AutoCAD® Civil 3D® 2013

API Developer's Guide

April 2012

Contents

API Developer's Guide

1

About the Developer's Guide

Intended Audience

The is designed for developers who want to customize AutoCAD® Civil 3D® or create applications using the underlying APIs. It can also be used for creating macros to automate repetitive tasks for AutoCAD Civil 3D users and for developers of custom subassemblies.

AutoCAD Civil 3D APIs

There are three APIs available for customizing AutoCAD Civil 3D:

- .NET API — allows you to write extensions to AutoCAD Civil 3D in any .NET language. In general, the AutoCAD Civil 3D.NET API performs significantly faster than the COM API. Development requires Microsoft Visual Studio 2008 SP1 or better.
- COM API — you can create clients that access the COM API from managed (.NET) or unmanaged (C++) code. See Creating Client Applications (page 404). In addition, this API can be used in the Visual Basic for Applications (VBA) IDE, which is available as a separate download. VBA support is deprecated.
- Custom Draw API (in C++) — an extension of the AutoCAD ObjectARX API that allows you to customize the way AutoCAD Civil 3D renders objects. Development requires Microsoft Visual Studio.

The COM and .NET APIs are described in this guide. For more information about the Custom Draw API, see the Custom Draw API Reference (civildraw-reference.chm).

In addition, an API is provided for creating custom subassemblies in .NET. See Creating Custom Subassemblies Using .NET (page 143).

Which API you choose to use depends on what you want to do:

If you want to:	Use:
Customize the way objects are rendered in AutoCAD Civil 3D	the Custom Draw API. The Custom Draw API is an extension of the AutoCAD ObjectARX API. For example, if you wanted to number the triangles on a TIN surface, you could create a DLL using the Custom Draw API. See the sample applications shipped with AutoCAD Civil 3D for an example.
Create macros to automate repetitive actions	.NET or COM API.
Create applications to manipulate AutoCAD Civil 3D objects	.NET or COM API.

NOTE

Where possible, you should use the Civil .NET API instead of the COM API, especially for longer operations, as the .NET API is a thin layer to Civil objects and has better performance. However, you may find you need to use the COM object to access some functionality or object members that are not yet exposed by the .NET API. In this case it's possible to use both. See Limitations and Using Interop (page 19) .

Organization

This guide is organized by the major features of AutoCAD Civil 3D. It consists of the following chapters, each of which includes samples from applicable APIs taken from one or more demonstration programs:

Chapter 1: Getting Started (page 9)

Explains how to set up a COM and .NET project. Also discusses porting COM to .NET code, limitations of the .NET API, and how to use interop to access COM objects.

Chapter 2: Root Objects and Common Concepts (page 21)

Explains how to obtain the base .NET objects representing the documents and databases, which are required for all subsequent chapters. Also discussed are objects common to many features, such as ambient settings and label styles.

Chapter 3: Surfaces (page 32)

Explains how to import surfaces from files, manipulate point data directly, insert breaklines, manage borders, modify contours, and analyze elevation and watershed information.

Chapter 4: Alignments (page 65)

Explains how to create alignments through layout functions, from polyline entities, or based on an offset from existing alignments, using the .NET API. Includes discussion of stations, design speeds, and superelevation.

Chapter 5: Profiles (page 78)

Explains the creation of profiles and profile styles, using the .NET API.

Chapter 6: Pipe Networks (page 92)

Explains the creation of pipes, structures, and pipe networks, including interference detection, using the .NET API.

Chapter 7: Corridors (page 108)

Explains how to gather and modify information about existing corridors, baselines, feature lines, assemblies and subassemblies in a document, using the .NET API.

Chapter 8: Points (page 128)

Explains how to access and create points in a document's point collection, how to create point groups and populate them using queries, and how to style points using point description keys.

Chapter 9: Creating Custom Subassemblies (page 143)

Explains how to create and install custom subassemblies using Visual Basic .NET and the creation of catalog files which enables users to access custom subassemblies. You can also convert subassemblies written in VBA to .NET (see the Appendix Converting VBA Subassemblies to .NET (page 191) for more information).

Appendix A: Converting VBA Subassemblies to .NET (page 191)

Explains how to convert legacy custom subassemblies written in Visual Basic for Applications (VBA) to .NET.

Appendix B: COM API (page 203)

Covers the Legacy COM API.

New Features in the .NET API

This section covers changes to the .NET API for AutoCAD Civil 3D for the 2013 release.

Points

COGO points and related features are now fully supported in the .NET API with these new classes:

- `CogoPoint` class: exposes Coordinate Geometry (COGO) points , including styles and labels.
- `PointGroup` class: exposes point groups.
  - `CivilDocument.PointGroups` is the collection of all point groups in the drawing.
  - `CogoPoint.PrimaryPointGroupId` indicates the primary point group a `CogoPoint` belongs to.
- Point Group Queries: Standard and custom point group queries are exposed: `StandardPointGroupQuery` and `CustomPointGroupQuery` classes.
- User-defined properties are exposed. You can create/modify UDPs, and they can be set on `CogoPoint` objects. The collection of all point user-defined properties defined in a drawing is accessed via the `CivilDocument.PointUDPs` property.

- Description keys are exposed with the `PointDescriptionKey` class. The collection of all point description key sets in a drawing is accessed with the static `PointDescriptionKeySetCollection.GetPointDescriptionKeySets()` method.

Surfaces

Additional changes have been made to the Surface API:

- You can now calculate the bounded volume of a surface with `Surface.GetBoundedVolumes()`.
- The `SurfaceOperationAdd3DFaces` Surface operation is now exposed.
- You can now sample Surface Points along linear entities with the `SampleElevation()` method.
- You can now extract contour information from a surface with several new `ExtractContours*()` and related methods.
- You can now extract elements from surface objects (`ExtractBorder()`, `ExtractWatershed()`, and `ExtractGridded()`).

Profile Views

Stacked and multiple profile views can now be created using `ProfileView.Create()`.

Labels

The following changes have been made to the Labels API in this release:

- Parcel Area labels are exposed (`ParcelAreaLabel` class). These labels are obtained from the `Parcel.GetAvailableParcelAreaLabelIds()` method.
- Pipe labels are exposed: `PipeLabel`, `SpanningPipeLabel`, `PipeProfileLabel`, `PipeSectionLabel`, `SpanningPipeProfileLabel`, `StructureLabel`, `StructureProfileLabel` and `StructureSectionLabel` classes.
- Catchment labels are exposed: `CatchmentAreaLabel` and `FlowSegmentLabel`.
- Sampleline, Section, and SectionView labels: `SectionViewDepthLabel`, `SectionViewOffsetElevationLabel`, `SectionProjectionLabel`, `SectionLabelSetItem`, `SectionSegmentLabelGroup`, `SectionGradeBreakLabelGroup`, `SectionDataBandLabelGroup`, `SectionSegmentBandLabelGroup`, `SampleLineLabelGroup`.
- Plan production labels: `ViewFrameLabelGroup` and `MatchLineLabelGroup`

- General labels: `NoteLabel`, `GeneralSegmentLabel`
- Intersection Label: `IntersectionLocationLabel`
- The `GetAvailableLabelGroups()` method for existing `*LabelGroup` classes is deprecated in favor of the more accurately named `GetAvailableLabelGroupIds()`.

Transportation

The following changes have been made to the Transportation-related API in this release:

- You can access FeatureLines by code and modify their properties. You can also export Corridor FeatureLines as Polylines, Grading FeatureLines, Alignments and Profiles.
- You can export a `Corridor` as COGO points.
- You can get the offset for a