walk through the elements of the VsNameNode object, just as you can walk through the elements of a Job tree. The next most obvious question is, what information does the VsNameNode object provide?

VsNameNode Properties

As we've already said, a VsNameNode represents either a Step or a Datum. Therefore, the properties are intended to describe that Step or Datum.

tagNAMENODE_TYPE NameType { set; get; }

This property returns a value that identifies the type of object represented by the NameNode. Possible values are:

- NAMETYPE_DATUM — The NameNode represents a Datum object.
- NAMETYPE_STEP — The NameNode represents a Step object.
- NAMETYPE_FIELD — The NameNode represents a field.

tagNAMENODE_CAT NameCat { set; get; }

Returns a value that indicates the category of the Step or Datum. This is roughly equivalent to the Category property of Step and Datum. Possible values for Datums:

- VS_INPUT_DATUM
- VS_OUTPUT_DATUM
- VS_RESOURCE_DATUM

Possible values for Steps:

- VS_POSTPROC_STEP
- VS_PREPROC_STEP
- VS_PRIVATE_STEP
- VS_SETUP_STEP
- VS_PART_STEP

string ProgID { set; get; }

Returns a string that is equivalent to the Step and Datum object's Type property. This is in the form "Step.type.1" or "Datum.type.1" where "type" would represent the actual type of the Step or Datum.

string SymbolicName { set; get; }

The symbolic name of the Step or Datum. This is equivalent to the NameSym property of Step and Datum. Return value is a string.

int TaggedForUpload { get; }

Only applies to Datums. Returns 1 if this Datum has been selected for upload, 0 if not. In other words, this Datum was added to the Inspection Step's "Select Results to Upload" list and will, therefore, show up in the list of results in each inspection report.

string UserName { set; get; }

The user assigned name of the Step or Datum. This is equivalent to the Name property of Step and Datum. Return value is a string.

string GUID { set; get; }

A string that represents the actual GUID of the object

int Handle { set; get; }

This is functionally equivalent to the Handle property of the Step and Datum objects.

VsNameNode Inspection { get; }

Returns a reference to a VsNameNode that represents the parent Inspection Step.

VsNameNode Parent { set; get; }

Returns a reference to the parent VsNameNode.

int Count { get; }

Returns a count of how many child VsNameNodes are in your collection.

VsDevice Device { set; get; }

Returns a reference to the VsDevice object that produced this VsNameNode.

VsNameNode Methods

VsNameNodeCollection SearchForType(string *bstrType*)

Allows you to search for all of the child nodes that are of the type specified by the bstrType parameter. A VsNameNodeCollection is returned that contains all of the nodes found.

```
//get a list of all fast edge steps under this namenode
VsNameNodeCollection allFastEdge =
nnSnap.SearchForType("Step.EdgeFast.1");
//iterate the list of fast edge steps
foreach(VsNameNode nnFastEdge in allFastEdge)
{
    Console.WriteLine("Fast Edge Step Name: " +
nnFastEdge.UserName);
}
```

VsNameNode FindParentOfType(string *ProgID*)

Allows you to search for a parent Step or Datum that is of the type specified by the ProgId parameter. A VsNameNode reference is returned.

```
//find the parent inspection of this namenode
VsNameNode parentInsp =
nnSnap.FindParentOfType("Step.Inspection.1");
```

string MakePath(string *StopAtType*)

Builds a path string from the current node up to the first parent node that is of the type specified by the StopAtType parameter. If StopAtType is an empty string, the routine defaults to the Inspection step.

VsNameNodeCollection SearchForTagged(string *strTypeIn*)

Returns a collection of name nodes that are selected for upload. This only applies to Datums, so only VsNameNodes that represent Datums will be in the list. The strTypeIn parameter can be used when you only want your

list to include datums of a certain type, pass an empty string to return all Datum types.

```
VsNameNode nnJob = m_dev.Namespace;
//get a list of all datums selected for upload
VsNameNodeCollection uploadList = nnJob.SearchForTagged("");
//get a list of only Point List Datums that are selected for Upload
uploadList = nnJob.SearchForTagged("Datum.PtList.1");
```

A Detailed Look at VsCoordinator

In general, you will use VsCoordinator to provide access to the list of available Devices. VsCoordinator does have other advanced uses however. In this section, we will take a detailed look at all of VsCoordinator's capabilities.

Device Collection

All devices are accessible via the Devices collection property of VsCoordinator. You can iterate over accessible devices using code such as:

```
VsCoordinator m_coord = new VsCoordinator();
foreach(VsDevice dev in m_coord.Devices)
{
        Console.WriteLine("Name = " + dev.Name);
}
```

DeviceFocusSet

VsCoordinator maintains a list of all available Devices, but you can specify one of those Devices to be the "Focus" device. This is done by calling the DeviceFocusSet method, which will cause the OnDeviceFocus event to be fired. This is an advanced method that you can use in your application to allow multiple forms and controls to all synchronize themselves to a given Device. If each form and control in your application has it's own instance of a VsCoordinator object, then you could call the DeviceFocusSet method in one place in your code, and then the OnDeviceFocusEvent would be fired in each of those VsCoordinator instances. You could then add logic to each of your forms and controls to respond to the event, and automatically update themselves whenever the selected Device changes. This is how the Device focus is set:

```
//dev is a VsDevice object that we want
//to be the "focus" device
m_coord.DeviceFocusSet(m_dev, -1);
```

The second parameter is a group index, which can be used in cases where you want to have some of your forms and controls “focused” on different devices. The group index is passed back in the OnDeviceFocus event. A group index of -1 is typically used when you don’t wish to use this feature.

- There is also a OnDeviceFocusChanging event that is fired before the actual OnDeviceFocusChanged event, which is useful if there is some processing to do before any other control handles the OnDeviceFocus event.

To clear the device focus, pass null to signify that no device is selected:

```
m_coord.DeviceFocusSet(null, -1);
```

You can retrieve the device with the focus at any time by calling the DeviceFocusGet method. Be sure to always check if the device is valid, by always writing code such as:

```
VsDevice dev = m_coord.DeviceFocusGet(-1);
if(dev != null)
{
    //do something
}
```

The parameter passed to DeviceFocusGet is again a group ID, set this to -1 if you are not using a group ID.

Device Focus Property

It’s also possible to use a shorthand method to set and get the device focus in situations where a GroupID is not necessary. You can simply use the DeviceFocus property of VsCoordinator:

```
//set the focus device like this
m_coord.DeviceFocus = m_dev;
//get the focus device like this
VsDevice dev = m_coord.DeviceFocus;
```

DeviceFocusSetOnDiscovery

If you know the name of the Device that you want to set the focus to, but you don't have the device object, you can use the DeviceFocusSetOnDiscovery method. Earlier in this chapter we discussed how to use this method to wait for your smart camera to be discovered when your application is first starting up. For example, to set the device focus to "MyCamera":

```
m_coord.DeviceFocusSetOnDiscovery("MyCamera", -1);
```

This will cause the OnDeviceFocus event to be sent when MyCamera is discovered. This will occur immediately if MyCamera has already been discovered, or after a delay of up to five seconds if no UDP packets have yet been received from that Device.

Finding a Device by Name or IP

You can also look up a device in the coord.Devices list quickly by using the FindDeviceByName method. A null value is returned if the device can not be found. For example:

```
VsDevice dev = m_coord.FindDeviceByName("MyHawkEye_1600T");
```

If you do not know the name of the device, but know the IP Address, use the FindDeviceByIP method. For example:

```
VsDevice dev = m_coord. FindDeviceByIP("10.2.1.198");
```

You can also get the IP address of a device with a given name by using the LookupIPAddress function.

```
string devIP = m_coord.LookupIPAddress("MyHawkEye_1600T");
```

OnDeviceDiscovered Event

The OnDeviceDiscovered event is fired whenever a new device has been found. This event is useful when you are trying to display a list of current devices to the user.

Using Message Broadcasting to Simplify Application Design

There are some functions of VsCoordinator that help with organizing a project in which information needs to be shared between different

components. The most useful of these is the Broadcast mechanism. By using the methods BroadcastMessage or BroadcastObj, you can cause the OnBroadcastMessage or OnBroadcastObj events to be raised for all other VsCoordinators. For example, let's say you have a button on a form, and when you press it you want all other forms to receive an event. You can do this as follows:

```
private void MyButton_Click(object sender, EventArgs e)
{
    m_coord.BroadcastMessage(this.Name, "MyPrivateMessage",
"param");
}
```

- The first parameter is a name that identifies the originator of the message. In this example, we are using the form name that is accessed using this.Name. If this code was in a user control, you would use UserControl.Name instead. You will see why the name is important in a moment.
- The second parameter is a string that represents the actual message to be broadcast.
- The third parameter is optional; you can use it to add additional parameters to the message.

The BroadcastMessage function will cause the OnBroadcastMessage event to fire for every VsCoordinator in the application. For example, if another form was interested in this message, you would implement the OnBroadcastMessage event as follows (assume we our instance of VsCoordinator is named m_coord):

```
//wire up the event handler
m_coord.OnBroadcastMessage += m_coord_OnBroadcastMessage;

void m_coord_OnBroadcastMessage(string bstrSender,
 string bstrMsg, string bstrParam)
{
    if(bstrSender != this.Name)
    {
        if(bstrMsg == "MyPrivateMessage")
        {
            //do something
        }
    }
}
```

Notice the first statement that exits if the bstrSender parameter is the same as the name of the form. This is simply a way of checking if the broadcast came from this form or from somewhere else. This may not be necessary depending on what you are doing, but it's often the case that the originator of a message does not want to handle the message themselves. After that you would simply enter an 'if' or 'switch' statement to check if the message is something you want to process.

A variation of the broadcast message allows you to send an object as the parameter. For example:

```
private void MyButton_Click(object sender, EventArgs e)
{
    object someObject = new MySpecialObject();
    m_coord.BroadcastObj(this.Name, "MyPrivateMessage",
someObject);
}
```

And the event handler:

```
void m_coord_OnBroadcastObj(string strSender, string
strMsg,object obj)
{
    switch(strMsg)
    {
        case "MyPrivateMessage":
            //do something
            break;
    }
}
```

UpdateUI Method

Calling the UpdateUI of VsCoordinator causes the OnUpdateUI event to be raised for every VsCoordinator reference. You can use this method as a way to inform every form or control that something has changed that requires a display update. For more complex projects, the BroadcastMessage approach is preferred because you can define your own messages.

LogMessage and the Debug Window

As an aid to debugging, you can use the Visionscape debug window (which is accessed via the AvpSvr taskbar application) to log messages, if desired, by using the LogMesssage method:

```
m_coord.LogMessage("Something important happened!", false);
```

The second parameter should be set to True if you are reporting an error condition (and will appear red in the display), and False if the message is for information purposes only. You can show or hide the debug window using the function ShowLogWindow:

```
m_coord.ShowLogWindow(true); // to show the display
m_coord.ShowLogWindow(false); // to hide the display
```

Getting Information About Local Network Interface Controllers

You can retrieve the state of any local Network Interface Controller devices using the NetworkAdapters property of VsCoordinator. You can use a VsNetworkAdapter to traverse this collection:

```
foreach(VsNetworkAdapter nic in m_coord.NetworkAdapters)
{
    Console.WriteLine("Description = " + nic.Caption);
    Console.WriteLine("Gateway = " + nic.DefaultIPGateway);
    Console.WriteLine("DHCP = " + nic.DHCPEnabled);
    Console.WriteLine("DHCP Server = " + nic.DHCPServer);
    Console.WriteLine("IP Address = " + nic.IPAddress);
    Console.WriteLine("Subnet Mask = " + nic.IPSubnet);
    Console.WriteLine("MAC Address = " + nic.MACAddress);
}
```

You can check for changes to the network adapters by calling the RefreshNetworkAdapters method. If the list of network adapters has changed since your application started, or since the last time you called RefreshNetworkAdapters, then the OnNICChange event will be raised. For example, if a wireless connection has been made or dropped, or a change has been made to the network configuration via the control panel.

VsCoordinator Reference

VsCoordinator is part of the **Visionscape.Devices** namespace. Following is a complete list of all properties, methods and events.

Device Enumeration and Device Focus

VsDeviceCollection Devices { get; }

This property is a collection of all vision devices that can be connected to.

VsDevice FindDeviceByName(**string** *strName*)

This function looks up a device using the specified name. This function returns a VsDevice object if found, and returns null if not found.

VsDevice DeviceFocus { set; get; }

This property sets/gets the focus device. If you need to use a GroupID, to allow more than one focus device, then use DeviceFocusGet instead.

string LookupIPAddress(**string** *Name*)

Looks up the IP address of the device with the specified name. It returns the device's IP address if found, and an empty string if not found.

void DeviceFocusSet(**VsDevice** *pDevice*, **int** *nGroupID*)

This method sets the focus to the specified VsDevice object. Use the GroupID parameter in situations where multiple focus devices are required. Use the DeviceFocus property if don't need multiple focus Devices.

VsDevice DeviceFocusGet(**int** *nGroupID*)

This function returns the current focus device. Use the GroupID parameter if multiple focus devices have been specified.

VsDevice FindDeviceByIP(**string** *strIP*)

This function searches the Device collection for the supplied IP Address string. It returns a VsDevice if found, and null if not found.

void DeviceFocusSetOnDiscovery(**string** *bstrName*, **int** *GroupID*)

This method sends a OnDeviceFocus event when the specified device name is available for use. For network devices (i.e., smart cameras) this is the preferred method for connecting.

`Event` **OnDeviceDiscovered** (**VsDevice** *newDevice*)

This event occurs when a new network device (i.e., smart camera) is discovered.

`Event` **OnDeviceFocus** (**VsDevice** *newDevice*)

This event occurs whenever the current focus device changes. The focus device is selected via the DeviceFocus property, or the DeviceFocusSet method.

`Event` **OnDeviceLost** (**VsDevice** newDevice)

This event occurs when a device can no longer be communicated with; for example, if it has been physically disconnected.

`Event` **OnDeviceFocusChanging**(**VsDevice** *objDevice*, **int** *nGroupID*)

This event is sent just before the device focus is about to change.

`Event` **OnDeviceFocusEx** (**VsDevice** objDevice, **int** *nGroupID*)

This event is similar to the OnDeviceFocus event, except it provides the GroupID. This is primarily useful if there are multiple focus devices.

UI Coordination

void UpdateUI()

This method sends the OnUpdateUI event from all VsCoordinators. Use this to force various controls to refresh.

void BroadcastMessage(**string** *bstrSender*, **string** *bstrMsg*, **string** *bstrParam*)

This method sends the OnBroadcastMessage from all VsCoordinators. Use this to send coordinating messages between controls. The bstrMsg parameter is a user defined string that identifies the action to be taken. Use the bstrParam to supply additional information.

void BroadcastObj(**string** *bstrSender*, **string** *bstrMsg*, **object** *obj*)

This method sends the OnBroadcastMessage from all VsCoordinators. Similar in function to the BroadcastMessage function, except that an object can be passed as a parameter.

void SetGlobalString(**string** *bstrKey*, **string** *bstrString*)

This method associates a string with a symbolic name so that it can be retrieved later, even from a different form or module.

string LookupGlobalString(**string** *bstrKey*)

This function retrieves a Global String that was set via the SetGlobalString function.

`Event` **OnUpdateUI()**

This event occurs as a result of calling the UpdateUI method.

`Event` **OnBroadcastMessage** (**string** *bstrSender*, **string** *bstrMsg*, **string** *bstrParam*)

This event occurs as a result of calling the BroadcastMessage method, and it allows you to send your own custom messages along with a string parameter.

`Event` **OnBroadcastObj** (**string** *bstrSender*, **string** *bstrMsg*, **object** *obj*)

This event occurs as a result of calling the BroadcastObj method. It allows you to send your own custom messages, along with an object variable.

`Event` **OnNICChange()**

This event occurs if a change has been made to any Network Interface Controller settings (for example, the host PC's IP address or Net Mask).

Miscellaneous

void ShowLogWindow(**bool** *bShow*)

This method shows or hides the Visionscape debugging window.

void LogMessage(**string** *strMsg*, **bool** *bError*)

This method displays a message to the Visionscape debugging window. If bError is True, then the message is displayed in red.

object Job { set; get; }

This property gets/sets the currently loaded Job. Please refer to the detailed documentation for further information about the Job property. For more information, see “Connecting Jobs to Visionscape Devices” on page 3-6.

VsNetworkAdapterCollection NetworkAdapters { get; }

This property returns a collection of VsNetworkAdapter objects, containing information about the Network Interface Controllers on the PC.

VsDevice Reference

The VsDevice object is part of the **Visionscape.Devices** namespace. Following is a complete list of all properties, methods and events.

Identification and Information

Table 3–2 summarizes the identification and informational properties of VsDevice.

TABLE 3–2. Identification Properties of VsDevice

Property Name	Description
Name	The name of the Device. For smart cameras, this name is user assigned.
Key	A unique key string identifying the device.
DirectoryID	An ID that identifies the device in the VsDirectory structure.
DeviceClass	The class of device. The value can be one of the following: DEVCLASS_UNKNOWN=0 DEVCLASS_SOFTWARE_EMULATED=1 DEVCLASS_HOST_BOARD=2 DEVCLASS_PROCESSOR_BOARD=3 DEVCLASS_SMART_CAMERA=4 DEVCLASS_SMART_CAMERA_OLDER=5 DEVCLASS_SMART_CAMERA_UNREACHABLE=6 DEVCLASS_CAMERA=7
DeviceModel	The device model number.
DigitizerModel	The digitizer model number.
IsHostBased	True if the host PC's CPU is used to run the inspections.
SoftwareVersion	A string representing the software revision loaded on the device.
IPAddress	The IP address of the device.
MACAddress	The MAC address of the device.
NetMask	The Network Mask of the device.
NetworkConnectable	True if the device is on the same subnet as the host PC.
TimeSinceLastRefresh	The time in milliseconds since the last UDP info message has been received from the device.

TABLE 3–2. Identification Properties of VsDevice

NameOfController	The name or IP address of the PC that has control of the device.
IsInTryout	True if the device is in Tryout mode.
IsInLive	True if the device is in acquire live mode.
IsInAcquire	True if the device is acquiring an image for setup.
HaveControl	True if the current process has taken control of the device by using the TakeControl method.
DeviceState	The current state of the device. Can be one of the following values: DEVSTATE_UNKNOWN=0 DEVSTATE_RUNNING=1 DEVSTATE_STOPPED=2 DEVSTATE_NOJOB=3 DEVSTATE_NOCOMM=4 DEVSTATE_ERROR=100 DEVSTATE_FILE_XFER=101 DEVSTATE_TRYOUT=102 DEVSTATE_EDIT=103 DEVSTATE_LIVE=104 DEVSTATE_FUNCTION=105 DEVSTATE_ACQUIRE=106

Download / Upload Job

TransferStatus Download(JobStep *job*, **bool** *bWait*)

This function downloads the specified Job to the device. If the Job contains multiple VisionSystem Steps, the first is downloaded. If the bWait parameter is true, the function will not return until the download is complete. If false, an asynchronous download is started and the function returns immediately. The status of the download is returned in the enumerated TransferStatus value. This function can also throw an exception.

int Download(VisionSystemStep *objVS*, **int** *bAsync*)

This overloaded version of the Download function takes in a VisionSystemStep rather than the Job step. You can ignore the 2nd parameter, this version always performs an Asynchronous download. You would typically use this version if your Job contains more than one VisionSystemStep, and you need to be able to download any of them.

TransferStatus DownloadAVP(**string** *JobPath*, **bool** *bWait*)

This function downloads the Job contained in the specified AVP file. If the bWait parameter is true, the function will not return until the download is complete. If false, an asynchronous download is started and the function returns immediately. The status of the download is returned in the enumerated TransferStatus value. This function can also throw an exception.

bool Upload(**JobStep** *job*)

This function uploads the Job that is currently loaded on the device. This function uploads a VisionsSystem step (and all of its child steps), and inserts it into the specified JobStep. Refer to the detailed documentation for further information on uploading. For more information, see “Uploading a Job” on page 3-15.

tagXFERSTATUS CheckXferStatus(**int** *sleep_ms*)

After initiating an upload or download in asynchronous mode, call the CheckXferStatus function in a loop until the transfer is complete. The sleep_ms parameter specifies the number of milliseconds to sleep if the transfer is not complete. It’s recommended that the loop contain an Application.DoEvents call so that the user interface remains responsive.

Control

bool TakeControl(**string** *bstrUID*, **string** *bstrPWD*)

This function allows you to TakeControl of the smart camera by logging in with a User ID and Password.

bool HaveControl { get; }

This property returns True if you currently have control.

void ReleaseControl()

This method releases control of a smart camera.

void StartAll()

Starts all inspections on the device.

void StartInspection(**int** *nInsp*, **int** *nCycles*)

This method starts the specified inspection (0 based index) on the device. If nInsp is set to -1, all inspections are started. You can use the nCycles parameter to specify the number of cycles to run, set this to 0 to run for an infinite number of cycles.

void StopAll()

Stops all running inspections on the device.

void StopInspection(**int** *nInsp*)

This method stops the specified inspection (0 based index) on the device. Pass in -1 to stop all inspections.

bool IsInspectionRunning(**int** *nInsp*)

This function returns True if the specified inspection is running on the device.

bool IsAnyInspectionRunning { get; }

This property returns True if any inspection is running on the device.

int NumInspections { get; }

This property returns the number of inspections on the device.

void ResetCounters(int nInsp)

This method resets the inspection counters for the specified inspection.

int ResetDevice(**string** *bstrUser*, **string** *bstrPassword*)

This function reboots the Network Device. It requires that you specify a username and password. Has no effect on Host based Devices.

Advanced

VsioCaps QueryioCaps()

This function queries the device for a structure that describes the I/O capabilities, such as the number of each type of I/O supported.

VsUDPInfo UDPInfo { get; }

This property returns a structure with all the information contained in the UDP info packet sent by smart camera network devices.

object ProgramController { set; get; }

If a Job is loaded into host memory, each VisionSystemStep can be accessed via its “Program Controller” interface. This property provides the means to access that interface.

void GetDeviceBufferDm(**string** *bstrPath*, out **Visionscape.Steps.BufferDm** *objBufDm*)

Use this method to directly read a buffer out of a device and copy the data into the supplied BufferDm. The bstrPath parameter should be the symbolic name path of the Buffer desired.

4

CHAPTER 4

Receiving Data with Report Connections

Introduction to the Visionscape.Communications Namespace

In the previous chapters we explained how to load vision Jobs, and how to download them and get them running on any Visionscape Device. In this chapter we will discuss how to receive information from your running inspections. The objects in the Visionscape.Communications namespace will allow you to receive and handle reports at runtime that can contain images and/or result data.

Assembly Names: Visionscape.dll & Visionscape.Communications.dll

Namespace: Visionscape.Communications

To access this namespace, add the following .NET references to your project:

```
Visionscape
Visionscape.Communications
```

Add the following statement to the top of your C# files in order to make access to this namespace easier (all sample code in this chapter assumes the following statement is presnt):

```
using Visionscape.Communications;
```

ReportConnection Object:

The ReportConnection object can connect to any Visionscape Device, and it allows you to receive inspection results, inspection stats, and images while the Device is running. Key points to understand:

- Connects to any Visionscape Device type. It will receive reports across the network when connecting to a smart camera, or across threads within your PC application process when dealing with GigE or Software Systems. It can even receive reports across processes when dealing with GigE and Software Systems.
- A ReportConnection connects to one and only one Inspection Step at a time. If you are running multiple inspections, you will create a separate ReportConnection for each. You can also have multiple ReportConnections connected to the same inspection.
- You will receive the NewReport event to notify you of new inspection cycle reports.

A ReportConnection can be configured to be “lossless”, insuring that you will receive a cycle report from every inspection cycle. Or it can be configured to be “lossy” meaning it will throw away cycle reports when your application is too busy to receive them.

This NewReport event passes you an InspectionReport object. The InspectionReport object will contain the inspection stats (in the form of a ReportInspectionStats object), the uploaded results (in the form of an InspectionReportValues object) and potentially a collection of images (in a collection of BufferDm objects, accessed via the Images property). More on that later, lets start with the basics.

Creating a Report Connection

The following sample code demonstrates how to establish a report connection to a Device. In this example, assume you have a GigE system named “GigeVision1” on which a Job is already loaded and running:

```
//Declare a member variable of type ReportConnection
private ReportConnection m_RepCon = new ReportConnection();
private void frmMain_Load(object sender, EventArgs e)
{…
```

```
        //connect to the 1st inspection on the device "GigeVision1"
        m_RepCon.Connect("GigeVision1", 1);
        //wire up our event handler
        m_RepCon.NewReport += m_RepCon_NewReport;
}
//You will now receive this event whenever a new
// Inspection report is available
void m_RepCon_NewReport(object sender,
ReportConnectionEventArgs e)
{
    Console.WriteLine("Received a New Inspection Cycle
Report");
}
```

You simply declare a member variable in your form/class to be of type ReportConnection, instantiate the object, and call one of the Connect methods. In this case, we connected by passing in the name of the device, and the 1 based index of the inspection we wanted to receive reports from. You must also wire up the event handler, which in this case is our m_RepCon_NewReport function. At runtime, the m_RepCon_NewReport function will be called whenever a new report is generated (new reports are always generated at the end of an inspection cycle). The event will pass you a ReportConnectionEventArgs object, which contains an InspectionReport object. This object contains all of your inspection counts, timing, uploaded results, and even images if you choose to add them to the report (more on that later). We will cover the contents of the InspectionReport object in more detail later on in this chapter.

Connection Details

As we just demonstrated, the Connect method of ReportConnection establishes a report connection to one of the inspections on a running device. We provide several overloaded versions of the Connect function:

bool Connect(string *deviceName*, **int** *inspIndex*, **ReportConnection.ConnectOptions** *reportOptions*)

- *deviceName* – The name of the Device to connect to.
- *inspIndex* – The 1 based index of the inspection you want to receive reports from.

– *reportOptions* – Specifies the type of data that should be included in the Inspection Report. This is specified using the enumerated type ConnectOptions, which is a member of the ReportConnection object. ConnectOptions is a Flag type, meaning you can combine options. The Connect Options are:

  - `ConnectOptions.NONE`: This option will automatically include nothing in the report data. You would use this option when you only want to receive data that you added programmatically via the DataRecordAdd method.
  - `ConnectOptions.STATS`: Include inspections stats in the report. This includes inspection counts, timing info, and memory data, among other values.
  - `ConnectOptions.TAGGED_FOR_UPLOAD`: Include the inspection Datums that have been tagged for upload. This would include any values that you have selected in the Inspection Step's "Results to Upload" datum, as well as any Datum values that you have programmatically tagged for upload.
  - `ConnectOptions.DEFAULT`: The default is to include both the Stats and the Datums that are tagged for upload. This option is equivalent to:
  - `ConnectOptions.STATS` | ConnectOptions.TAGGED_FOR_UPLOAD
  - The following two options are rarely used, but are available.
  - `ConnectOptions.NO_IMAGES`: Never include images in the report. One way to include image buffers in your report is to add them to your Inspection's "Result to Upload" list in FrontRunner. In that case, you could combine this option with the TAGGED_FOR_UPLOAD option in situations where you only wanted the inspection data, and wanted the images to be left out to improve performance.
  - `ConnectOptions.IGNORE_UPLOAD_QUALIFIER`: The Inspection Step has a Datum named "Results Upload Qualified Condition". This datum allows you to enter an expression that determines when results should be uploaded.

Setting this flag will cause that expression to be ignored, and results will be uploaded after each cycle.

Connects to the specified inspection, on the specified Device. The data that will be automatically added to the report is determined by the value you specify in the reportOptions parameter. The function returns true if the connection is successfully established, false if not. Once connected, you will receive the NewReport event whenever the inspection has a report to send to you.

bool Connect(**Visionscape.Devices.VsDevice** *device*, **int** *inspIndex*, **ReportConnection.ConnectOptions** *reportOptions*)

Identical to the Connect method just documented, only this version takes a reference to the VsDevice object you are connecting to, rather than it's name.

bool Connect(**string** *deviceName*, **int** *inspIndex*)
bool Connect(**Visionscape.Devices.VsDevice** *device*, **int** *inspIndex*)

These versions connect to the Device specified by either it's name or by the actual VsDevice object. The inspection is specified by the inspIndex parameter. These versions however do not require you to specify any ConnectOptions. This version automatically specifies ConnectOptions.DEFAULT.

bool Connect(**string** *deviceName*)
bool Connect(**Visionscape.Devices.VsDevice** *device*)

These versions of Connect will connect to the Device specified by it's name or by the actual VsDevice object. However these versions automatically choose the first inspection, and automatically specify ConnectOptions.DEFAULT as the ConnectOptions value.

There are several other important properties of the ReportConnection object that allow you to fine-tune the performance of your connection.

public **bool DropWhenBusy** { set; get; }

The default for this property is true, which means your reports will be dropped if the Inspection is too busy to send them. This means your connection will be lossy. If you require a lossless connection, meaning you don't want any reports to be dropped, then you must set this property to false. You should understand however that a lossless connection may

impact the performance of your inspection. If your inspection completes a cycle, and it tries to send a report, but your report connection is still busy trying to send the report from the previous cycle, then the inspection will block until it can send the new report. This could cause problems for high-speed, time-critical inspections. If this property is set to true, the Inspection would not block; it would simply drop the report and continue on to the next inspection cycle.

public **int MaxRate** { set; get; }

Use this property to set the maximum number of reports that can be sent per second. The default is 0, which means to send the maximum number of reports possible, no limit. A setting of 2 would mean no more than 2 per second. Using 2 as an example, the connection would translate this setting into a number of milliseconds (1000ms / 2 = 500ms). Whenever the connection sends a report, it will start timing, and if another report is ready for transfer in less than 500ms, then it will be thrown away. In other words, your running inspection will only send one report every 500ms.

ReportConnection.FreezeModeOptions FreezeMode { set; get; }

By default, your report connection will try to send a report after every inspection cycle. You can use this property to change that behavior by specifying one of the enumerated values in ReportConnection.FreezeModeOptions :

- `FreezeModeOptions.SHOW_ALL` — Default, send a report after every inspection cycle.
- `FreezeModeOptions.SHOW_FAILED` — Only send a report when the inspection fails.
- `FreezeModeOptions.FREEZE_THIS` — Freezes the report connection, which means no more reports will be sent.
- `FreezeModeOptions.FREEZE_NEXT_FAILED` — Freezes the report connection on the next failed inspection. Not to be confused with the SHOW_FAILED option, reports will be sent continuously while in this mode UNTIL there is an inspection failure, at which time it will freeze.
- `FreezeModeOptions.FREEZE_LAST_FAILED` — Switching to this mode will cause a report to be sent from the last failed inspection cycle, and then the connection will be frozen.

– `FreezeModeOptions.FREEZE_NEXT_QUAL` — This option only applies if you are using the “Freeze Qualified Condition” datum in the Inspection Step. The datum allows you to specify an inspection condition which, if evaluated as True, will freeze the connection. Without this setting, the datum in the inspection step has no effect on your report connection.

bool ExcludeTaggedImages { set; get; }

The default is False. When set to True, any images buffers that have been added to the list of results to upload will be excluded from the report. This property has the same effect as specifying ConnectOptions.NO_IMAGES when calling the Connect method.

public **bool GraphicsOn** { set; get; }

The default is True. When set to False, any images included in the report will not include graphics.

The NewReport Event:

Once you have established a report connection, you will receive inspection cycle reports by handling the NewReport event. To handle this event you must wire up an event handler. The function signature for the NewReport event handler must be as follows:

```
void NewReportHandler(object sender,
ReportConnectionEventArgs e)
```

– sender: The object that generated the event

– e: A ReportConnectionEventArgs object. This is derived from EventArgs and adds the following two properties:

  • Report: This is a reference to the InspectionReport object. All of your cycle data is contained within this one object..

  • GoingToFreeze: This bool value will be set to true when the image display is about to be frozen, false otherwise.

The following example demonstrates wiring up an event handler, and then extracting the InspectionReport when an event is generated. Assume our ReportConnection object is named m_RepCon:

```
//connect to the 1st inspection on the device "GigeVision1"
m_RepCon.Connect("GigeVision1", 1);
//wire up our event handler
m_RepCon.NewReport += m_RepCon_NewReport;
//Our event handler
void m_RepCon_NewReport(object sender,
ReportConnectionEventArgs e)
{
   //get the report from the ReportConnectionEventArgs object
InspectionReport report = e.Report;
}
```

We will cover the InspectionReport object and it's contents in more detail later on in this chapter.

Adding Records to Your Report Programmatically

When you establish a report connection, you will typically receive just the inspection stats and the results that you selected for upload when you were building your Job in FrontRunner. You have the option of adding other datums to the report programatically however. You accomplish this by using one of the overloaded versions of the DataRecordAdd method:

void DataRecordAdd(Visionscape.Steps.IDatum *datumToAdd*)

datumToAdd: The datum that you want to add to the list of uploaded results.

This method adds the specified datum to the end of the list of results to upload. It must be called after your call to Connect. You would use this method when you are loading the Job in your application, and you can then locate the datums you want to add. Refer to Chapter 2 and the "Finding Steps in the Step Tree" section for more information on how to locate steps and datums.

void DataRecordAdd(Visionscape.Devices.VsNameNode *nnForStep*, **string** *DatumName*)

nnForStep: A VsNameNode object that represents the parent Step of the datum you wish to add.

DatumName: The name of the datum to add.

This method adds a datum to the list of results to upload. It must be called after your call to Connect. The Datum is specified by passing in a VsNameNode object that represents the parent Step, and a string that holds the name of the Datum. You would use this method when you are writing a monitoring application, and you do not have the actual Job loaded in your process. Refer to Chapter 3 and the “Namespace Information” section for more information.

void DataRecordAdd(string *dataRecordPath*)

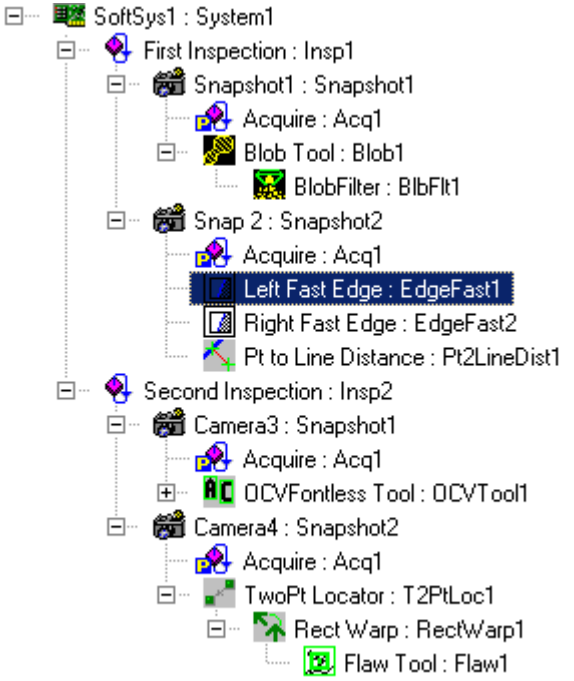
dataRecordPath: A string that specifies the symbolic name of the datum that you want to add to the report. This must be the full path to the datum, up to its parent Snapshot Step.

This method adds a Datum to the report as specified by the dataRecordPath parameter. This string represents the symbolic name of the Datum you want to add. This must be the full path to the Datum, up to its parent Snapshot Step.

DataRecordAdd Examples:

Consider the Job tree shown below; this is from the ProgSample_MultiCam.avp file installed with the Visionscape VSKit.NET Programmers Toolkit:

FIGURE 4–1. Job Tree



The Job Tree shown here is displaying the symbolic names for each Step. The first Fast Edge Step under the second Snapshot Step is highlighted. Let's say we wanted to add the output edge point from this Fast Edge Step to our list of uploaded results. The symbolic name for the "Edge Point" datum is EdgePt (remember, you can look up the symbolic name of any datum using the StepBrowser utility, refer to "Using StepBrowser to Look Up Symbolic Names" on page 2-30 for details). The symbolic name of the step is EdgeFast1, and its parent Snapshot Step's symbolic name is Snapshot2. So, we could make the following call to DataRecordAdd:

```
m_RepCon.DataRecordAdd("Snapshot2.EdgeFast1.EdgePt");
```

In this case, we specified the complete path string of the Datum. If you have the Job loaded in memory, then you could also add the Datum by directly passing it to DataRecordAdd, like this:

```
//locate the fast edge step in our job
Step fastedge = m_Job.FindByName("Left Fast Edge");
//pass the EdgePt datum to DataRecordAdd
m_RepCon.DataRecordAdd(fastedge.Datum("EdgePt"));
```

If you don't know the symbolic name of the Datum you want to add, and you don't have the Job loaded in memory, you can still add a Datum to the result upload list by analyzing the Namespace of the Device. You would typically need to do this when your application is simply monitoring a running device. The following example adds the same EdgePt datum to the report, but does so using the Namespace information.

```
//Tell the VsDevice object to update its namespace information
m_dev.QueryNamespace();
//Search the Namespace for all of the Fast Edge Steps
VsNameNodeCollection fedgeNodes =
 m_dev.Namespace.SearchForType("Step.Edgefast.1");
//Loop through all of the Fast Edge Nodes
foreach(VsNameNode fedgeNode in fedgeNodes)
{
     //is this the Fast Edge step named "Left Fast Edge"?
     if(fedgeNode.UserName == "Left Fast Edge")
     {
         //Add the "EdgePt" datum of this Step to the Report
         m_RepCon.DataRecordAdd(fedgeNode, "EdgePt");
     }
}
```

Remember, a call to any of the DataRecordAdd functions will fail if you have not successfully connected first.

Adding Images to Your Report

There are two ways to include images in your inspection report.

- When programming your Job in FrontRunner, go to the Inspection Step and add the output buffer of the snapshot(s) you want to upload to the "Select Results to Upload" list. Then, these buffers will be

added to the Images collection of the InspectionReport object you receive in the NewReport event.

- Add them programmatically.

The first option should be self explanatory, so let's focus on the second option. In the previous section, we explained how to add datums to your inspection report using the DataRecordAdd method. You should understand that the images in your Job are contained within Buffer Datum objects and are, therefore, added to the inspection report just like any other datum. Each Snapshot Step has an output Datum named "SnapOutputBuffer", symbolic name "BufOut". This datum always holds the most recent image acquired by the Snapshot. So, the call to include the image from a Snapshot would look something like this:

```
m_RepCon.DataRecordAdd("Snapshot1.BufOut");
```

Our assumption here is that the symbolic name of the Snapshot is "Snapshot1". But, what if you don't know what the symbolic name of the Snapshot is? What if your inspection contains multiple Snapshots, and you want to upload the images from all of them? Not to worry, ReportConnection provides several easy ways to add images to your reports.

Adding All of the Snapshot Images in an Inspection to the Report

If your application is using a JobStep to load an AVP file into memory, or simply monitoring a running Device, there are two easy ways to add images to your report. Simply call the appropriate version of the AddSnapBuffers() function. This function will add to the report all of the Snapshot Output Buffers in the Inspectin you are connected to.

int AddSnapBuffers(Visionscape.Steps.JobStep *job*)

job: A reference to the Job Step that is currently loaded on the Device.

This function will scan the specified Job for the Inspection that you are currently connected to. It will then find all of the Snapshot steps witin that Inspection, and automatically add their output buffers to the report. You must successfully connect before calling this function. If successful, the number of snapshot output buffers added to the report is returned, 0 is returned on a failure.

int AddSnapBuffers(Visionscape.Devices.VsNameNode devNamespace)

devNamespace: The namespace property from the Device you are currently connected to.

This version scans the namespace for the Inspection you are currently connected to, and will also add all of its snapshot output buffers to your report. You must have successfully connected first. If successful, the number of snapshot output buffers added to the report is returned, 0 is returned on a failure.

AddSnapBuffers Examples

Assume you have the current job loaded into a JobStep named m_Job, and you are connecting to a VsDevice object named m_dev. The following example creates and connects a ReportConnection object, and then adds all snapshot buffers to the report.

```
//connect to the 1st inspection on this device
m_RepCon = new ReportConnection();
m_RepCon.Connect(m_dev);
//Add all snapshot buffers in the inspection to the report
m_RepCon.AddSnapBuffers(m_Job);
```

In this example, assume you are only monitoring a VsDevice object named m_dev, and you do not have the Job loaded in your process. Do the following to add all Snapshot buffers to your report:

```
//connect to the 1st inspection on this device
   m_RepCon = new ReportConnection();
   m_RepCon.Connect(m_dev);
//Tell the device to update it's namespace information first
   m_dev.QueryNamespace();
   //now add all snapshot buffers to the report
   m_RepCon.AddSnapBuffers(m_dev.Namespace);
```

Now I Have Images, How Do I Display Them?

Now that you know how to include images in your inspection reports, the next obvious question is how to display them. The easiest and most powerful Visionscape control for displaying images is the BufferView control. In Visual Studio's Toolbox, a tab should have been created by the Visionscape installer named "Visionscape". This tab will contain all of the visual controls provided by Visionscape. You should find the BufferView

control in this tab. Drag a BufferView control onto your form and rename it to ctlBufView (refer to Chapter 6 for a complete description of the BufferView control).

When you receive the NewReport event, the Images collection of the InspectionReport object will contain the images from every snapshot you added to the report. The Images property is a collection of BufferDm objects. Visionscape always uses the BufferDm object to represent an image buffer. The BufferView control displays BufferDms; you simply need to set its Buffer property to the BufferDm you wish to display. The following example shows how you would extract the first image from your report, and display it in the BufferView control named ctlBufView in your NewReport event handler.

```
//The Event handler for the NewReport event
void m_RepCon_NewReport(object sender,
ReportConnectionEventArgs e)
{
  //get the Inspection report from the
  //ReportConnectionEventArgs object
    InspectionReport report = e.Report;
    //does our report contain any images?
    if(report.Images.Count > 0)
    {
        //extract the image as a BufferDm,
        // and set it into the Buffer View
        ctlBufView.Buffer = (BufferDm)report.Images[0];
    }
}
```

If your report contains multiple images, then you would add multiple BufferView controls to your form, and iterate through the Images collection in order to display them all.

Performance Considerations

The ReportConnection is very flexible and can be deployed in a number of ways to handle most application scenarios. How it is deployed depends on your situation. In this section we list several important topics that impact the performance of your ReportConnection, and your running inspections. Keep these issues in mind when deciding how best to configure your ReportConnections.

Lossy vs Lossless

The DropWhenBusy property governs whether or not your ReportConnection is Lossy or Lossless. When set to true, your connection is lossy, if false, it is lossless.

Lossy: Your report connections are “lossy” by default. This means that you will not necessarily receive a report from every inspection cycle. Cycle reports (represented by the InspectionReport object) may be thrown away at the end of a cycle if your application is still busy handling the report from the previous cycle. This prevents the Inspection thread from being blocked while it waits for you to handle the new report, and insures that your application will not negatively impact the speed of your inspections. You should choose a lossy connection when you don’t need to receive data from every inspection cycle, and it is most critical that your application not impact the speed of your inspections.

Lossless: This means that your running inspections will NOT throw away cycle reports when your application is busy. Your inspection will block if your application is still busy handling the previous report. This means that your application could cause your inspections to run slower. Choose a lossless connection when it is critical that you receive data from every inspection cycle, and you are not concerned with the potential impact on the inspection speed.

Don’t Spend too much time in the NewReport event handler:

For most of our customers, it is critical that their applications not impact the speed of their vision inspections. For this reason, a lossy report connection is most appropriate for them. However, we often hear people say “…but I still need to get most if not all of the inspection cycle data”. In this case, the best strategy is to spend as little time in the NewReport event handler as possible. If your application is in the NewReport event handler, and your Inspection completes another cycle and has a new report for you, that report will be thrown away when you have a lossy connection. You should try to simply grab the InspectionReport (assign it to a member variable, put it into a List, etc) and then exit. You could then process the reports based on a timer, or even pass them to a different thread. The more time you spend in the NewReport event handler, the greater the risk that you will drop reports.

Separate ReportConnections for Images and Results:

In all of the examples we've presented in this chapter, we have demonstrated receiving both the images and the results in the same ReportConnection. For most applications, the images are simply displayed, and it is not critical that you receive every one. However it is very common that the result data IS critical, and the application must receive every cycle report. In these cases, we recommend that you create one ReportConnection to receive images, and a separate ReportConnection to receive Inspection results and stats. The one that will receive images can be configured to be lossy, and the MaxRate can be set to some appropriate value (say 2 or 4). The ReportConnection that will receive inspection results and stats can be configured so that MaxRate is set to 0, and DropWhenBusy is set to true (lossy) if you want most of the results or false (lossless) if you want all of the results. In all but a few cases, the images represent much more data than the inspection results and stats, so it is better to throw away the images when your application is busy and upload just the results and stats on a separate connection.

The InspectionReport Object:

At the end of each inspection cycle, the Inspection Step will create a report that contains the cycle data you requested from your ReportConnection object. This raw report is then transferred to your ReportConnection object, where it is transformed into an InspectionReport object. So, think of the InspectionReport object as a report on one cycle of your inspection. This report is passed to you via the report proprety of the ReportConnectionEventArgs object in the NewReport event. The following properties of the InspectionReport object hold your report data:

ReportInspectionStats InspectionStats { get; }

Returns a reference to a ReportInspectionStats object that contains information like the inspection status (pass/fail), cycle counts, timing, etc. Refer to "ReportInspectionStats Object" on page 6-13 for a complete description.

ReportMemoryInfo Memory { get; }

Returns a ReportMemoryInfo object that contains information on the device's current memory usage. Refer to “ReportMemoryInfo” on page 6-17 for a complete description.

ReportResultList Results { get; }

Provides access to the list of uploaded inspection results. This is a list of InspectionReportValue objects, which are objects that represent a single inspection result. There will be one InspectionReportValue object in the list for every result selected for upload. The following sample demonstrates how you might access the results in this list:

```
void m_RepCon_NewReport(object sender,
ReportConnectionEventArgs e)
{
    //get the Inspection report
    InspectionReport report = e.Report;
//iterate through all of the results
    foreach(InspectionReportValue val in report.Results )
    {
        Console.WriteLine("Result Type: " + val.Type);
    }
    //access individual results
    InspectionReportValue rec = report.Results[0];
    InspectionReportValue rec2 = report.Results[2];
}
```

Refer to “InspectionReportValue Object” on page 6-16 for a complete description of the contents of this object.

BufferDmList Images { get; }

A list of BufferDm objects. This collection holds all image buffers that were added to the report, if any. The default report connection does not contain images, so this collection will be empty unless you have added images to the report. Refer to the previous section “Adding Images to your Report” for a description of how to do this.

InspectionReport also provides the following useful properties and methods:

IEnumerable<InspectionReportValue> GetResults(string *typeString*)

typeString: The type of datum you want to retrieve.

This method allows you to iterate through only the results that are of a certain type. The typeString is used to specify the Datum type you want to retrieve. The following example demonstrates how you might iterate through only the Status datums in your result list:

```
 foreach(InspectionReportValue val in
report.GetResults("Status"))
    {
        Console.WriteLine(val.Name + " : " + val.NameSym);
        Console.WriteLine("Status Value = " + val.AsBool);
    }
```

double TimeStamp { get; }

Returns the timestamp from when report was created.

string ReportString(Visionscape.Communications.ReportLogOptions *logOptions*).

This function converts your report data into a string based on the options you specify in the logOptions parameter. Refer to “Logging Results to File” on page 4-14 for a complete description of the ReportLogOptions object.

bool FileSave(string *bszName*)
bool FileLoad(string *bszName*)

bszName: File path to which report should be saved/loaded.

These two methods allow you to save and reload inspection reports to disk. The entire contents of the report, including images, will be saved.

ReportInspectionStats Object

The properties of this object provide information on the inspection you are connected to, the cycle counts, inspection timing, overruns, buffer usage, etc. A description of each property follows:

ReportSnapshotInfo[] Snapshot { get; }

Returns an array of ReportSnapshotInfo objects, one for each of the snapshots in the inspection. The ReportSnapshotInfo contains the following information for each snapshot.

int BufferPoolCount { get; }

The size of the buffer pool.

int BufferPoolUsed { get; }

The number of buffers being used.

int CameraTimeouts { get; }

Contains a count of the number of camera timeout errors for this snapshot.

int FifoOverruns { get; }

Contains a count of the number of Fifo Overrun errors for this snapshot.

int ProcessOverruns { get; }

Contains a count of the number of Process Overrun errors for this snapshot.

int TriggerOverruns { get; }

Contains a count of the number of Trigger Overrun errors for this snapshot.

int CycleCount { get; }

This property returns inspection cycle count.

int CycleTime { get; }

This property returns Cycle time in msecs.

int CycleTimeMax { get; }

This property returns the maximum cycle time in msecs.

int DrawTime { get; }

This property returns the time in msecs spent creating graphics metafile.

int FailedCount { get; }

This property returns the number of failed inspections.

int IdleTime { get; }

This property returns the idle time in msecs.

int PartQueueSize { get; }

Read-only. The current number of entries in the Part Q.

int PartQueueSizeMax { get; }

Read-only. The maximum Size of the Part Q.

bool Passed { get; }

True if inspection passed, False if inspection failed.

int PassedCount { get; }

This property returns the number of passed inspections.

int ProcessTime { get; }

This property returns the processing time in msecs (time spent in inspection processing images).

int ProcessTimeMax { get; }

This property returns the maximum processing time.

int RatePPM { get; }

This property returns the Inspection rate as parts per minute.

int RatePPMMax { get; }

This property returns the maximum Inspection rate as parts per minute.

InspectionReportValue Object

This object wraps a single result in the list of uploaded results. You are provided with properties that allow you to access the result data, its name, its error code, etc. The result data can be of any type, including array data, so various accessor properties are provided that will return the data to you as the correct type, rather than just return an object type.

bool Calibrated { set; get; }

Gets/Sets whether values should be returned in calibrated units or in pixels. Set to True, and when you use one of the value access properties (AsDistance, AsPoint, etc), your value will be returned in calibrated units. Please note, you must have calibrated your inspection in order for this property to have any effect. If this value is set to False, or the inspection has not been calibrated, all value accessors will return their values in pixels.

string Type { get; }

Gets the type of the uploaded result. This is a string which identifies the type of datum. You can use this value to determine which of the accessor methods you should use:

```
  InspectionReportValue rec5 = report.Results[5];
if (rec5.Type == "Distance")
{
    Console.WriteLine(rec5.AsDistance.Dist);
}
```

bool Is1DArray { get; }
bool Is2DArray { get; }

These properties allow you to test if the value contains 1 dimensional or 2 dimensional data.

int Error { get; }

This property returns the error code of the datum. This will be 0 when the Step ran successfully, non-zero indicates an error. In many cases, an error will mean that the data was not updated, so you should not trust the data if Error is non-zero.

string Name { get; }

This property returns the user name of the datum that generated this record. Will be in the form Step.Datum.

string NameSym { get; }

This property returns the symbolic name of the datum that generated this record. It will be in the form Step1.Dm.

object AsObject { get; }

Returns the value of the uploaded result as an object.

The following properties all provide type specific access to the result data:

bool AsBool { get; }

Returns the value of the uploaded datum as a boolean. This would typically be used for Status Datums. A WrongTypeException will be thrown if the result's data is not boolean.

int AsInt { get; }

Returns the value of the uploaded datum as an integer. A WrongTypeException will be thrown if the result's data is not an integer.

double AsDouble { get; }

Returns the value of the uploaded datum as a double. This property can be used to extract the data from any result that produces a scalar floating point value, such as Distance, Angle, Area and of course Double. A WrongTypeException will be thrown if the result's data is not floating point.

string AsString { get; }

Returns the value of the uploaded datum as a string.

A WrongTypeException will be thrown if the result is not a String Datum. This property does not attempt to convert numeric values to strings. To do that, use the AsObject accessor, and call the standard ToString() method:

```
string strdis = rec.AsObject.ToString();
```

double[] As1DDoubleArray { get; }

Returns the value of the uploaded datum as a 1 dimensional array of doubles. This property can be used to extract the data from any Datum that produces a one dimensional array of values, like Points and Lines for example. A WrongTypeException will be thrown if the result's data is not represented by a 1 dimensional array.

double[,] As2DDoubleArray { get; }

Returns the value of the uploaded datum as a 2 dimensional array of doubles. This property can be used to extract the data from any Datum that produces a two dimensional array of values, like BlobTrees, DMR Results and OCV Results for example. A WrongTypeException will be thrown if the result's data is not represented by a 2 dimensional array.

ReportLine AsLine { get; }

Returns the uploaded value in the form of a ReportLine object, which provides individual properties to access the A, B and C elements of the line equation (Ax + By + C = 0).

```
//extract the 3rd result from the InspectionReport object
//'report'
    InspectionReportValue rec = report.Results[2];
    ReportLine line = rec.AsLine;
    //access the individual elements of the line equation
    double A = line.A;
    double B = line.B;
    double C = line.C;
```

A WrongTypeException will be thrown if the result is not a LineDm.

ReportPoint AsPoint { get; }

Returns the uploaded value in the form of a ReportPoint object, which provides individual properties to access the X, Y, Scale and Angle values of the point.

```
//extract the 3rd result from the InspectionReport object
//'report'
    InspectionReportValue rec = report.Results[2];
    ReportPoint pt = rec.AsPoint;
    //access the individual elements of the point
    double X = pt.X;
    double Y = pt.Y;
    double Angle = pt.Angle;
    double Scale = pt.Scale ;
```

A WrongTypeException will be thrown if the result is not a PointDm.

ReportDistance AsDistance { get; }

Returns the uploaded value in the form of a ReportDistance object, which provides the Dist property to return the distance value as a double.

```
//extract the 3rd result from the InspectionReport object
//'report'
    InspectionReportValue rec2 = report.Results[2];
    //get the distance value
    double theDistance = rec2.AsDistance.Dist;
```

A WrongTypeException will be thrown if the result is not a DistanceDm.

ReportAngle AsAngle { get; }

Returns the uploaded value in the form of a ReportAngle object, which provides an Angle property to return the angle value as a double.

```
//extract the 3rd result from the InspectionReport object
//'report'
    InspectionReportValue rec2 = report.Results[2];
    //get the angle value
    double theAngle = rec2.AsAngle.Angle;
```

A WrongTypeException will be thrown if the result is not an AngleDm.

ReportArea AsArea { get; }

Returns the uploaded value in the form of a ReportArea object, which provides an Area property to return the area value as a double.

```
//extract the 3rd result from the InspectionReport object
//'report'
    InspectionReportValue rec2 = report.Results[2];
    //get the area value
    double theArea = rec2.AsArea.Area;
```

A WrongTypeException will be thrown if the result is not an AreaDm.

ReportBlobTree AsBlobTree { get; }

Use this property to retrieve the result data when uploading the Blob Tree from a Blob Step. The data is returned in the form of a ReportBlobTree object. You can loop through all of the blobs in the tree and analyze their contents as shown in the sample below:

```
if(rec.Type == "BlobTree")
{
        ReportBlobTree btree = rec.AsBlobTree;
      foreach(ReportBlob blob in btree)
      {//The ReportBlob object provides property access to
       // all blob data
          double fval = blob.XCenter;
          fval = blob.YCenter;
          int nval = blob.Color;
          fval = blob.UnrotWidth;
      }
}
```

A WrongTypeException will be thrown if the result is not a BlobTreeDm

ReportDMR[] AsDMRResults { get; }

Use this property when you are uploading the SymResults datum from a Data Matrix tool. It returns an array of ReportDMR objects, one for every Data Matrix decoded by the DMR tool. The ReportDMR object provides property access the DMR result data.

A WrongTypeException will be thrown if the result is not a DMRResults datum.

ReportBCR[] AsBCRResults { get; }

Use this property when you are uploading the SymResults datum from a Bar Code tool. It returns an array of ReportBCR objects, one for every Barcode decoded by the tool. The ReportBCR object provides property access to the decoded barcode result data.

A WrongTypeException will be thrown if the result is not a DMRResults datum.

ReportMemoryInfo object:

This object provides information on the state of a remote Device's memory. You should understand that the data in this object is only valid when dealing with a smart camera (Vision HAWK, for example). The data is not valid when dealing with host based devices, (GigE and Soft Systems) as your inspections are running under Windows in that case, and there are many Windows API calls available that will provide you with information on the current state of PC memory. The properties of this object provide the following information:

int Available { get; }

This property returns the size in bytes of available memory in the general memory heap.

int Contiguous { get; }

This property returns the size in bytes of the largest contiguous block of memory in the general heap.

int Frags { get; }

Read-only. This property returns the number of memory fragments in the general memory heap.

int Size { get; }

This property returns the overall size in bytes of the general memory heap.

int Used { get; }

This property returns the size in bytes of the amount of used general memory.

int UsedMax { get; }

This property returns the maximum general memory used ever (in bytes).

Handling Reports on Separate Threads

The .NET languages make it relatively easy to create multi-threaded programs. In general, multi-threading should be avoided unless absolutely necessary. Issues such as thread synchronization and thread deadlocks make your program considerably more complex than a single threaded implementation. That being said, there are many times when a multi-threaded approach is the best solution to performance issues. In a Visionscape .NET application, the most likely reason you would want to use multiple-threads would be to handle your report connections. C# and VB.NET provide all of the functionality you need to do this. You need to understand a couple of things:

1. In order to receive the NewReport even on a separate thread, your ReportConnection object must be instantiated in that thread.

2. If you need to display the results or images in the report, you can NOT access the controls on your Form from your worker thread, this is not allowed by .NET. You will need to create a delegate, and Invoke it from your thread in order to callback to the Form on the main thread.

Item 2 above is perhaps the most complicated issue. If you are not displaying the contents of your reports, then you do not need to worry about this. But most people want to display their results and images, so in this section we will demonstrate how you might handle the NewReport event on a separate thread, and then pass the report data back to the owning Form on the main thread.

Create the ThreadedResults Class

We will create a class that will do the work of creating our report connection on a separate thread, and handling the NewReport event. This class assumes that you've already loaded a job, and downloaded it to the device. The constructor of this class will take two arguments:

```
ISynchronizeInvoke owner:
NewReportDelegate receiveResultsDelegate
```

The Form class in C# implements the ISynchronizeInvoke interface. We can use this interface to "Invoke" a function in our main form, but the function will run in the main thread context, allowing us to access the controls on the form. The function that we will invoke is specified by the

NewReportDelegate which we will define in our class. Our class looks like this:

```
class ThreadedResults
{
    //member variables
    private VsDevice _device;
    private int _inspIndex;
    private ReportConnection _repcon;
    private Thread _thread;
    private bool _bConnected = false;
    private ISynchronizeInvoke _owner;
    public AutoResetEvent _connectComplete = new
AutoResetEvent(false);
    //define the delegate that will be used to callback to
    //owner form
    public delegate void NewReportDelegate(InspectionReport
report,
 int inspIndex);
    private NewReportDelegate _notifyOwner;
    //constructor takes a reference to the form and a ref
    //to the callback function(delegate)
    public ThreadedResults(ISynchronizeInvoke owner,
    NewReportDelegate receiveResultsDelegate)
    {
        _owner = owner;
        _notifyOwner = receiveResultsDelegate;
    }
    public bool Connect(VsDevice dev, int inspindex)
    {
        _device = dev;
        _inspIndex = inspindex;
       //create thread to handle our results
        if(_thread == null)
            _thread = new Thread(ThreadedConnect);
        else
        {
            _thread.Abort();
        }
        //start the thread
        _connectComplete.Reset();
        _thread.Start(this);
       //wait for the function to complete, so we can return
       //the status
        _connectComplete.WaitOne();
```

```
        return _bConnected;
    }
    public void Disconnect()
    {
        if(_thread != null)
            _thread.Abort();
        if(_repcon != null && _repcon.IsConnected)
            _repcon.Disconnect();
    }
    private void ThreadedConnect(object obj)
    {
        if (_repcon != null && _repcon.IsConnected)
             _repcon.Disconnect();
        else
             _repcon = new ReportConnection();
        try
        {    //Connect our report connection
            _repcon.Connect(_device, _inspIndex);
            //wire up the delegate
            _repcon.NewReport += _repcon_NewReport;
           //try to add all of the snapshot buffers to our
          //report
           //Tell the device to update it's namespace
          //information first
            _device.QueryNamespace();
            //now add all snapshot buffers to the report
            _repcon.AddSnapBuffers(_device.Namespace);
            _bConnected = true;
        }
        catch (Exception)
        {
            _bConnected = false;
        }
        //signal that connection is complete
        _connectComplete.Set();
    }
    //The NewReport event. This will be received on a
   //separate thread
    void _repcon_NewReport(object sender,
ReportConnectionEventArgs e)
    {
        if (_owner != null)
        {    //Pass the report back to our owning form by
//invoking the delegate that was passed in
            object[]args = { e.Report, _inspIndex};
            _owner.Invoke(_notifyOwner, args);
```

```
        }
    }
```

Use the ThreadedResults Class in the Form

Add the following member variables to your Form class:

```
private object _synchObject = new object();
private ThreadedResults _resultHandler;
```

We will use _synchObject for synchronization, to insure that multiple threads don't access our callback function simultaneously.

We need to create a function in our main form that matches the signature of the NewReportDelegate delegate defined in the ThreadedResultsClass. This function will be called whenever the thread has a new report to be displayed. So add the following function to your main form:

```
public void ReceiveResults(InspectionReport report, int
inspIndex)
{
    //prevent multiple threads from coming in here
    //simultaneously
    lock(_synchObject)
    {
        if(report.Images.Count > 0)
        {
            BufferDm buf = report.Images[0];
            ctlBufView.Buffer = buf;
        }
    }
}
```

Lastly, we need to instantiate and connect our ThreadedResults object. So add the following code where you are currently connecting your ReportConnection:

```
//create the ThreadedResults object,
// Form implements the ISynchronizeInvoke interface, so we
//pass "this"
//as the first argument, and our callback function as the 2nd
    _resultHandler = new ThreadedResults(this,
ReceiveResults);
    //connect, passing in a reference to the device (m_dev),
    //and the index of the inspection we are connecting to.
```

```
_resultHandler.Connect(m_dev, 1);
```

Your application will now receive the NewReport event on a separate thread, allowing you to perform time consuming tasks, and you can pass the report safely back to the main thread so that you can display your images and data.

Report Queue Connections

Note: If you are unfamiliar with the Part Queue, see the Visionscape FrontRunner User's Manual for more information.

The ReportQueueConnection object retrieves Part Queue data from a running inspection. When programming your inspection in FrontRunner, you must enable the Part Queue, set its size, and set the type of data you want to queue up. Once running, the inspection will then queue up images, results or both based on your settings. A typical scenario is where a user wishes to queue up the last 20 failed images. If you wish to retrieve this Part Queue data in your application, then you would use the ReportQueueConnection object, which is used in a manner that is very similar to the ReportConnection object. Generally, you would follow these steps:

1. Instantiate a ReportQueueConnection object, and connect it to the Inspection using the Connect method.
2. Use the Summary method to determine if there are any records in the Part Queue.
3. If any, retrieve them all using the RecordGetAll method, which returns you an InspectionReportList object, which is a list of InspectionReport objects, which we have already covered at length.

Here is an example function demonstrating how you might retrieve the Part Queue from a specified inspection on a specified device:

```
private int UploadPartQ(VsDevice dev, int inspIndex)
{
    int qSize = 0;
    //Create and connect our ReportQueueConnection object
```

```
    ReportQueueConnection qconn = new
ReportQueueConnection();
    bool bRes = qconn.Connect(dev, inspIndex);
    if(bRes)
    {
        //get the summary
        ReportQueueSummary summary = qconn.Summary();
        //are there any entries in the Q currently?
        if(summary.CurrentEntries > 0)
        {
            //yes, so upload them
            InspectionReportList replist =
qconn.RecordGetAll();
            qSize = replist.Count;
            //cycle through all of the report records
            int i = 0;
            foreach(InspectionReport report in replist)
            {
                //do something with the results...
                foreach(InspectionReportValue value in
report.Results)
                {
                    //...????....
                }

                //save all the images to disk
                foreach(BufferDm buf in report.Images)
                {
                    ++i;
                    buf.SaveImage("C:\Image" + i + ".tif",
 EnumImgFileType.ftTIF );

                }
            }
        }
    }
    return qSize; //return the number of records uploaded
}
```

Visionscape provides a control that can be used to view the contents of the InspectionReportList. The QueueView control is contained in the Visionscape.Display.Runtime.Dll library. You simply pass the InspectionReportList object to the control, and you can view all of the images and results. You could easily create your own Report Queue

viewer by adding a BufferView to a form (for image display) and any controls you wish to display the results.

ReportQueueConnection Object

Let's take a closer look at the methods and properties of this object, starting with the Connect method:

bool Connect(**Visionscape.Devices.VsDevice** *dev*, **int** *InspIndex*)

dev: The device you are connecting to.

InspIndex: Specifies the 1 based index of the inspection you want to connect to.

Connects to the specific Inspection on the given device. Returns true if connected successfully, false if not.

void Disconnect()

Disconnects from the device.

ReportQueueSummary Summary()

Retrieves a summary of the current queue state. The data is returned in a ReportQueueSummary object if successful, null is returned if the Summary could not be retrieved.

InspectionReportList RecordGetAll()

Retrieves all the records in the Part Queue. They are returned in the form of an InspectionReportList object, which is simply a list of InspectionReport objects. Each record will contain the images, results and stats from the inspection cycle in which it was entered into the Part Queue. The Part Queue is cleared after this call.

InspectionReport RecordGetAt(**int** *indexOrCycleCount*)

Returns a single Part Queue Record by index or by cycle count. The index is 0 based. If specifying a cycle count, you must pass a negative number. The record is returned as an InspectionReport object. The record is NOT removed from the Part Queue when you retrieve it with this function.

InspectionReportList RecordGetRange(**int** *first*, **int** *last*)

Retrieves all of the part queue records in the specified range. The first and last parameters are 0 based indexes into the queue. The records are removed from the queue after they are uploaded.

void RecordClearAll()

Clears the Part queue on the device without uploading it.

bool Connected { get; }

Read-only. This property returns True when connected.

int InspIndex { get; }

Read-only. This property returns the Inspection Index passed in to Connect(). Returns -1 if not connected.

VsDevice Device { get; }

Read-only. This property returns the device that the object is connected to. Returns null if no connection.

5

CHAPTER 5 I/O Capabilities

The IOConnection Object

The IOConnection object allows you to connect to any Visionscape Device for the purpose of interfacing with its I/O. You can read and write to any type of I/O point or block of points, including Physical and Virtual I/O. You can also receive transition events notifying you of I/O state changes. The IOConnection object is part of the Visionscape.Communications namespace.

Assembly Names: Visionscape.dll & Visionscape.Communications.dll
Namespace: Visionscape.Communications

To access the IOConnection object in your .NET project, add the following .NET references to your project:

```
Visionscape
Visionscape.Communications
```

Add the following statement to the top of your C# files in order to make access to this namespace easier (all sample code in this chapter assumes the following statement is present):

```
using Visionscape.Communications;
```

I/O Basics

The IOConnection object allows you to interface to both Physical I/O and Virtual I/O.

Physical I/O: Actual, physical I/O points that can be wired to an external logic controller. The amount of physical I/O that is available is determined by the hardware that you are using.

Virtual I/O: This is software I/O, that has no physical representation. There are 2048 Virtual I/O points, and these points are global. By global we mean that the Virtual I/O is shared across all Devices.

The type of I/O that is supported depends on the type of Visionscape Device you are using:

Smart Cameras: Support both Virtual I/O and Physical I/O.

GigE Systems: Support Virtual I/O only. However, if you use the Visionscape PCIe Digital I/O board, the inputs and outputs on that board will be mapped to Virtual I/O points. This means that you can use Virtual I/O points to read and write the physical I/O points on the Visionscape Digial I/O board. The I/O is mapped as follows:

Type	Digital I/O Board	Mapped to Virtual I/O Points
Inputs	I/O Points 1 - 16	161 - 176
Outputs	I/O Points 17 - 32	177 - 192

Software Systems: Support only Virtual I/O.

How to Use IOConnection

To get and set I/O values in Visionscape, you simply need to instantiate an IOConnection object, and connect it to the Device whose I/O you wish to control by calling the Connect method. Once connected, use the PointRead function to read I/O, and the PointWrite function to write I/O. If you wish to receive events notifying you of I/O transitions, then you will need to handle the IOTransition event. The following sample code demonstrates how you might use the IOConnection object:

```
//declare our global IOConnection object
private IOConnection m_IO = new IOConnection();

private void frmMain_Load(object sender, EventArgs e)
{

//Your app initialization code goes here…

     //Connect to the device like this,
     //(assume that our VsDevice variable is named 'm_dev')
     m_IO.Connect(m_dev);

     //wire up our delegate to handle the IOTransition event
     m_IO.IOTransition += m_IO_IOTransition;

       //We can now get and set IO values
       //Read the value of Virtual IO 1 (0 based index)
     bool vioState = m_IO.PointRead(VsIoType.Virtual, 0);

     //Turn Virtual IO 2 on
     //(if not type is specified, Virtual IO is default)
     m_IO.PointWrite(1,true);

     //Read the value of Physical IO 8
     bool phyState = m_IO.PointRead(VsIoType.Physical, 7);

     //Turn physical IO 9 OFF
     m_IO.PointWrite(VsIoType.Physical, 8, false);

}
//this function will pulse the specified virtual IO point,
//this could be used to generate a software trigger
private void GenerateVirtualTrigger(int ioNum)
{
    if(m_IO.IsConnected())
    {
        //turn it on
        m_IO.PointWrite(ioNum, true);
```

```
        //turn it off
        m_IO.PointWrite(ioNum, false);
    }
}

//Our Event Handler for the IOTransition event
void m_IO_IOTransition(object sender, IOConnectionEventArgs
e)
{
    //check the type of the IO
    switch(e.IoType)
    {   //dump a message displaying the io number and its
state
        case VsIoType.Virtual:
            Console.WriteLine("Virtual IO " + e.IoNum + " is
" +
(e.On ? "ON" : "OFF"));
            break;
        case VsIoType.Physical:
           Console.WriteLine("Physical IO " + e.IoNum + " is
" +
(e.On ? "ON" : "OFF"));
            break;
        default:
            break;
    }
}
```

In the above example code, we instantiated the IOConnection object in the Form_Load event, connected it to our chosen Device, and wired up a delegate to handle the IOTransition event. We then created a GenerateVirtualTrigger() function that will toggle a specified Virtual I/O point on and off. If your running inspection was set up to be triggered by a Virtual I/O point, then this routine would, in fact, generate a trigger, and cause your inspection to run for one cycle. Lastly, we showed the IOTransition event handler. This event passes you an IOConnectionEventArgs value, which contains properties to query the type, index and current state of the I/O point whose state has just changed. We simply put in some code to dump a message to the Debug window identifying the source of the transition.

Properties and Methods of IOConnection

bool Connect(Visionscape.Devices.VsDevice *dev*)

This method will connect your object to the specified VsDevice object. Returns true if successful, false if not. Once connected, you can read and write I/O values.

bool IsConnected()

Returns true if the object is currently connected to a Device, false if not.

bool Disconnect()

Disconnects the IOConnection object from the Device.

void EnableTransitions(int *ioNum*, int *ioCount*)

void DisableTransitions(int *ioNum*, int *ioCount*)

These two functions allow you to enable or disable I/O transition events for a range of Virtual I/O points.

bool PointRead(VsIoType *ioType*, int *ioNum*)

Reads the current state for the I/O of the specified type and index. The ioNum parameter is a 0 based index.

bool PointRead(int *ioNum*)

This overloaded version of PointRead will read the current state of the specified Virtual I/O value.

void PointWrite(VsIoType *ioType*, int *ioNum*, bool *bOn*)

Sets the specified I/O Point to the specified state. The type and index of the I/O point are specified by ioType and ioNum. If bOn is true, the I/O point is turned on, if it is false, the I/O point is turned off.

void PointWrite(int *ioNum*, bool *bOn*)

This overloaded version of PointWrite sets the specified Virtual I/O point to the state specified by bOn.

bool[] BlockRead(int *ioNum*, int *ioCount*)

bool[] BlockRead(VsIoType *ioType*, int *ioNum*, int *ioCount*)

ioNum: The 0 based index of the starting I/O point.
ioCount: The number of I/O points to read.
ioType: The type of I/O to read.

The BlockRead functions allow you to read a range of I/O points all at once. An array of Boolean values is returned that will hold the state of each of the I/O points. One version allows you to specify the type of I/O you want to read, while the other defaults to Virtual I/O.

void BlockWrite(int *ioNum*, bool[] *boolArray*)

void BlockWrite(VsIoType *ioType*, int *ioNum*, bool[] *boolArray*)

ioNum: The 0 based index of the starting I/O point.
boolArray: This is the array of Boolean values that will be written.
ioType: The type of I/O to be written.

The BlockWrite functions allow you to write a range of I/O points all at once. You specify the starting I/O point, and a Boolean array of values to be written. The size of the array determines how many I/O points are written. One version allows you to specify the type of I/O you want to write, while the other defaults to Virtual I/O.

int StartTrigger(int *ioNum*, int *pulseInterval*)

int StartTrigger(int *ioNum*, int *pulseInterval*, int *pulseWidth*, bool *pulseLowToHigh*)

ioNum: 0 based index of the Virtual I/O point to pulse.
pulseInterval: Time in milliseconds between pulses.
pulseWidth: Time that I/O point should stay on before turning off.
pulseLowToHigh: Set to true if you want a low to high pulse, false if you want high to low.

The StartTrigger functions can be used to pulse a specified Virtual I/O point at a specified time interval. This can be used to simulate triggers in a real-world application. The functions return a trigger ID value that you will use to stop the timer, a 0 is returned if unable to start the timer. You may start up to 4 trigger pulses simultaneously.

void StopTrigger(int *idTimer*)

Stops the timer specified by the idTimer value. This is the trigger ID value returned by the StartTrigger function.

Events

IOTransition: This event is fired when a transition has occurred on a specific I/O point.

Event Handler Function Signature:

```
void IOTransition(object sender,
IOConnectionEventArgs e)
```

The IOConnectionEventArgs parameter holds the following key properties:

VsloType IoType: This enum value identifies the type of the I/O point that changed state.

int IoNum: This is the 0 based index of the I/O point that has changed state.

bool On: Holds the current state. If true, the point is ON, if false, it is OFF.

6

CHAPTER 6 Image Display Controls

Visionscape provides several controls that make it easy to display the images from your inspections. In this chapter we will cover the controls contained with the Visionscap.Display.Image.DLL library.

Assembly Name: Visionscape.Display.Image

Namespace: Visionscape.Display

BufferView: A simple control to display an image. Also provides scrolling and zooming capabilities.

Filmstrip: A control to display a running “filmstrip” of the last *n* images.

Adding the Controls to the Visual Studio Toolbox

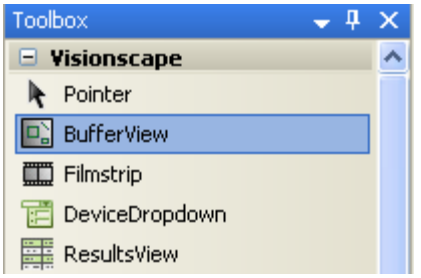
If Visual Studio was installed on your PC at the time when you installed Visionscape, then all of the visual controls should have been added to a “Visionscape” tab in the Visual Studio Toolbox. If you installed Visual Studio after installing Visionscape, then you can install the controls to your Toolbox by running the VsToolboxInstall.exe utility:

Start -> Programs -> Microscan Visionscape -> Tools -> Install Controls to Visual Studio Toolbox

This will install all Visionscape controls to the Toolbox. If you prefer to only install the controls that you need, you can always install the controls manually by simply dragging the DLL and dropping it on the Toolbox.

Assuming you have installed Visionscape to the folder C:\Vscape, go to the folder C:\Vscape\Assembly\Display, and find the file visionscape.display.image.dll:

The BufferView Control



The BufferView control is used to display images. This is perhaps the most commonly used control, and it is very easy to use. The BufferView control displays BufferDm objects. In Visionscape, the BufferDm (Buffer Datum) object is used to represent an image. To dispay an image in the BufferView control, you simply need to assign a BufferDm to its Buffer property. To add this control to your form, simply open the Visionscape tab on the Visual Studio Toolbox, select BufferView, and drag it to your Form. In chapter 4, we demonstrated how to setup a ReportConnection to receive images via the NewReport event. Following is that sample where we retrieve the 1st image contained in the uploaded InspectionReport, and then display that image in a BufferView control named ctlBufView:

```
//The Event handler for the NewReport event
void m_RepCon_NewReport(object sender,
ReportConnectionEventArgs e)
{
  //get the Inspection report from the
  //ReportConnectionEventArgs object
    InspectionReport report = e.Report;



    //does our report contain any images?
    if(report.Images.Count > 0)
    {
```

```
        //extract the image as a BufferDm,
        //and set it into the Buffer View
        ctlBufView.Buffer = (BufferDm)report.Images[0];
    }
}
```

That's all there is to it. The BufferView control also provides methods to control the scrolling and zooming of the image. Following is a complete list of all methods and properties provided by the BufferView control:

bool AutoZoom { set; get; }

Gets/Sets the autozoom state of the control. When true, the control will automatically scale the image to fit within its bounds. Set this property to false when you want to control the zooming yourself.

long ZoomIn( )

Zooms the image in by increasing the scale by a factor of 2. The resulting scale factor is returned. Has no effect if AutoZoom is true.

long ZoomOut( )

Zooms the image out by decreasing the scale by a factor of 2. The resulting scale factor is returned. Has no effect if AutoZoom is true.

long ZoomTo(**int** *scaleNumerator*, **int** *scaleDenominator*)

Zooms the image to the specified scale factor. The scale factor is specified by passing the numerator and denominator of the value you wish to be applied. Has no effect if AutoZoom is true.

```
//turn off autozoom
ctlBufView.AutoZoom = false;
//zoom to 1/3 the size
ctlBufView.ZoomTo(1, 3);
//zoom to 3x the size
ctlBufView.ZoomTo(3, 1);
```

double ZoomFactor { set; get; }

Gets/sets the current zoom factor.

long ScrollTo(**int** x, **int** y)

Scrolls the image such that image is displayed with the input point (x,y) at the upper left corner of the view.

int ScrollPositionX { set; get; }

int ScrollPositionY { set; get; }

These properties get/set the current X and Y scroll position. This is the position of the image at the upper left corner of the view.

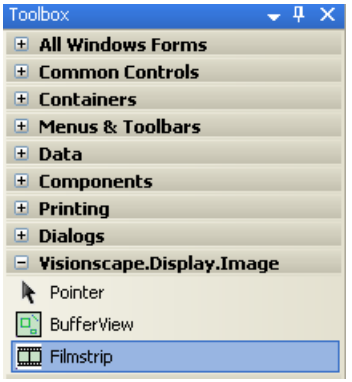
bool ShowStatusBar { set; get; }

When true, the status bar at the bottom of the control is shown. If false, it is hidden.

long OpenImage(**string** *strFilename*)

Opens the specified image file and displays it in the control. This only works with images of type TIFF and Bitmap.

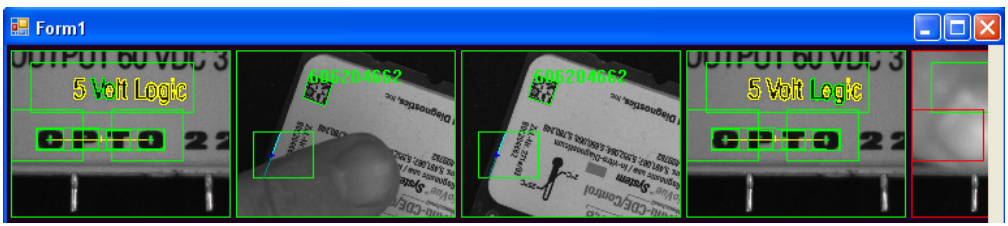
The Filmstrip Control



The Filmstrip control can also be used to display images. It differs from the BufferView in that it displays a running “filmstrip” of the last n images that were displayed in the control. The images in the Filmstrip are always autosized, there is no ability to scroll or zoom the images as there is in the BufferView control. The layout of the filmstip depends upon the

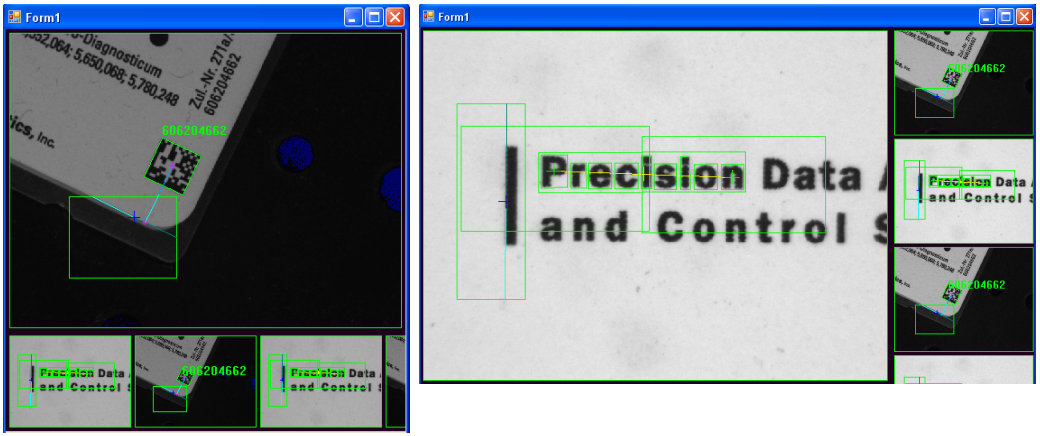
dimensions of the control. If you want a vertical filmstrip, make your control tall and thin, as shown below:

If you want a horizontal filmstrip, make your control short and wide, as shown below:



If you size the control to be more “square”, then it will display a large version of the most recent image with a filmstrip running across either the

right or bottom edge of the screen, depending on the height to width ratio of the control.



Just like the BufferView control, the Filmstrip displays BufferDm objects. To add images to the Filmstrip control you simply call the NewBufferDm method. If we added a FilmStrip control to our form named ctlFilmStrip, we could modify our earlier BufferView example to display images in the FilmStrip like this:

```
//The Event handler for the NewReport event
void m_RepCon_NewReport(object sender,
ReportConnectionEventArgs e)

{

  //get the Inspection report from the
  //ReportConnectionEventArgs object
     InspectionReport report = e.Report;

    //does our report contain any images?
     if(report.Images.Count > 0)

    {

        //extract the image as a BufferDm,
         //and add it to the FilmStrip control
        ctlFilmStrip.NewBufferDm((BufferDm)report.
        Images[0]);
    }
}
```

There is also an overloaded version of the NewBufferDm method that takes a bool value specifying the pass/fail state of the image. The

Filmstrip will then display a green border around images that pass, and a red border around those that fail. The pass fail state can easily be extracted from the report data, so our earlier example could be modified to look like this instead:

```
ctlFilmStrip.NewBufferDm((BufferDm)report.Images[0],
                                    report.InspectionStats.Passed);
```

Properties and Methods of Filmstrip

void NewBufferDm(Visionscape.Steps.BufferDm *buf*)

Displays the specified buffer datum in the filmstrip. All previous images are shifted one position in the filmstrip.

void NewBufferDm(Visionscape.Steps.BufferDm *buf*, **bool** *bPassed*)

This overloaded version adds the bPassed parameter. When bPassed is true, the image is drawn with a green border, if it is false, the image is drawn with a red border.

CHAPTER 7 Device Selection Controls

This chapter covers the Visionscape.Display.Devices assembly. This assembly provides controls that present you with a list of available Visionscape Devices, allowing you to select the Device your application will connect to.

Assembly Name: Visionscape.Display.Devices.dll

Namespace: Visionscape.Display.Devices

DeviceDropdown: A dropdown list box with a list of all available Visionscape Devices.

ToolStripDeviceDropdown: A version of the DeviceDropdown control that can be used in a ToolStrip control. Note that this control does not show up in the Visual Studio Toolbox.

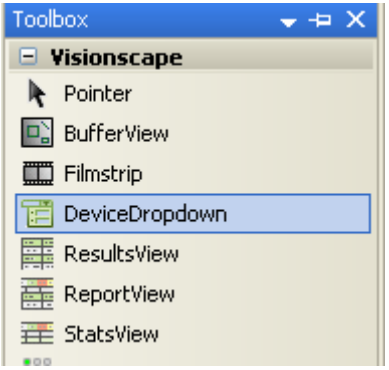
Adding the Controls to the Visual Studio Toolbox

If Visual Studio was installed on your PC at the time when you installed Visionscape, then all of the visual controls should have been added to a “Visionscape” tab in the Visual Studio Toolbox. If you installed Visual Studio after installing Visionscape, then you can install the controls to your Toolbox by running the VsToolboxInstall.exe utility:

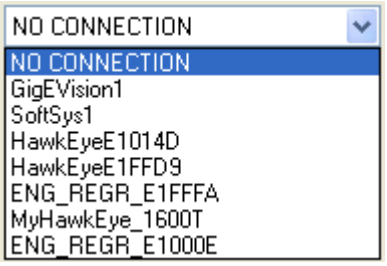
Start -> Programs -> Microscan Visionscape -> Tools -> Install Controls to Visual Studio Toolbox

This will install all Visionscape controls to the Toolbox. If you prefer to only install the controls that you need, you can always install the controls manually by simply dragging the DLL and dropping it on the Toolbox. Assuming you have installed Visionscape to the folder C:\Vscape, go to the folder C:\Vscape\Assembly\Display, and find the file visionscape.display.devices.dll:

The DeviceDropdown Control



The DeviceDropdown control provides a simple dropdown combo box that is populated with a list of available Visionscape Devices. This is useful when you have multiple Visionscape Devices in your PC or on your Network, and you need to provide a way for the user to select the Device that they wish to connect to. Simply drop this control onto a form, an it will automatically detect all of the Visionscape Devices that are present in your PC and on your network, and add them to the list. A populated DeviceDropdown control might look like this if you had many cameras on your network:



The DeviceSelected event is raised when the user selects a device in the list. Several properties, methods and events are provided to control the behavior of the DeviceDropdown control.

Events:

DeviceSelected: Fired when the user selects a device in the list.

Event Handler Function Signature:

```
void deviceDropdown1_DeviceSelected(VsDevice dev)
dev: The VsDevice object selected by the user.
```

SelectedDeviceLost: Fired when the device that is currently selected is no longer detected. This would typically happen when a smart camera is unplugged from the network.

Event Handler Function Signature:

```
void deviceDropdown1_SelectedDeviceLost()
```

FilterDevice: Fired before a Device is added to the list. You can apply your own logic to prevent certain devices from being added to the list by setting the bApproved parameter to false.

Event Handler Function Signature:

```
void deviceDropdown1_FilterDevice(VsDevice dev,
                                  ref bool bApproved)
dev: The device to be added to the list.
bApproved: Set to false if you want to remove this device
from the list.
```

Properties:

Visionscape.Devices.VsDevice Device { set; get; }

Gets/Sets the currently selected device in the list.

bool AutoConnect { set; get; }

Gets/Sets the current AutoConnect State. If true, whenever the selected device changes, the VsCoordinator OnDeviceFocus event will fire. Other controls can look for this event and "Auto connect" to the selected device.

int GroupId { set; get; }

Gets/Sets the group ID assigned to this control. This is used along with the AutoConnect feature. This group ID will be included in the

OnDeviceFocus event that is fired by the VsCoordinator when AutoConnect is active. This allows you to use multiple DeviceDropDown controls, with different sets of controls “AutoConnecting” to each.

int ListIndex { set; get; }

Gets/Sets the index of the currently selected Device in the list.

int ListCount { get; }

Returns the current number of devices in the list.

Methods:

string DeviceName(**int** *index*)

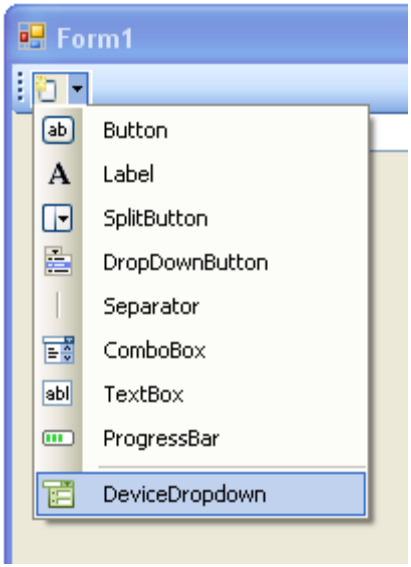
Returns the name of the device at the specified index in the list.

void ResetList( )

Commands the control to clear its list and rebuild it.

The ToolStripDeviceDropdown Control

The ToolStripDeviceDropdown control provides the same functionality as the DeviceDropDown control, however it is intended for use only in ToolStrips. This control does NOT show up in the Visual Studio Toolbox, as it can not be dropped onto a form, it can only be used in a ToolStrip control. When adding a control to your ToolStrip, you will see the DeviceDropdown as one of the options:



All functionality is identical to the DeviceDropdown control. Refer to the DeviceDropdown documentation for details.

CHAPTER 8 Report Display Controls

This chapter covers the Visionscape.Display.Reporting assembly. This assembly contains controls that can be used to display the data from your running inspections as well as controls that can be used to view and manipulate I/O.

Assembly Name: Visionscape.Display.Reporting.dll

Namespace: Visionscape.Display.Reporting

ResultsView: Displays inspection result data in a grid format.

StatsView: Displays the runtime stats from an inspection.

ReportView: This control combines the ResultsView and StatsView into one control.

IOView: Allows you to both view and set the current state of a range of I/O points.

IOTriggerView: Provides the ability to generate Virtual I/O trigger pulses.

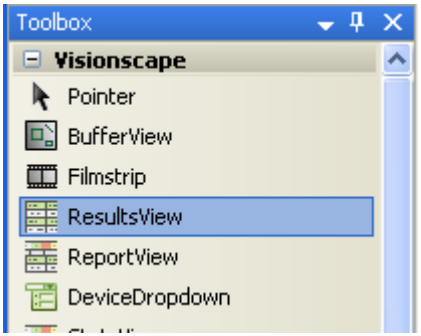
Adding the Controls to the Visual Studio Toolbox

If Visual Studio was installed on your PC at the time when you installed Visionscape, then all of the visual controls should have been added to a “Visionscape” tab in the Visual Studio Toolbox. If you installed Visual Studio after installing Visionscape, then you can install the controls to your Toolbox by running the VsToolboxInstall.exe utility:

Start -> Programs -> Microscan Visionscape -> Tools -> Install Controls to Visual Studio Toolbox

This will install all Visionscape controls to the Toolbox. If you prefer to only install the controls that you need, you can always install the controls manually by simply dragging the DLL and dropping it on the Toolbox. Assuming you have installed Visionscape to the folder C:\Vscape, go to the folder C:\Vscape\Assembly\Display, and find the file visionscape.display.reporting.dll.

The ResultsView Control



The ResultsView control displays the result data contained in an InspectionReport object, or to be more specific, it displays the contents of the InspectionReport.Results property. The data is displayed in a grid format. To use this control, you simply need to set it’s Report property to the InspectionReport whose results you wish to display. This would typically be done when responding to the NewReport event of a ReportConnection (Refer to Chapter 4 for a full description of ReportConnections and the InspectionReport object). In the following example, we demonstrate handling the NewReport event, retrieving the

InspectionReport, and then displaying it's results in a ResultsView control named resultsView1:

```
//Event handler for the NewReport event of ReportConnection
void m_RepCon_NewReport(object sender,
ReportConnectionEventArgs e)
{


  //get the Inspection report from the
  //ReportConnectionEventArgs object
  InspectionReport report = e.Report;
  //Pass the report to our ResultsView control
  resultsView1.Report = report;
}
```

After the Report property is set, the control will look something like this:

Fast Edge.Status	True				
Fast Edge.Edge Point	196.602	198.000	1.628	1.000	
Fast Edge.Edge Line	-.998	-.057	207.654		
Blob Tool.Blob Tree	View BlobTree				

	XCenter	YCenter	AreaPix	Color
Blob1	368.864	263.591	11.000	-1
Blob2	358.577	263.423	13.000	-1
Blob3	348.346	262.346	13.000	-1
Blob4	337.577	261.577	13.000	-1
Blob5	327.346	261.346	13.000	-1
Blob6	317.115	260.500	13.000	-1
Blob7	452.864	258.591	11.000	-1
Blob8	442.438	257.750	16.000	-1
Blob9	431.962	256.962	13.000	-1
Blob10	369.577	253.423	13.000	-1
Blob11	359.136	252.500	11.000	-1
Blob12	348.688	252.063	16.000	-1
Blob13	338.500	251.500	14.000	-1
Blob14	328.192	250.731	13.000	-1
Blob15	317.577	249.885	13.000	-1

BlobFilter.Center Point	365.461	193.868	-.012	1.000
BlobFilter.Top Point	270.500	118.500	.000	1.000
BlobFilter.Bottom Point	462.500	266.500	.000	1.000
BlobFilter.Area	26213.000			
BlobFilter.Roundness	.735			

You can clear the control by setting the Report property to null.

AutoSizing Behavior:

The ResultsView control will automatically adjust it's height to encompass the list of results it is displaying. When you need to display the results from multiple inspections, you can insert multiple ResultsView controls into a .NET container control (Panel, SplitContainer, etc), set the Dock property for each to "Top", and the controls will be autosized and positioned tightly up against each other. This frees you from having to right tedious sizing and positioning code.

Properties:

Visionscape.Communications.InspectionReport Report { set; get; }

You display results in the control by setting this property to the InspectionReport object you wish to display. Set this property to null if you want to clear the displayed results.

int PrecisionPix { set; get; }

Gets/Sets the number of decimal places used when displaying floating point values that are in pixel units.

int PrecisionWorld { set; get; }

Gets/Sets the number of decimal places used when displaying floating point values that are in world units.

bool GridAutoSize { set; get; }

Set to true if you want the width of the grid cells to automatically be expanded to fit the displayed data. If you set to false, the grid cells to not change widths.

bool CalibratedValuesEnabled { set; get; }

Set to true if you want the result data to be displayed in calibrated units. This will have no effect if the inspection that produced the InspectionReport has not been calibrated.

int NumResults { get; }

The total number of results being displayed in the control.

int NumStatsDms { get; }

The total number of Statistic Datums being displayed in the control.

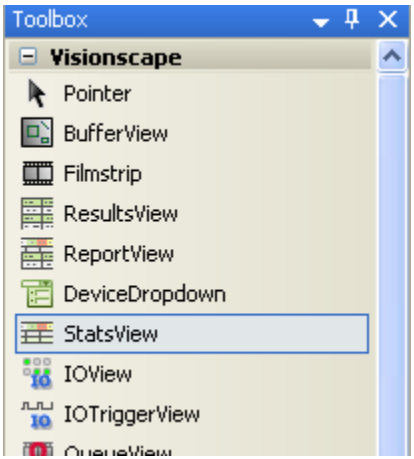
int NumTimerDms { get; }

The total number of Time Datums being displayed in the control.

Methods:

The ResultsView control has no methods.

The StatsView Control



The StatsView control displays common runtime information from a running inspection such as Cycle Time, Process Time, Number of Image buffers in use, etc. This is all information that is included in an InspectionReport object, and just as with the ResultsView object, you will update the display of this control by setting it's Report property to an InspectionReport. This would typically be done when responding to the NewReport event of a ReportConnection (Refer to Chapter 4 for a full description of ReportConnections and the InspectionReport object). In the following example, we demonstrate handling the NewReport event, retrieving the InspectionReport, and then displaying it's stats in a StatsView control named statsView1:

```
void m_RepCon_NewReport(object sender,
ReportConnectionEventArgs e)
{
  //get the Inspection report from the
  //ReportConnectionEventArgs object
    InspectionReport report = e.Report;

  //Pass the report to our StatsView control
    statsView1.Report = report;
}
```

Once updated, the StatsView control would look something like this:

Cycle	151	Cyc Worst	9895	Process	4	Process W...	5
PPM	397	PPM Worst	6	Draw	0	Idle	171
Buffers	2 of 16 used (12%)			Overruns	None		

Properties:

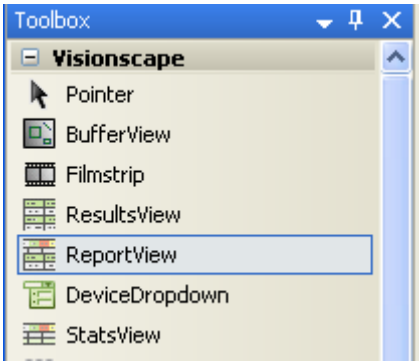
Visionscape.Communications.InspectionReport Report { set; get; }

Set this property to an InspectionReport object, and the runtime stats of that object will be displayed in the control. Set this property to null to clear the contents.

Methods:

The StatsView control has no methods.

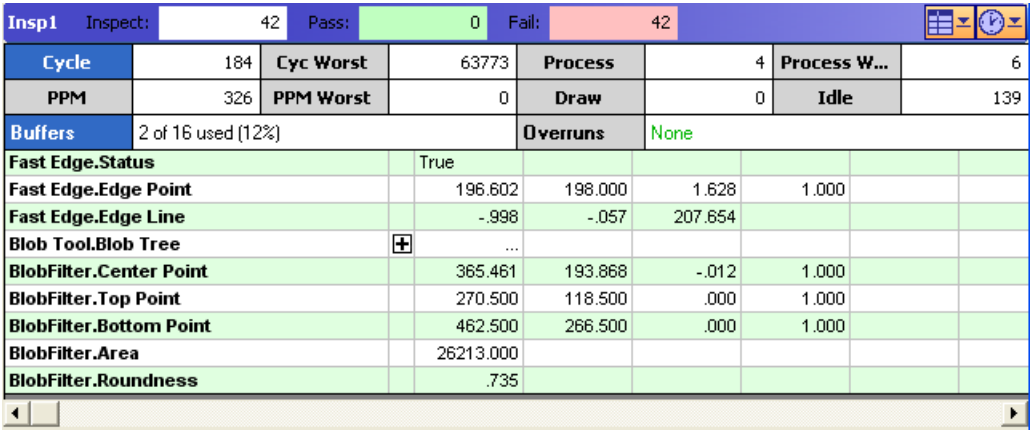
The ReportView Control:



The ReportView control combines the ResultsView and StatsView controls into a single control, and adds a tool bar at the top to display the inspection counts, and to allow you to show or hide either the results or stats. So this control is intended to display all of the important report data in a single control. You update the dislay of this control by setting it's Report property to an InspectionReport. This would typically be done when responding to the NewReport event of a ReportConnection (Refer to Chapter 4 for a full description of ReportConnections and the InspectionReport object). In the following example, we demonstrate handling the NewReport event, retrieving the InspectionReport, and then displaying all of it's relevant data in a ReportView control named reportView1:

```
void m_RepCon_NewReport(object sender,
ReportConnectionEventArgs e)
{
  //get the Inspection report from the
  //ReportConnectionEventArgs object
    InspectionReport report = e.Report;

    //Pass the report to our ReportView object
    reportView1.Report = report;
}
```

Once updated, the ReportView control would look something like this:



The toolbar at the top displays the current inspection counts (inspected, passed and rejected), and also provides two buttons that can be used to show/hide the StatsView or ResultsView.

AutoSizing Behavior:

The ReportView control will automatically adjust it's height to encompass the list of results it is displaying, as well as the controls that your user has chosen to show or hide. When you need to display the reports from multiple inspections, you can insert multiple ReportView controls into a .NET container control (Panel, SplitContainer, etc), set the Dock property for each to "Top", and the controls will be autosized and positioned tightly up against each other. If your user shows or hides the stats or results in one of the controls, the height of that conrol will automatically update, and the positions of all other controls will then automatically adjust so that they remain docked up against each other. This frees you from writing a lot of tedious sizing and positioning logic.

Properties:

Visionscape.Communications.InspectionReport Report { set; get; }

Update the display of the control by setting this property to an InspectionReport object. The control will display the inspection counts,

stats and results that are embedded in the InspectionReport. Set this property to null if you want to clear the report display.

int PrecisionPix { set; get; }

Gets/Sets the number of decimal places used when displaying floating point values that are in pixel units.

int PrecisionWorld { set; get; }

Gets/Sets the number of decimal places used when displaying floating point values that are in world units.

bool CalibratedValuesEnabled { set; get; }

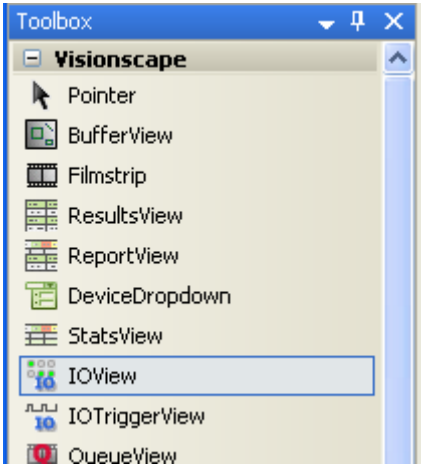
When true, results will be displayed in calibrated units. This property has no effect if the inspection that produced the InspectionReport has not been calibrated.

Methods:

void ClearCounts()

Sets the displayed counter values to 0. Understand that this does not clear the counts on your Visionscape Device, it only updates the display of the control. If you want to clear the result and stats data as well, set the Report property to null.

The IOView Control



The IOView control allows you to view the current state of a range of Virtual I/O points for a particular device. Each point is represented with a button, so you can also toggle the state of each I/O point. To use the IOView control:

- Call the Connect() method, and pass in a reference to the Visionscape Device whose I/O you want to view.
- Set the I/OFirst property to the index of the first I/O point you want to view.
- Set the I/OCount property to the number of I/O points you want to view.

In this example, assume we want to view Virtual I/O Points 10 – 14 on the device represented by the variable _device:

```
ioView1.Connect(_device);
ioView1.IOFirst = 10;
ioView1.IOCount = 5;
```

With the above settings, the control would look something like this:

In the above example, Virtual I/O points 11 and 13 are ON, while the other are off. If you prefer to use different colors to signify the ON and OFF states, you can modify them via the control properties.

Properties:

int IOFirst { set; get; }

Gets/Sets the 1 based index of the first Virtual I/O point to be displayed in the control.

int IOCount { set; get; }

Gets/Sets the number of I/O points to be displayed in the control. This property works together with the I/OFirst property to specify the range of I/O points to be displayed in the control.

Visionscape.Communications.IOConnection IOConnection { get; }

Returns a reference to the internal IOConnection object used by the control. Refer to chapter 5 for a description of the IOConnection object.

Color ColorOn { set; get; }

Gets/Sets the color that is displayed when an I/O point is ON.

Color ColorOff { set; get; }

Gets/Sets the color that is displayed when an I/O point is OFF.

Color ColorFontOn { set; get; }

Gets/Sets the text color that is used when an I/O point is ON.

Color ColorFontOff { set; get; }

Gets/Sets the text color that is used when an I/O point is OFF.

Color ColorFontDisabled { set; get; }

Gets/Sets the text color that is used when an I/O point is disabled.

Color ColorDisabled { set; get; }

Gets/Sets the color that is used when an I/O point is disabled.

int ButtonWidth { set; get; }

Gets/Sets the width in pixels of the I/O buttons.

int ButtonHeight { set; get; }

Gets/Sets the height in pixels of the I/O buttons.

Font ButtonFont { set; get; }

Gets/Sets the font used to display the button text.

Padding ButtonMargin { set; get; }

Gets/Sets the amount of space between buttons.

bool IsConnected { get; }

Returns true if the control is currently connected to a Device, false if not.

Methods:

bool Connect(**Visionscape.Devices.VsDevice** *dev*)

dev: The device to connect to.

Connects the control to the specified device. Returns true if successful. The control is not functional until it has been connected.

void SetIOPoint(**int** *ioNum*, **bool** *bOn*)

ioNum: The 1 based index of the I/O point.

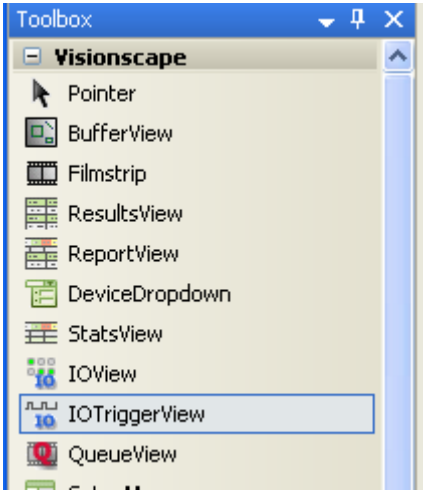
bOn: If true, the I/O point is turned ON, if false, it is turned OFF.

Sets the specified Virtual I/O point either ON or OFF.

void Disconnect()

Disconnects the control from the device. The control will be non-functional after this call completes.

The IOTriggerView Control



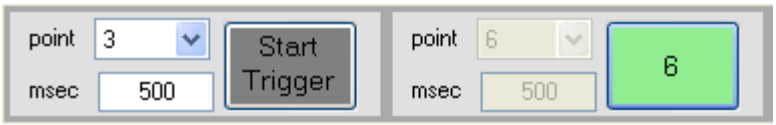
The IOTriggerView control provides one or more sub Trigger controls that allow you to pulse a Virtual I/O point at a specified interval. You can use this control to simulate triggers to your Inspections. To use the control:

- Set the NumTriggers property to the number of trigger pulses you want to generate.
- Call the Connect method, passing in a reference to the Device you want to generate triggers to.

The following example demonstrates how to configure the control for 2 triggers, and then connects it to a device represented by the VsDevice variable _device:

```
ioTriggerView1.NumTriggers = 2;
ioTriggerView1.Connect(_device);
```

Once connected, you use the combo box to select the Virtual I/O point you want to pulse, and then enter the time in milliseconds between triggers in the text box. Press the “Start Trigger” button to start the trigger pulses, press it again to shut the pulses off. Below is an example of the control set up to pulse Virtual I/O point’s 3 and 6.



Properties:

int NumTriggers { set; get; }

Gets/Sets the number of trigger controls to be displayed in the control.

Visionscape.Communications.IOConnection IOConnection { get; }

Returns a reference to the internal IOConnection object used by the control. Refer to chapter 5 for a description of the IOConnection object.

string CanStartText { set; get; }

Gets/Sets the text that is displayed in the I/O button when a trigger can be started.

string NeedPointText { set; get; }

Gets/Sets the text that is displayed in the I/O button when an I/O point has not yet been selected.

string NotConnectedText { set; get; }

Gets/Sets the text that is displayed in the I/O button when the control is not connected to a Device.

Color ColorOn { set; get; }

Gets/Sets the color that is displayed when an I/O point is ON.

Color ColorOff { set; get; }

Gets/Sets the color that is displayed when an I/O point is OFF.

Color ColorFontOn { set; get; }

Gets/Sets the text color that is used when an I/O point is ON.

Color ColorFontOff { set; get; }

Gets/Sets the text color that is used when an I/O point is OFF.

Color ColorFontDisabled { set; get; }

Gets/Sets the text color that is used when an I/O point is disabled.

Color ColorDisabled { set; get; }

Gets/Sets the color that is used when an I/O point is disabled.

Font ButtonFontTriggering { set; get; }

Gets/Sets the font used to draw button text when the control is triggering.

Font ButtonFont { set; get; }

Gets/Sets the font used to display the button text.

Padding ButtonMargin { set; get; }

Gets/Sets the amount of space between trigger controls.

Methods:

bool Connect(**Visionscape.Devices.VsDevice** *dev*)

dev: The device to connect to.

Connects the control to the specified device. Returns true if successful. The control is not functional until it has been connected.

void Disconnect()

Disconnects the control from the Device.

bool IsConnected()

Returns true if the control is currently connected to a device.

void UpdateAll()

Forces the control to update all of the trigger controls.

9

CHAPTER 9 Runtime Utility Controls

This chapter covers the Visionscape.Display.Runtime assembly. This assembly contains controls that are intended to provide useful features while your inspections are running.

Assembly Name: Visionscape.Display.Runtime.dll

Namespace: Visionscape.Display.Runtime

QueueView: Allows you to view all of the images and results in an InspetionReportList.

Adding the Controls to the Visual Studio Toolbox

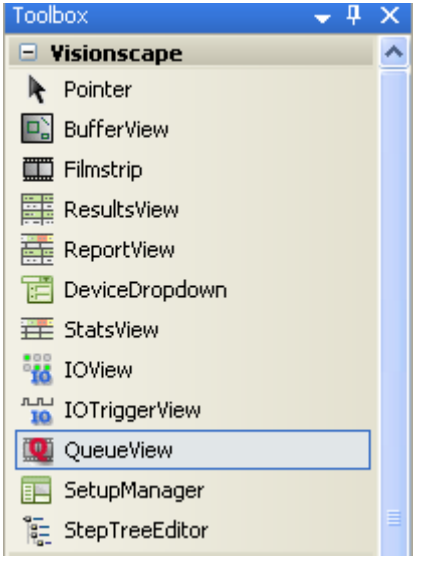
If Visual Studio was installed on your PC at the time when you installed Visionscape, then all of the visual controls should have been added to a “Visionscape” tab in the Visual Studio Toolbox. If you installed Visual Studio after installing Visionscape, then you can install the controls to your Toolbox by running the VsToolboxInstall.exe utility:

Start -> Programs -> Microscan Visionscape -> Tools -> Install Controls to Visual Studio Toolbox

This will install all Visionscape controls to the Toolbox. If you prefer to only install the controls that you need, you can always install the controls manually by simply dragging the DLL and dropping it on the Toolbox. Assuming you have installed Visionscape to the folder C:\Vscape, go to

the folder C:\Vscape\Assembly\Display, and find the file visionscape.display.runtime.dll.

The QueueView Control



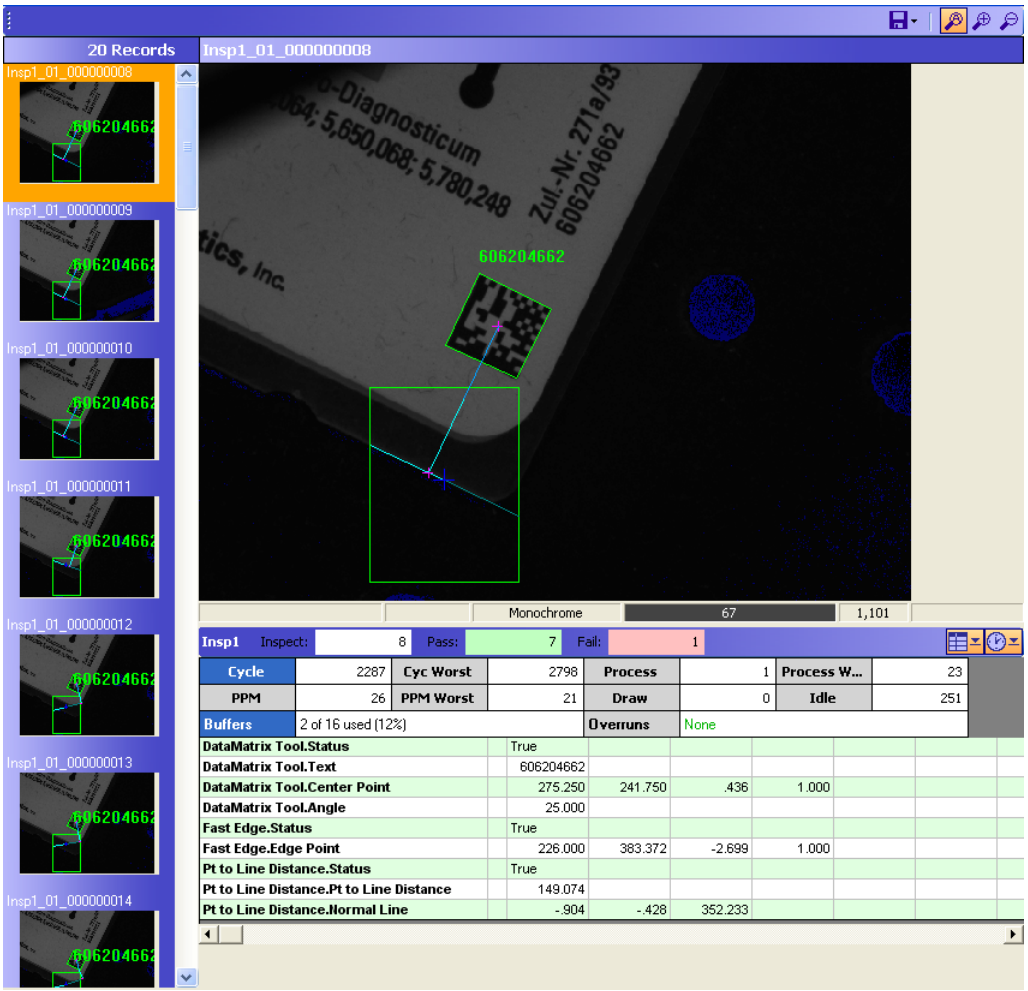
The QueueView control allows you to view the contents of a list of InspectionReport objects. The Inspection Step in Visionscape has a “Part Queue” feature that when enabled, will maintain a list of the last n cycles of inspection data (images, results and stats). The Part Queue data can be uploaded from a running inspection using the ReportQueueConnection object, and then displayed in this control. Refer to chapter 4 for details on the ReportQueueConnectin object. To use this control:

- Upload the Part Queue data from a running inspection, this will produce an InspectionReportList object.
- Set the QueueView control’s Reports property to the InspectionReortList object.

In the following example, we demonstrate how you might upload Part Queue data, and then display it in the QueueView control named queueView1:

```
//Upload the Queue from the specified inspection on the
//specified device, and display in a QueueView control
private int UploadAndDisplayPartQ(VsDevice dev, int
inspIndex)
{
    int qSize = 0;

    //Create and connect our ReportQueueConnection object
    ReportQueueConnection qconn = new
     ReportQueueConnection();
    bool bRes = qconn.Connect(dev, inspIndex);

    //if successfully conected...
    if(bRes)
    {
        //get the summary
        ReportQueueSummary summary = qconn.Summary();
        //are there any entries in the Q currently?
        if(summary != null && summary.CurrentEntries > 0)
        {
            //Yes, so go ahead and upload the Queue.
            //Retrieve the entire contents of the Queue
            InspectionReportList reports =
              qconn.RecordGetAll();

            //display the contents in our QueueView control
            queueView1.Reports = reports;

            qSize = reports.Count;
        }
    }
    return qSize; //return the number of records uploaded
}
```

The QueueView would look something like this when records are being displayed:



A scrollable list of thumbnail images is shown on the left side of the control. These represent all of the records in the Queue, simply click on the image in the list, and the full sized image will be displayed in the top right of the control, and the counts, stats, and results for that record will be displayed at the bottom-right of the control. The toolbar provides zooming options as well as options that allow you to save the currently selected image, or all images in the Queue.

Properties:

Visionscape.Communications.InspectionReportList Reports { set; get; }

You display the contents of a Part Queue by assigning this property to an InspectionReportList object. This would typically be uploaded via a ReportQueueConnection object (see previous example). Set this property to null if you want to clear the displayed records.

int SnapIndex { set; get; }

Gets/Sets the index of the Snapshot whose images are being displayed in the control. This is only relevant when the Inspection contains more than one Snapshot.

bool ShowToolbar { set; get; }

Shows/Hides the toolbar at the top of the control.

Methods:

bool SaveAllImages(**string** *strFolder*,
Visionscape.Steps.EnumImgFileType *fType*, **bool** *bDisplayFirstFileName*)

strFolder: The path to the folder where the images will be saved.

fType: Specifies the format in which the images should be saved. TIFF or BMP.

bDisplayFirstFileName: If true, a dialog will pop-up to display the file name of the first image. Saves all of the images in the InspectionReportList to disk. File names are assigned automatically. File Name format is:

InspectionSymbolicName_snapshotindex_cyclecount

CHAPTER 10

Setup Mode Controls

This chapter covers the Visionscape.Display.Setup assembly. The controls in this assembly would be used when you want to provide a “Setup Mode” capability in your application. This assembly provides powerful controls that allow you to edit your Jobs, adjust camera settings and vision tool settings, and to “tryout” your vision programs for test and debug purposes.

Assembly Name: Visionscape.Display.Setup.dll

Namespace: Visionscape.Display.Setup

SetupManager: Allows you to view images from your camera(s), adjust vision tool postions and settings, and provides the ability to “Tryout” your inspections, allowing you to debug inspection issues and improve peformance.

StepTreeEditor: Provides a Tree view of your vision Job. Allows you to edit the job by adding and removing Steps. This control is also embedded in the SetupManager control.

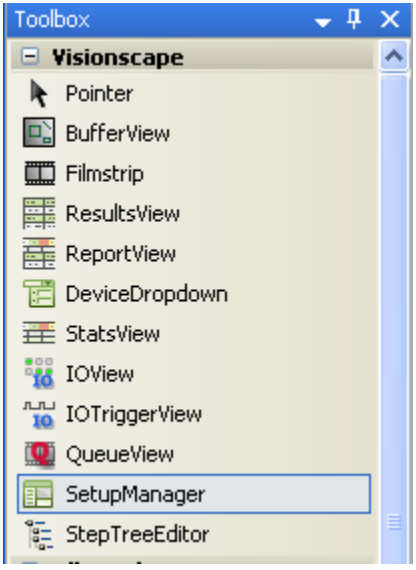
Adding the Controls to the Visual Studio Toolbox

If Visual Studio was installed on your PC at the time when you installed Visionscape, then all of the visual controls should have been added to a “Visionscape” tab in the Visual Studio Toolbox. If you installed Visual Studio after installing Visionscape, then you can install the controls to your Toolbox by running the VsToolboxInstall.exe utility:

Start -> Programs -> Microscan Visionscape -> Tools -> Install Controls to Visual Studio Toolbox

This will install all Visionscape controls to the Toolbox. If you prefer to only install the controls that you need, you can always install the controls manually by simply dragging the DLL and dropping it on the Toolbox. Assuming you have installed Visionscape to the folder C:\Vscape, go to the folder C:\Vscape\Assembly\Display, and find the file visionscape.display.setup.dll.

The Setup Manager Control



The SetupManager control provides full Setup Mode capabilities in a single control. This is by far the most powerful control we offer in VsKit.net. With SetupManager, you can acquire single images or live video, you can adjust your Vision tool ROIs and parameters, and you can

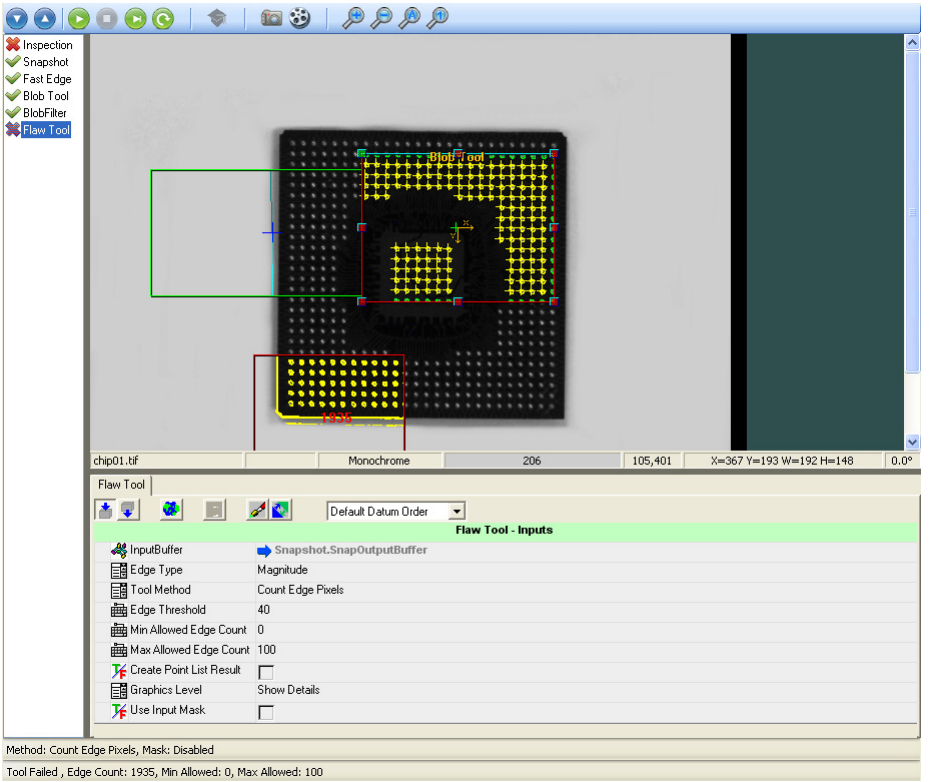
run your inspections in a “Tryout mode” which will show you the pass/fail status for each Step. You can tryout your entire inspection, or just a single step. To use the SetupManager control you must connect its RootStep property to a Step in your Job. You would typically connect it to an Inspection Step, but you can also connect it to a VisionSystem Step or a Snapshot. In the following example we will demonstrate connecting the SetupManager control to the first inspection Step in a Job. Specifically, the example does the following:

- In the Load event of our main form, we instantiate a VsCoordinator, and then wait for the Device named “GigeVision1” to be discovered.
- The OnDeviceFocusEventHandler function will be called when the Device is discovered. At this time, we will load an AVP file.
- We connect the Job to the GigE System by downloading it to the device.
- We find the first Inspection Step in the Job.
- We connect the SetupManager by setting its RootStep property to the Inspection Step.

```
private void frmMain_Load(object sender, EventArgs e)
{
    //instantiate our coordinator object
    _coordinator = new VsCoordinator();

    //wire up our event handler to the OnDeviceFocus event
    _coordinator.OnDeviceFocus += OnDeviceFocusEventHandler;
    //tell coordinator to fire the OnDeviceFocus event when
    //the device GigEVision1 is discovered
    _coordinator.DeviceFocusSetOnDiscovery("GigEVision1", -1);
}
private void OnDeviceFocusEventHandler(VsDevice objDevice)
{
    //We've discovered our Device....
    _device = objDevice;
    //Load our job
    _job.Load("C:\\Vscape\\Jobs\\GigETest.avp");
    //download the job to our device, so it is ready to run
    _device.Download(_job, true);

    //find the first inspection step in the Job
```

```
        Step insp = _job.FindByType("Step.Inspection");
        //connect Setup Manager
        ctlSetup.RootStep = insp;
}
```

The SetupManager would look something like this when connected:



Setup Manager Components:

By default, the Setup Manager is made up of 4 major components which we will describe here:

Toolbar:

The buttons on the toolbar allow you to acquire single images and live video, allow you to train Steps, and provide various tryout functions. All of the buttons have a corresponding public function exposed by SetupManager, so you can hide the toolbar and create your own. The meaning of each button is as follows:

Wizard Next: Pressing this button will run the current Step, and then move the selection to the next Step. By repeatedly clicking this button, you can step through your inspection one vision tool at a time.

Wizard Previous: Pressing this button moves the selection to the previous button in the list.

Tryout Start: Press this button to start a Tryout of your inspection. This will run your inspection for one cycle, and then the tryout will end. Each Step in the Job will be run in order, and it's status will be updated in the Setup Step List on the left side of the control. If a Step passes, a green check mark is displayed next to it's name, if it fails, a red X is displayed.

Tryout Stop: This button is only active when a Tryout has been started. Press this button to stop the Tryout.

Tryout This Step: Pressing this button will run only the currently selected Step. Its pass/fail status will be updated in the Setup Step List.

Tryout in a Loop: Runs your inspection continuosly in a loop. The tryout does not stop until the user presses the Tryout Stop button.

Tryout and Acquisition Options: A dropdown list of options that control how image Acquisition and/or Tryouts are executed.

The options are:

- Acquire Images During Tryout: When this option is enabled, and you run a Tryout, a new image will be acquired. This is the default behavior. Occasionally you will want to run over and over on the current image without acquiring a new one, in which case you should disable this option.
- Use Triggers: If the Acquire Step(s) in your Inspection have a Trigger assigned, and this option is enabled, then the Setup Manager will block and wait for a Trigger when ever you acquire a single image, or live video or when ever your start a Tryout. If you do not want to wait for the trigger, disable this option.
- Use I/O: This is identical to the “Use Triggers” option, only it applies to cases where you are using the Digital Input Step to wait for a specified input point to transition. If enabled, when you run a Tryout, your inspection will block and wait for the I/O point to transition. If you do not want to block on your Digital Input Steps, disable this option.
- Run Step after it is Modified: When enabled, a Vision tool in your Job will automatically be run whenever you move/resize it’s ROI, or modify one of it’s parameters. When working with large jobs on slower PCs, you may find that this makes performance somewhat sluggish, so turning this option off will prevent the Step from running after each modification.

Train Step: This button is used to train the currently selected Step. It is only enabled if the selected Step is trainable (e.g. One Pin Find, DataMatrix tool, OCV Fontless Step, etc) and is disabled for those Steps that do not need to be trained (Blob tool, Flaw tool, Fast Edge, etc). When a trainable Step is selected, the background color of the button will indicate whether it is currently trained or not. A green background means the Step is trained, red indicates that it needs to be trained.

Acquire an Image: Clicking this button will cause a new image to be acquired and displayed. If the “Use Triggers” option is enabled, and you have a trigger assigned in your Acquire Step, then Setup Manager will wait for the trigger before acquiring an image. If you want to acquire images without waiting for the trigger, click on the Tryout and Acquisition options button, and turn off the “Use Triggers” option.

Live Video: Clicking this option puts the Setup Manager into Live Video mode. If the “Use Triggers” option is enabled, and you have a trigger assigned in your Acquire Step, then Setup Manager will wait for a trigger before acquiring an image. If you want Live Video mode to not wait for the trigger, click on the Tryout and Acquisition options button, and turn off the “Use Triggers” option.

Zoom In/Out Buttons: Use these buttons to zoom the current image either in or out.

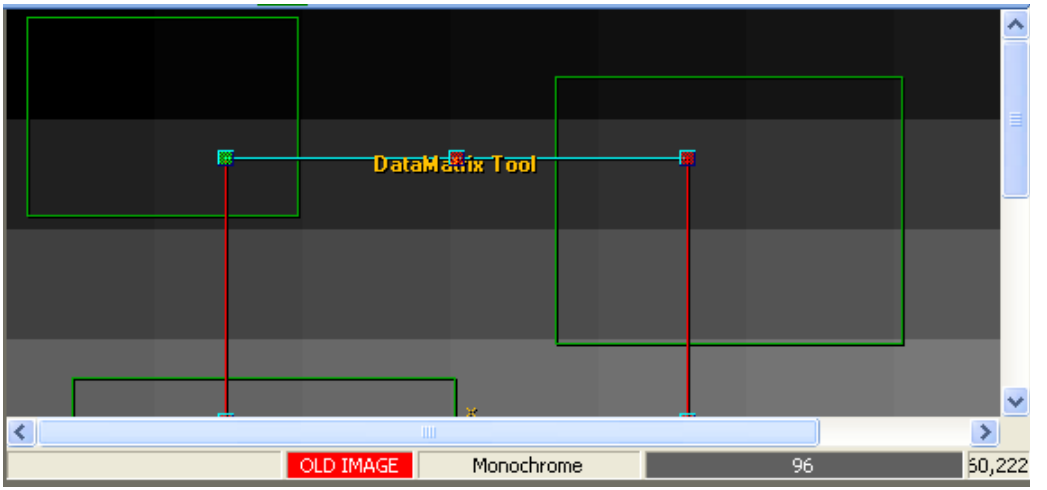
Zoom Auto: Automatically zooms the image so that it fits within the current image view.

Zoom 1:1: Causes the image to be displayed at a 1 to 1 ratio. In other words, the image is displayed without any zooming applied.

Setup Step List:

Displayed on the left side of the control, this is a flat list of all the Steps in your Job that may need to be set up. This list will not contain all of the Steps in your Job, only those that have an ROI (region of interest) and those that have a Status that you might want to watch when running tryouts (like the IF step, VarAssign step, Digital Output Step, etc). Selecting a Step in this list will also cause it to be selected in the Image View and in the Datum Grid View. When running Tryouts, a check mark will be displayed next to each Step in this list that passes, and a red X will be placed next to those Steps that fail.

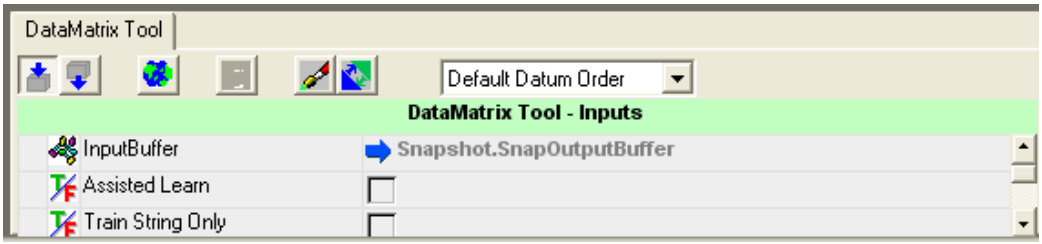
Image View:



Displayed at the top right of the control, this is where you will view your images. The ROI (region of interest) for each Vision tool in the currently selected Inspection will be displayed here as well. You can reposition any ROI by clicking and dragging, and you can resize them by grabbing the control points on the corners of the ROI, and dragging. Selecting any ROI

in the image will also cause that Step to be selected in the Setup Step List, and in the Datum View.

Datum View:



Displayed at the bottom right of the control, the Datum View displays the list of parameters for the currently selected Step. You would come here to adjust the performance of your vision tools.

Setup Manager Options:

The Options property can be used to adjust various options of the Setup Manager control. This is a bitfield property, so multiple values can be combined. The values of the bits are defined by the SetupManagerOptions enum. The possible enum values are as follows:

AllowMouseToolInsertion:

Allow tools to be inserted into the job by clicking and dragging in the image view. On by default.

AutoRegenerate:

When set, Steps will be automatically run whenever you adjust their ROI size or position, or when you change a parameter in the Datum View. On by default.

AutoRetrain:

Trainable Steps will be automatically trained whenever they are selected. This option is rarely used, and is off by default.

AllowContextMenu:

When the user right-clicks on the Image View, a context menu will pop-up providing various options. Clear this bit to disable the context menu. This is on by default.

EditGraphics:

Graphics should be displayed when the Steps are run. On by default.

RunGraphics:

This is not applicable.

AllowToolMovement:

On by default, clear this bit if you want to disable the movement of tools in the Image View.

DefaultOptions = AllowMouseToolInsertion | AllowToolMovement | AutoRegenerate | AllowContextMenu | EditGraphics | RunGraphics

To make it easier to set the individual Option values, the method OptionSet() if provided. Following are some examples of how you might set various Setup Manager options.

```
//Disable the context menu
ctlSetup.OptionSet(SetupManagerOptions.AllowContextMenu,
false);
//Disable tool movement
ctlSetup.OptionSet(SetupManagerOptions.AllowToolMovement,
false);
//Want Steps to auto run whenever they are modified
ctlSetup.OptionSet(SetupManagerOptions.AutoRegenerate,
true);
```

Showing, Hiding and Repositioning the Various Elements of Setup Manager:

As mentioned in the previous section, Setup Manager will, by default, show it's toolbar, the Setup Step List, the ImageView and the Datum View. All of these components can be hidden however, and there is also a StepTree View that can be shown, which allows you to view your entire Job, and to add or remove Steps. To show or hide individual control

elements, use the OptionsLayout property. This property is a bitfield, so multiple options can be combined. The values of the bits are defined by the SetupManagerLayoutOptions enum. The possible values are:

ShowDatumGrid:

Set this bit to show the Datum grid view, clear it to hide it. On by default.

ShowDatumGridOnRight:

Set this bit to position the Datum grid view on the right side of the control, rather than at the bottom. This bit is off by default.

ShowStepTree:

Set this bit to show the Step Tree view of the Job. This bit is ff by default.

ShowToolbar:

Clear this bit to hide the toolbar, which is shown by default. You would hide the toolbar when you want to create your own.

ShowView:

Clear this bit if you want to hide the image view. On by default.

ShowItemList:

Clear this bit if you want to hide the Setup Step List. On by default.

ShowStatusbar:

Clear this bit if you want to hide the status bar at the bottom of the control. On by default.

ShowInspErrorStatusbar:

Currently not used.

ShowBufStatusbar:

Clear this bit to hide the status bar that is show at the bottom of the Image View. On by default.

DefaultLayout = ShowView | ShowBufStatusbar | ShowItemList | ShowDatumGrid | ShowToolbar

These are the options that are set by default.

JobEditLayout = ShowBufStatusbar | ShowStepTree | ShowDatumGrid | ShowToolbar

This set of options provides you with a view that can be used to edit the Job.

JobEditLayoutWithImage = ShowView | ShowBufStatusbar | ShowStepTree | ShowDatumGrid | ShowToolbar,

This set of options allows you to edit the Job, and also see the image at the same time.

ShowEverything = ShowView | ShowBufStatusbar | ShowItemList | ShowDatumGrid | ShowToolbar | ShowStepTree

This set of options shows all views. This should be considered an advanced layout, as most users will find this layout to be somewhat cluttered and confusing.

Rather than have to perform complex bit math on this property, you can use the OptionLayoutSet function to turn options on or off. Following are some examples:

```
//use the standard layout
ctlSetup.OptionsLayout =
SetupManagerLayoutOptions.DefaultLayout;
//hide the toolbar
ctlSetup.OptionLayoutSet(SetupManagerLayoutOptions.ShowToolb
ar, false);
//hide the Setup list, and instead show the Step Tree
ctlSetup.OptionLayoutSet(SetupManagerLayoutOptions.ShowItemL
ist, false);
ctlSetup.OptionLayoutSet(SetupManagerLayoutOptions.ShowStepT
ree, true);
//show the Datum Grid on the right side of the control,
//rather than the bottom
ctlSetup.OptionLayoutSet(SetupManagerLayoutOptions.ShowDatum
GridOnRight, true);
```

Likewise, there is an OptionIsOn method that can be used to test if a certain bit is set or not.

```
if(ctlSetup.OptionIsOn(SetupManagerOptions.AutoRegenerate))
        {
         //if AutoRegenerate is on, do something....
        }
```

Adjusting the Tryout Options

SetupManager also provides options that control the behavior of Tryouts. These options are set via the OptionsTryout property. This is a bitfield, so multiple options can be combined. The values of the bits are defined by the SetupManagerTryoutOptions enum. The possible values are:

DoAcquire

When set, a new image will be acquired when a Tryout is run. Clear this bit if you want to run on the current image and not acquire a new one. This bit is on by default.

UseTriggers

When set, an Acquire operation, Live Video and Tryouts will stop on all Snapshots on which a Trigger has been assigned. It will block and wait for a trigger before acquiring an image and running the rest of the Steps. This is useful when a part can not be placed stationary in front of the camera. This bit is off by default.

UseIO

Similar to UseTriggers, but when this option is set, Tryouts will block on any Digital Input Steps that have their “Data Valid Signal I/O” values assigned to an I/O point. This option is off by default.

TackImage

When running a Tryout, Setup Manager will always update the Image View to show the parent buffer of the current Step that is running. Set this bit when you want the control to just show the current buffer. This bit is off by default.

DefaultOptions = DoAcquire

This will set the control to the default options, which is currently to have just the DoAcquire bit set.

Acquisition Methods:

In this section we'll describe the methods that allow you to start and stop image acquisitions and live video.

To command the Setup Manager to acquire a new image, and display it in the image view, you will call one of the methods that controls Acquisition. The easiest is the Acquire() method. In this example, assume our SetupManager control is named ctlSetup, and we simply want to start a synchronous acquisition:

```
//Perform a Synchronous acquire,
//ie. don't return until the acquire is complete.
bool bAcqSuccess = ctlSetup.Acquire();
```

In the previous example, your application will be blocked until the acquisition completes. If you have the "Use Triggers" bit set in the OptionsTryout property, then this method will block until you generate a trigger. If you can't generate a trigger, you'll be blocked forever. This may be confusing to your user, so if you are using the "Use Triggers" option, you will want to at least use the AcquireWithTimeout() method. As the name implies, you can specify a timeout value, so you will not block forever:

```
//Perform a Synchronous acquire,
//but timeout if the acquisition is not complete after 2
//seconds
bool bAcqSucess = ctlSetup.AcquireWithTimeout(2000);
```

The other alternative to specifying a timeout, is to perform an asynchronous acquire. This is done by calling the AcquireStart() method. This will start an acquisition, and immediately return control to your application. When the image acquisition is complete, you will receive a StateChanged event, which will return a StateChangedEventArgs parameter. The StateChangedEventArgs parameter has a StateChangedEvent property that will Identify "AcquireDone" as the state. You can also call the AcquireStop method to force the asynchronous acquire to stop. In the following example, we present a simple case where one command button is used to start an acquisition, and the second is

used to stop it. Assume these buttons are named butAcqStart and butAcqStop.

Note: While an asynchronous acquisition is active, you can not start live video or a tryout.

```
private void frmMain_Load(object sender, EventArgs e)
{
//wire up our event handler when the app starts up,
//so we can receive AcquireDone notifications
    ctlSetup.StateChanged += ctlSetup_StateChanged;
}

//The click event for our "Acquire Start" event
private void butAcqStart_Click(object sender, EventArgs e)
{
    //disable this button while the acquire is active
    butAcqStart.Enabled = false;
    //Start an Asynchronouse acquisition
    bool bAcqStarted = ctlSetup.AcquireStart();

    UpdateButtons();
}

//We'll receive this event to signify any SetupManager state
//change,
//including AcquireDone
void ctlSetup_StateChanged(object sender,
SetupManager.StateChangedEventArgs e)
{
    //Check if the state change is AcquireDone
    if(e.StateChangedEvent ==
StateChangedEvents.AcquireDone)
    {
       //Update the state of our buttons
        UpdateButtons();
    }
}

//The click event for our "Acquire Stop" button
private void butAcqStop_Click(object sender, EventArgs e)
{
    //force the asynchronous acquire to stop
    ctlSetup.AcquireStop();
    //update the button states
```

```
    UpdateButtons();
}

//Updates the Enabled/Disabled state of the buttons,
//based on whether an Acquisition is active or not
private void UpdateButtons()
{
    butAcqStart.Enabled = !ctlSetup.AcquireActive();
    butAcqStop.Enabled = ctlSetup.AcquireActive();
}
```

To command SetupManager to enter live video mode, you use the LiveVideoStart method. This is an asynchronous method that will start the live video and immediately return control to you.

```
//Start asynchronous live video mode
ctlSetup.LiveVideoStart();
```

You must call LiveVideoStop to exit live video mode. While live video is active, you can not run a tryout, or start an acquisition.

```
//Stop live video
ctlSetup.LiveVideoStop();
```

To check if live video is currently active or not, you can check the “Can” property of SetupManager.

```
//if we "can" start live video, it's not currently on
if (ctlSetup.Can.LiveVideoStart)
{
}
//if we "can" stop live video, that means it is currently on
if(ctlSetup.Can.LiveVideoStop)
{
}
```

Tryout Functionality:

SetupManager provides several methods that allow you to “Try out” your inspection(s). When we say tryout, we mean to run your inspection for the purposes of debugging and or testing. Running your inspection in tryout mode is not as fast as running in a full fledged run-mode, so you should never use the tryout functionality to run your inspections on the factory floor. When you start a tryout, each Step in the Setup List is run sequentially, and a green check mark is placed next to those Steps that

pass, and a red X is placed next to those that fail. So you receive constant feedback on how your inspection is performing. You can launch tryouts programactically using one of the several methods provided by SetupManager. In this section we provide a brief description and example of the various tryout options available to you.

Run the current Step:

You can run the current Step by simply calling the TryoutCurrentItem() method:

```
ctlSetup.TryoutCurrentItem();
```

This will run the currently selected Step, update it's run graphics in the image, and update the pass/fail status of the Step in the Setup List by displaying a green check for a pass, and a red X for a fail. This is an asynchronous method, meaning it will start the run of the selected Step, and immediately return. The StateChanged event will be sent with the TryoutDone notification when the run is complete.

Run the Inspection for One Cycle:

When you are building and testing a vision inspection, it is typical that you will want to run all of the Steps in the inspection for one cycle to see how it is performing. There are two ways that you can do this.

1. Call the TryoutStart() method, and specify you want to run for 1 iteration.
2. Call the TryoutOneCycle() method.

Each of these methods starts an asynchronous tryout. At the end of the cycle you will receive the TryoutIterationDone event, as well as the StateChanged event , which will specify the TryouDone notification. It is important to understand that you can not start other operations while a Tryout is active. You can not start an acquisition or live video for instance. When you start an asynchronous tryout, you should disable all options in your UI until it is complete, or until you have stopped the tryout by calling the TryoutStop method. Following is a simple example that demonstrates starting a Tryout for one cycle, disabling the controls in our UI, and then re-enabling them when we receive the notification that the Tryout is done.

```
// User clicked the "Tryout Start" button
private void butTryoutStart_Click(object sender, EventArgs e)
{
```

```
    //start an asynchronous tryout that will run for one
    //cycle
    ctlSetup.TryoutOneCycle(); //could also call
//ctlSetup.TryoutStart(1);
    //Disable all controls in your UI
    DisableUI();
}
//User clicked the "Tryout Stop" button
private void butAcqStop_Click(object sender, EventArgs e)
{
    //user wants to stop the tryout,
    ctlSetup.TryoutStop();
}
//The StateChanged event will let us know when the tryout is
//complete
void ctlSetup_StateChanged(object sender,
SetupManager.StateChangedEventArgs e)
{
switch( e.StateChangedEvent)
{
    case StateChangedEvents.TryoutDone:
        //Tryout is complete, reneable the UI
        ReEnableUI();
             break;
       }
}
```

Run continuously in a Loop:

To run your inspection continuously in a loop, you simply need to call the TryoutStart() method, specifying nothing for the number of iterations.

```
ctlSetup.TryoutStart();
```

This will run your inspection until you call the TryoutStop() method as shown in the example above.

Checking the Current State of the Control Using the "Can" Property

SetupManager has a property named "Can" that is intended to tell you what operations you can and can not currently perform safely. The Can property returns a variable of type SetupManagerCan. The properties of

this object identify which operations can and can not be performed. For example, you might check if it is OK to start an Acquisition like this:

```
//Start an Asynchronouse acquisition if we can
if(ctlSetup.Can.AcquireStart)
      ctlSetup.AcquireStart();
```

Likewise, you could check if it was OK to start a Tryout like this:

```
if (ctlSetup.Can.TryoutStart)
      ctlSetup.TryoutStart();
```

As demonstrated earlier, you can also use this property to tell you if Live Video is currently active. The LiveVideoStop property will only be true when live video has been turned on, so if you needed to make sure that Live Video was off before performing some action, you could do the following:

```
//if live video is On, turn it off
if (ctlSetup.Can.LiveVideoStop)
      ctlSetup.LiveVideoStop();
```

Detecting State Changes

Often you will want to be informed when the user starts a Tryout, or turns on live video, or changes the state of the control in some way. Rather than have a separate event for each of these state changes, SetupManager provides just one, the StateChanged event. This event passes a StateChangedEventArgs variable to your event handler. StateChangedEventArgs contains a StateChangedEvent property that identifies the event that has changed state. Following is an example where we respond to all of the possible state change events, and simply dump text to the console window identifying the state:

```
private void frmMain_Load(object sender, EventArgs e)
{
//wire up our StateChanged event handler
    ctlSetup.StateChanged += ctlSetup_StateChanged;
}
void ctlSetup_StateChanged(object sender,
 SetupManager.StateChangedEventArgs e)
{
switch( e.StateChangedEvent)
{
    case StateChangedEvents.AcquireDone:
```

```
            Console.WriteLine("Aquisition is complete");
            break;
        case StateChangedEvents.AcquireStarted:
            Console.WriteLine("Aquisition has been started");
            break;
        case StateChangedEvents.LiveDone:
            Console.WriteLine("Live Mode has exited");
            break;
        case StateChangedEvents.LiveStarted:
            Console.WriteLine("Live Mode started");
            break;
        case StateChangedEvents.TryoutDone:
            Console.WriteLine("Tryout is complete");
            break;
        case StateChangedEvents.TryoutStarted:
            Console.WriteLine("Tryout has been started");
            break;
        case StateChangedEvents.WizardNext:
           Console.WriteLine("User pressed Wizard Next button");
            break;
        case StateChangedEvents.WizardPrev:
            Console.WriteLine("User pressed Wizard Previous
button");
            break;
}
}
```

Properties, Methods and Events

In the following section we provide a complete description of all Setup Manager properties, methods and events.

Properties:

SetupManagerCan Can { get; }

Returns a SetupManagerCan object, the properties of which identify which actions currently “can” be executed. All properties of SetupManagerCan are of type bool, and they identify a method of SetupManager that is OK to call. Following are a few examples of SetupManageCan properties.

SetupManagerCan.Acquire: It is OK to call the Acquire method.

SetupManagerCan.AcquireStart: It is OK to call the AcquireStart method.

SetupManagerCan.AcquireStop: It is OK to call the AcquireStop method (an acquire is active).

SetupManagerCan.AcquireWithTimeout: It is OK to call the AcquireWithTimeout method.

SetupManagerCan.LiveVideoStart: It is OK to call LiveVideoStart.

SetupManagerCan.LiveVideoStart: It is OK to call LiveVideoStop (Live video is currently active).

SetupManagerCan.TryoutStop: It is OK to call TryoutStop (a tryout is currently active).

int ImageViewSize { set; get; }

Gets/Sets the size of the splitter panel in which the ImageView is contained.

bool IsCurrentItemTrainable { get; }

Returns true if the Step that is currently selected is a trainable Step.

bool IsCurrentItemTrained { get; }

Returns true if the currently selected Step is trained.

bool IsCurrentItemUntrainable { get; }

Returns true if the currently selected Step can be untrained. This typically only applies to Data Matrix tools.

List<SetupItem> ItemList

Returns a List of all of the items in the Setup List. Each item is represented by a SetupItem object, which has properties identifying the Step for this item, as well as the next Step and the previous Step in the list.

int ListViewSize { set; get; }

Gets/Sets the size of the splitter pane in which the List View is contained.

SetupManagerOptions Options { set; get; }

Gets/Sets the generic SetupManager options. This property is of type SetupManagerOptions, the members of which identify the available options. This is a bitfield, so multiple options can be combined. It is generally easier to use the OptionIsOn() method to check if a particular option has been enabled, and it is eaiser to use the OptionSet() method to turn options on and off. Refer to the “Setup Manager Options” section earlier in this chapter for a list of the available options.

SetupManagerLayoutOptions OptionsLayout { set; get; }

Gets/Sets a bitfield that controls which elements of SetupManager are shown/hidden, as well as controlling how some elements are positioned. Multiple options can be combined. The available options are represented by the SetupManagerLayoutOptions enum, which has the following fields:

ShowDatumGrid: When set, the datum view will be shown, if cleared, it will be hidden.

ShowDatumGridOnRight: When set the datum view is positioned on the right side of the control, if cleared, it is positioned at the bottom of the control.

ShowStepTree: When set, the Step Tree View is displayed.

ShowToolbar: Clear this bit when you want to hide the toolbar and create your own.

ShowView: Shows/hides the image view.

ShowItemList: Shows/hides the Setup Item List.

ShowStatusbar: Shows/hides the status bar at the bottom of the control.

ShowBufStatusbar: Shows/hides the status bar at the bottom of the Image View.

It is generally easier to use the OptionLayoutIsOn() and OptionLayoutSet() methods to get and set various options. Refer to the

earlier section “Showing, Hiding and Repositioning the Various Elements of Setup Manager” for examples of how to manipulate this property.

SetupManagerTryoutOptions OptionsTryout { set; get; }

Gets/Sets a bitfield that controls how Tryouts and Image acquisitions are performed. Multiple options can be combined. The available options are represented by the values in the SetupManagerTryoutOptions enum, which has the following fields:

DoAcquire: When set, a new image will be acquired everytime you run a Tryout. Clear this bit if you want to run a tryout repeatedly on the current image. This bit is on by default.

UseTriggers: When set, tryouts, acquire and live video will all wait for a trigger (if one is set in the Snapshot) before acquiring an image. Clear this bit if you don’t want to wait for triggers before acquring an image. This bit is off by default.

UseIO: Similar to UseTriggers, but this only effects Tryouts, and will cause you to block on DigitalInput Steps that have an I/O point assigned to their “Data Valid Signal I/O” datums. Off by default.

TackImage: When running a Tryout, SetupManager will automatically update the ImageView to show you the buffer for the currently selected Step. When this bit is set, the current image is “tacked”, and it will always show the current image buffer, regardless of what Step is selected. Off by default.

DefaultOptions = DoAcquire: This represents the default settings, which is currently to have just the DoAcquire option turned on.

Step RootStep { set; get; }

Use this property to connect the SetupManager control to your Job. You will typically set the RootStep property to one of the Inspection Steps in your Job, though you may also connect to the VisionSystem Step or a Snapshot Step. Once you have assigned a Step to this property, SetupManager will update itself, and you will be ready to acquire images, run tryouts, adjust job parameters, etc.

Step SelectedStep { set; get; }

Gets/Sets the Step that is currently selected in the SetupManager.

Methods:

bool Acquire()

Runs a synchronous acquire operation, which means that this function will not return until the Acquisition is complete. Returns true if the acquisition was successful. WARNING, if you have activated the “UseTriggers” option in the OptionsTryout property, and the current Snapshot has a trigger assigned, this function will block forever until the trigger is received. If you are unable to send a trigger, this will cause your application to lock up. If you are planning to use the “UseTriggers” option, you should never call this function, and should instead use either the AcquireWithTimeout() function, or start an asynchronous acquisition using AcquireStart.

bool AcquireActive()

Returns true if there is currently an acquisition operation in process. You would typically use this method after starting an asynchronous acquisition with AcquireStart.

bool AcquireStart()

Starts an asynchronous acquisition, and returns immediately. The acquisition will run on a separate thread, so your application will not be blocked waiting for it to complete. You can poll to see if the acquisition has completed by calling the AcquireActive() method, or you can respond to the StateChanged() event, which will set the AcquireDone flag in the StateChangedEvent property of the StateChangedEventArgs argument that is passed to your event handler, as shown below:

```
void ctlSetup_StateChanged(object sender,
SetupManager.StateChangedEventArgs e)
{
switch( e.StateChangedEvent)
      {
       case StateChangedEvents.AcquireDone:
            Console.WriteLine("Aquisition is complete");
                  break;
       ……
```

Returns true if the acquisition was stated successfully, false if unable to start. All other operations are not allowed while an acquisition is in process, i.e. you can’t start Live Video or a run a Tryout.

void AcquireStop()

Stops an asynchronous acquisition that was started with the AcquireStart method.

bool AcquireWithTimeout(**int** *timeout*)

Performs a synchronous acquisition, but lets you specify a timeout value, so you won't block forever waiting for it to complete. The timeout is specified in milliseconds. Returns true if the acquisition completed successfully, false if you timed out.

bool LiveVideoStart()

Puts SetupManager into Live Video mode, and then returns immediately. Returns true if able to start Live Video, false if not. Images will be acquired and displayed as fast as possible, all other operations will be disabled while Live Video is active, i.e., you can't run tryouts or call one of the Acquire methods while Live Video is active. Call LiveVideoStop to exit Live Video mode.

bool LiveVideoStop()

Exits Live Video mode, returns true if successful.

bool OptionIsOn(**SetupManagerOptions** *op*)

Checks the Options bitfield property to see if the specified option is on, and returns true if so, false if not.

bool OptionLayoutIsOn(**SetupManagerLayoutOptions** *lo*)

Checks the OptionsLayout bitfield property to see if the specified option is on, and returns true if so, false if not.

void OptionLayoutSet(**SetupManagerLayoutOptions** *lo*, **bool** *bSet*)

lo: The layout option whose state you want to change.

bSet: If true, the option is turned on, if false, it is cleared.

Sets the specified bit in the OptionsLayout property, to the specified state. This method provides an easy way to set and clear various layout options, rather than having to perform the bit math your self.

void OptionSet(**SetupManagerOptions** *op*, **bool** *bSet*)

op: The option whose state you want to change.

bSet: If true, the option is turned on, if false, it is cleared.

Sets the specified bit in the Options bitfield property to the specified state.

bool OptionTryoutIsOn(**SetupManagerTryoutOptions** *op*)

Returns true if the specified tryout option is currently turned on, false if it is not.

SetupManagerTryoutOptions
OptionTryoutSet(**SetupManagerTryoutOptions** *op*,**bool** *bSet*)

op: The option whose state you want to change.

bSet: If true, the option is turned on, if false, it is cleared.

Sets the specified tryout option to the specified state. This provides an easy way to set and clear tryout options without having to perform complex bitmath on the OptionsTryout property directly.

bool SelectNextItemInList()

Causes SetupManager to jump from the currently selected Step to the next item in the Setup List.

bool SelectPrevItemInList()

Causes SetupManager to jump from the currently selected Step to the previous item in the Setup List.

void SelectStep(**Visionscape.Steps.Step** *step*)

Selects the specified Step.

void TrainCurrentItem()

Trains the currently selected Step. Has no effect if the currently selected Step is not trainable.

bool TryoutCurrentItem()

Performs a tryout on the currently selected Step. This will cause the current Step to run, as well as any other Steps that it is dependent upon.

bool TryoutOneCycle()

Starts an asynchronous tryout operation that will run all of the Steps in the Setup List for one cycle and then stop. The tryout is started, and then control is immediately returned to you. Returns true if the tryout was successfully started. You will receive the TryoutIterationDone event as well as a StateChanged event when the tryout is complete. The event handler for the StateChanged event will be passed a StateChangedEventArgs parameter, and the StateChangedEvent property will be set to TryoutDone.

The pass/fail status of each Step will be displayed in the Setup List (green check for pass, red X for fail). All other operations are not allowed while a Tryout is in process, i.e., you can't call one of the Acquire methods, or start Live Video.

bool TryoutStart()

Starts an asynchronous tryout operation that will run all of the Steps in the Setup List repeatedly until TryoutStop is called. The tryout is started, and then control is immediately returned to you. Returns true if the tryout was successfully started. You will receive the TryoutIterationDone event after each completed cycle. When the tryout is stopped by a call to TryoutStop, you will receive the StateChanged event when the tryout is fully stopped. The event handler for the StateChanged event will be passed a StateChangedEventArgs parameter, and the StateChangedEvent property will be set to TryoutDone.

The pass/fail status of each Step will be displayed in the Setup List (green check for pass, red X for fail). All other operations are not allowed while a Tryout is in process, i.e., you can't call one of the Acquire methods, or start Live Video.

bool TryoutStart(**int** *nIterations*)

This overloaded version of TryoutStart allows you to specify how many cycles you want the tryout to run for before exiting.

bool TryoutStop()

Stops an asynchronous tryout, returns true if the tryout was successfully stopped.

void UntrainCurrentItem()

Untrains the currently selected Step. This currently only applies to Data Matrix tools.

bool WizardNext()

Runs the current Step, then selects the next Step in the list.

bool WizardPrev()

Changes the selection from the currently selected Step the previous Step in the list.

void ZoomAuto()

Puts the ImageView into Auto Zoom mode. This will automatically zoom the image so that it fits entirely within the visible area.

void ZoomIn()

Zooms in on the current image.

void ZoomOne()

Displays the image at a 1 to 1 zoom ratio, meaning that no zoom is applied.

void ZoomOut()

Zooms out on the current image.

Events:

DatumChanged

Event Handler Function Signature:

```
void ctlSetup_DatumChanged(object sender,
 SetupManager.DatumEventArgs e)
```

Fired whenever the user changes a datum value in the Datum View. The modified datum is specified in the DatumEventArgs parameter.

DatumSelected

Event Handler Function Signature:

```
void ctlSetup_DatumSelected(object sender,
SetupManager.DatumEventArgs e)
```

Fired whenever the user selects a datum in the Datum View. The selected Datum is specified in the DatumEventArgs parameter.

JobChanged

Event Handler Function Signature:

```
void ctlSetup_JobChanged(object sender,
SetupManager.JobChangedEventArgs e)
```

Fired whenever the job is modified. How the job was modified is identified by the JobChangedEventArgs parameter, which has a JobChangedEvent property of the type JobChangedEvents. This enum has the following possible values:

RootStepChanged

ToolInserted

ToolMoved

ToolToBeDeleted

ToolTrained

LayoutChanged

Event Handler Function Signature:

```
void ctlSetup_LayoutChanged(object sender,
SetupManager.LayoutChangedEventArgs e)
```

Fired whenever the OptionsLayout property is modified.

OptionsChanged

Event Handler Function Signature:

```
void ctlSetup_OptionsChanged(object sender,
SetupManager.OptionsChangedEventArgs e)
```

Fired whenever the Options property is modified.

StateChanged

Event Handler Function Signature:

```
void ctlSetup_StateChanged(object sender,
SetupManager.StateChangedEventArgs e)
```

Fired whenever a major operation has started or completed. Your event handler will receive a StateChangedEventArgs parameter, which has a StateChangedEvent property that identifies the state change. This property is of type StateChangedEvents, which is an enum that identifies the possible state changes. Refer to the earlier section “Detecting State Changes” for an example of how to respond to this event, and a list of the possible state changes.

StepSelected

Event Handler Function Signature:

```
void ctlSetup_StepSelected(object sender,
SetupManager.SelectionChangedEventArgs e)
```

Fired whenever the selected Step is changed. The newly selected step is specified in the SelectionChangedEventArgs parameter.

TryoutIterationDone

Event Handler Function Signature:

```
void ctlSetup_TryoutIterationDone(object sender,
SetupManager.TryoutIterationDoneEventArgs e)
```

Fired whenever a tryout iteration completes. When running a single tryout, this will fire once at the end of the cycle. When running tryouts in a loop, this event will fire at the end of each cycle. The TryoutIterationDoneEventArgs parameter will be passed to your event handler. This object contains a Report property that holds an InspectionReport object. The InspectionReport holds all of the results that are selected for upload in the Inspection, as well as the standard counts and stats. NOTE: The other events related to Tryouts, such as TryoutStart and TryoutStop, are sent via the StateChanged event. This event is not

handled by a StateChanged event, because then we wouldn't be able to include the InspectionReport.

TryoutOptionsChanged

Event Handler Function Signature:

```
void ctlSetup_TryoutOptionsChanged(object sender,
SetupManager.TryoutOptionsChangedEventArgs e)
```

Fired whenever the OptionsTryout property is modified.

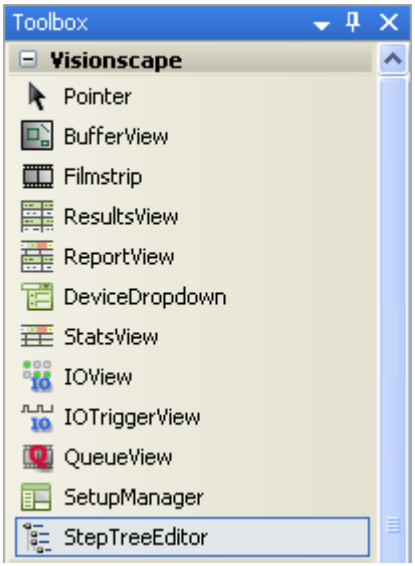
ZoomEvent

Event Handler Function Signature:

```
void ctlSetup_ZoomEvent(object sender,
SetupManager.ZoomEventArgs e)
```

Fired whenever the user zooms in or out, or changes the zoom mode to Auto or 1 to 1 mode.

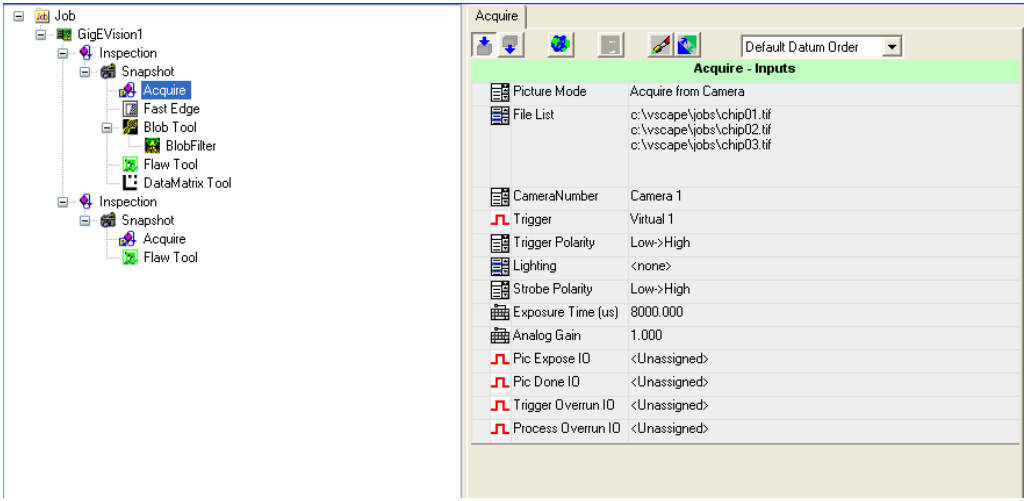
The StepTreeEditor Control



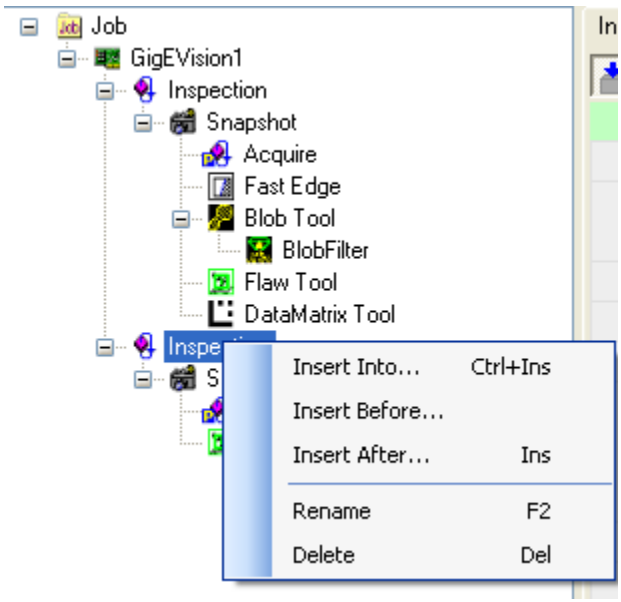
The StepTreeEditor control allows you to view your Job in its hierarchical tree structure, and to add or remove Steps. So you would use this control when you want to allow your users to edit their Jobs. To use this control you simply connect its RootStep property to one of the Steps in your Job. The StrepTreeEditor will then display the RootStep and all of the children beneath it. You would typically connect the RootStep to either your JobStep or to one of the VisionSystemSteps in your Job. Assuming we had a JobStep variable named _job, we would connect the StepTreeEditor to it like this:

```
//Connect our StepTreeEditor to our JobStep
    ctlStepTree.RootStep = (Step)_job;
```

Once connected, the StepTreeEditor would look something like this:



When the user selects a Step in the tree, all of the Datums for that Step are displayed in the DatumView on the right side of the control. You can hide the DatumView if you wish using the ShowDatums property.



The user can add or remove Steps by right-clicking on the tree. This will bring up a context menu that provides options to insert into, before or after the current Step. Selecting one of these options will cause the “Insert Step” dialog to be displayed. This dialog allows the user to select which type of Step they want to add to their Job. Once the Step type is selected, and OK is pressed, the new Step is added relative to the currently selected Step. If Insert Into was chosen, the new Step is inserted into the selected Step (always at the end of the child list). If Insert Before or Insert After are chosen, the new Step is inserted immediately before or after the selected Step. The context menu also allows you to rename the current Step, or delete the current Step.

The StepTreeEditor control exposes the InsertStep method which allows you to programmatically launch the Insert Step dialog. This would allow you to create a toolbar for the StepTreeEditor if you wished, with buttons to represent the Insert Into, Before and After options.

Following is a complete list of the Properties, Methods and Events of the StepTreeEditor control.

Properties:

Visionscape.Steps.Step RootStep { set; get; }

Connect the StepTreeEditor to your Job using this property. When this property is set, the control will update to display this Step as the root of the tree, with all of its child Steps displayed hierarchically beneath it. You would typically set the RootStep to either your JobStep or to a VisionSystemStep.

Step SelectedStep { set; get; }

Gets/Sets the Step that is currently selected in the tree.

bool ShowDatums { set; get; }

Set to true if you want the Datum View to be displayed, set to false if you want it hidden.

bool ShowDatumsInTree { set; get; }

Set to true if you want all of the Datums for each Step to also be displayed in the tree. Set to false if you only want to see Steps in the tree, which is the default behavior.

int SplitterDistance { set; get; }

Gets/sets the distance in pixels from the left edge of the control to the splitter that separates the tree from the Datum View.

Methods:

void EditStep(**Visionscape.Steps.Step** *step*)

Connects the control to the specified Step. This is identical to setting the RootStep property.

void InsertStep(**InsertStepOption** *option*)

Starts an Insert Step opertation. This causes the “Insert Step” dialog to be launched, which allows your user to select the type of Step they want to insert. The selected Step will then be created and added to the Job relative to the currently selected Step. Where the newly added Step is inserted depends on the option parameter. The option parameter is of type InsertStepOption, which provides 3 options.

InsertStepOption.Into: Inserts the new Step into the currently selecte Step, and the end of its child list.

InsertStepOption.After: Inserts the new Step immediately after the selected Step.

InsertStepOption.Before: Inserts the new Sep immediately before the selected Step.

void RefreshTree()

Forces the control to rebuild and redisplay its tree. You might call this if you are modifying the Job programmatically outside of the StepTreeEditor, and want to force your changes to be seen in the tree.

void SelectStep(**Visionscape.Steps.Step** *step*)

Sets the selected Step in the tree to the specified Step.

Events:

DatumChanged

Event Handler Function Signature:

```
void ctlStepTree_DatumChanged(Step step, Datum dat)
```

Fired whenever the user changes a Datum value in the Datum view. The Datum that was modified, and its owning Step, are passed to your event handler.

DatumSelected

Event Handler Function Signature:

```
void ctlStepTree_DatumSelected(Step step, Datum dat)
```

Fired whenever the user selects a Datum in the Datum view. The Datum that was selected, and its owning Step, are passed to your event handler.

StepInserted

Event Handler Function Signature:

```
void ctlStepTree_StepInserted(Step step)
```

Fired whenever a new Step is inserted into the Job. A reference to the newly inserted Step is passed to your event handler.

StepRenamed

Event Handler Function Signature:

```
void ctlStepTree_StepRenamed(Step step)
```

Fired whenever the name of a Step is changed.

StepSelected

Event Handler Function Signature:

```
void ctlStepTree_StepSelected(Step step)
```

Fired whenever a new Step is selected in the tree. A reference to the newly selected Step is passed to your event handler.

StepToBeDeleted

Event Handler Function Signature:

```
void ctlStepTree_StepToBeDeleted(Step step)
```

Fired whenever a Step is about to be deleted. A reference to the Step is passed to your event handler.

APPENDIX A

Loading and Running Jobs

Loading and Running Jobs, Receiving Results and Images

In this C# example we will walk you step by step through the process of building a very simple user interface. This example will demonstrate the following:

- Loading a Job (AVP file) from Disk.
- Connecting that Job to a Visionscape Device.
- Running the Job.
- Creating a report connection to handle images and results.
- Displaying the images and results at runtime.

Getting Started

– Launch Visual Studio.

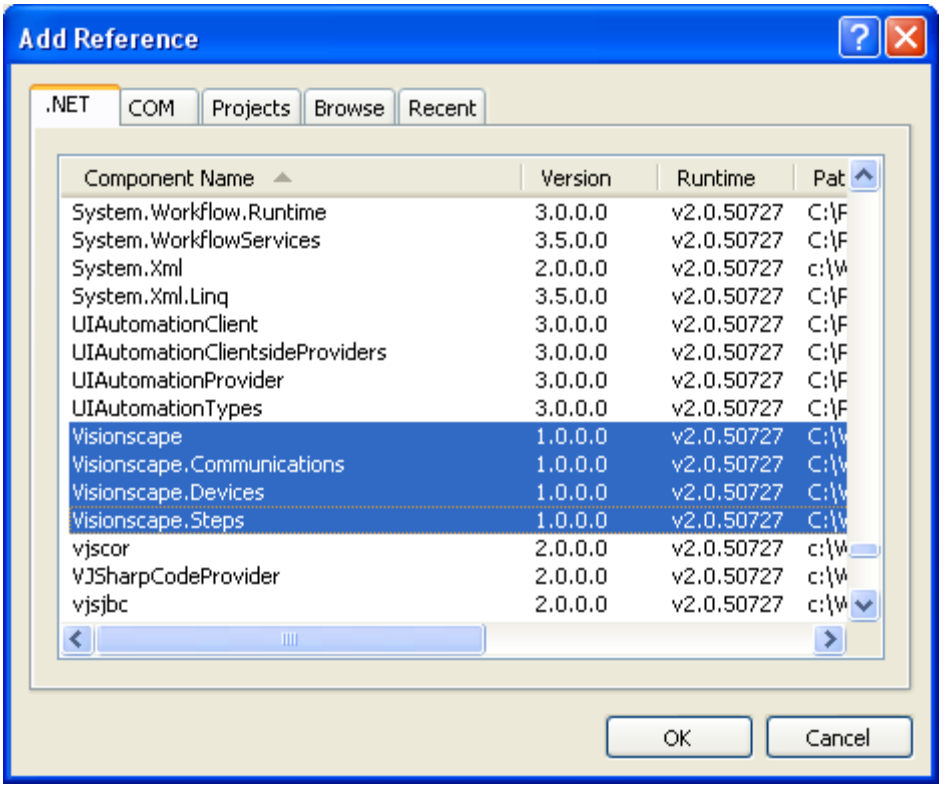
– From the File Menu, choose “New Project…”

– In the New Project dialog, choose “Visual C#” as your language, and then choose “Windows Forms Application” as your project type.

– Enter “VisionscapeSample” for the project name, and press OK.

Add References to your project

– Go to the Project menu and select “Add Reference…”. In the “Add a Reference” dialog, under the “.NET” tab, add references to the following Visionscape assemblies:



– Go to the View menu and select “Toolbox”. This will show the toolbox panel on the left-hand side of the window. If Visual Studio was installed on your PC when you installed Visionscape, then you should have a tab named “Visionscape”, and it will contain all of the visual components provided by Visionscape. If you don’t have this tab, you can run the InstallCtrlsToToolbox.bat utility to install it.

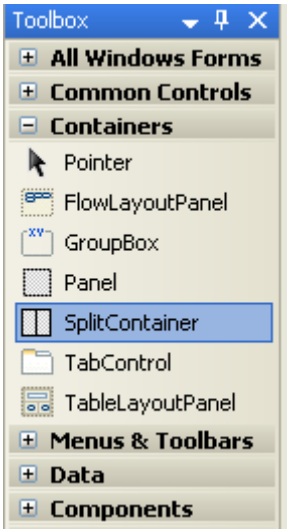
Rename the Main Form

– Change the name of “Form1.cs” in your Project to “frmMain.cs”

Add Components to the Main Form

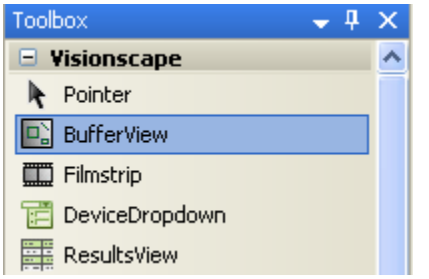
In this example we will display an image using our BufferView control, and we will display the uploaded inspection results using our ReportView control. We will insert the two controls into a Splitter Control, so the user can adjust the size of each.

Add Split Container, BufferView and ReportView



– Add a “Split Container” control to the main form. This control is found in the “Containers” tab of the ToolBox, as shown above.

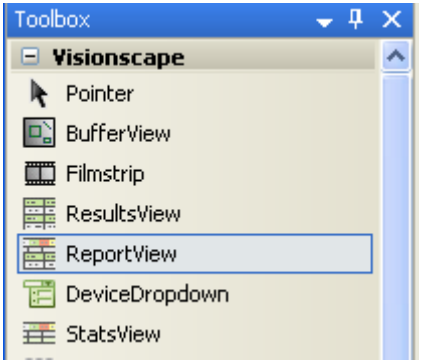
– Drag a BufferView control from the Toolbox to Panel1 of the SplitContainer. This should be found in the “Visionscape” tab of the Toolbox, as shown below:



• Change the following properties:

Property Name	Value
(Name)	ctlBufView
Dock	Fill

– Drag a ReportView control from the Toolbox to Panel2 of the SplitContainer. We will use this to display reports from the running inspections. This component is also found in the Visionscape tab of the Toolbox, as shown below:



• Change the following properties:

Property Name	Value
(Name)	ctlReportView
Dock	Fill

Add the Code

Go to the frmMain code window. At the top of the file, we will add the following “using” statements for the various Visionscape namespaces we will be accessing:

```
using Visionscape;
using Visionscape.Steps;
using Visionscape.Devices;
using Visionscape.Communications;
```

At the top of the frmMain class, add the following member variables:

```
public partial class frmMain : Form
{ ///////////////////////
//CONSTANTS
const int RESULTS_ROW = 6;

///////////////////////
//MEMBER VARIABLES
 //VsCoordinator tells us what hardware is available

VsCoordinator m_Coord;
VsDevice m_Dev;  //will hold a ref to the chosen Device
JobStep m_Job; //A JobStep loads and saves AVPs
VisionSystemStep m_VS; //the VisionSystem step
//We will use this object to receive images and results
//from the inspection
ReportConnection m_repcon;
//reference to the most recent inspection report
InspectionReport m_report;
//We will use this to save/restore application settings
Properties.Settings m_appsettings;
```

Application Startup

When our application starts up, we will load one of the Sample Jobs that is installed with Visionscape. We will connect it to the Software System in your PC (always created) and then prepare it to run. In frmMain’s Load event, we will need to do the following:

1. Instantiate our VsCoordinator and JobStep objects.
2. Load the Job from disk.
3. Get a reference to the Software System.

4. Connect the Job to the Software system by downloading to it.
5. Run the inspection.
6. Connect our report connection, so we can receive images and results at runtime.

Add an event handler for frmMain's Load event, and add the following code:

```
private void frmMain_Load(object sender, EventArgs e)
{
    //instantiate our coordinator and Job Step
    m_Coord = new VsCoordinator();
    m_Job = new JobStep();
    try
    {
        //load the sample AVP
        m_Job.Load("C:\\Vscape\\Tutorials and samples\\
                                Sample Jobs\\Wrench
                                Gauge\\example_wrenchgauge.avp
                                ");
        //Now get a reference to the Software system
        m_Dev = _coord.FindDeviceByName("SoftSys1");

        //connect the job to hardware
        m_Dev.Download(_job.VisionSystemStep(), 1);

        //Start the inspection running
        m_Dev.StartAll();

        //Connect our report connection
        ConnectReport();

    }
    catch (Exception ex)
    {
        MessageBox.Show("Exception thrown while starting up:
" + ex.Message);
        this.Close();
    }
}
```

Making Report Connections

Add the ConnectReport function that we are calling from frmMain_Load. This function will instantiate our ReportConnection, connect it to our device, and configure it to upload images.

```
private void ConnectReport()
{
    //Create a report connection
    m_repcon = new ReportConnection();
    //Connect it to the first inspection on our device
    m_repcon.Connect(_device, 1);

    //We want images to be included in our report,
    //so add them now.
    //Find the first Inspection step
    InspectionStep insp =
    m_Job.VisionSystemStep().FindByType("Step.Inspection")
                                        as InspectionStep;
    //Add all snapshot buffers in this inspection to the report
    m_repcon.AddSnapBuffers(insp);

    //lastly, we need to wire up our event handler
    m_repcon.NewReport += m_repcon_NewReport;
}
```

Handling Reports

Now that we can receive reports, in this section we will add the code to handle them. Our NewReport event handler will receive an inspection cycle report, and we will then display the first image in our BufferView control, and display the inspection data in our ReportView.

```
//This event will be received after every inspection cycle
void _m_repcon_NewReport(object sender,
                                ReportConnectionEventArgs e)
{
    InspectionReport report = e.Report;

    //display the first image in our BufferView control
    if(report.Images.Count > 0)
    {
        ctlBufView.Buffer = report.Images[0] as BufferDm;
    }

    //display the report data in our ReportView control
```

```
        ctlReportView.Report = report;
}
```

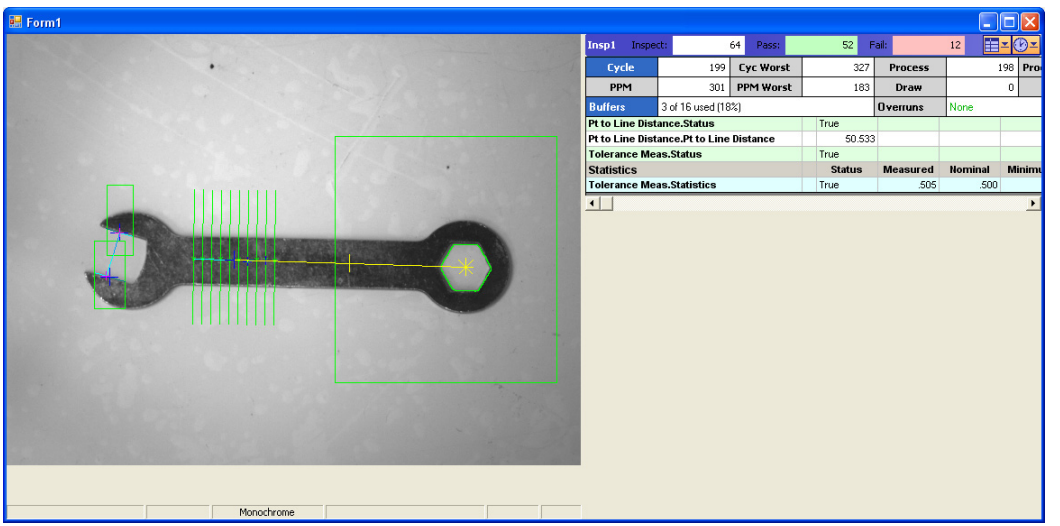
Handling Application Shut Down

When our application shuts down, we need to make sure that we disconnect our ReportConnection, and stop the running inspection. To do this, add the following code to the FormClosing event handler:

```
private void frmMain_FormClosing(object sender,
                                 FormClosingEventArgs e)
{
        //Disconnect the report connection
        if m_repcon.IsConnected
        {
            m_repcon.Disconnect();
        }

        //Stop the running inspections
        _device.StopAll();
}
```

Compile and Run

You should now be able to build the project, and run it. When your application runs, you should see something like this:



Additional Samples

This application is obviously not production level, but it hopefully illustrates the fundamentals involved with creating a Visionscape runtime user interface. If you would like to look at a more complete sample, look at the two samples that are installed with VsKit.NET

LoadAndRun:

This sample is a basic runtime user interface that provides the following features:

- Provides a toolbar that lists all available devices, and allows you to choose the device you want to work with.
- Provides a toolbar button that allows you to load any AVP file to your chosen device. This illustrates the cleanup that is required when switching from one job to another.
- Rather than displaying the results in a ReportView control, we use a standard .NET DataGridView control and illustrate how you can walk all of the results of a report and populate a grid yourself.

LoadRunAndSetup:

This sample builds on the LoadAndRun sample by adding both a Run Mode and Setup Mode. It illustrates the following concepts:

- How to implement a Setup Mode in your program using the SetupManager control.
- Illustrates a simple class that can be used to handle report connections for multiple inspections.
- Illustrates a simple control that can display multiple images at runtime. Up to four.
- Illustrates how to take control of a smart camera.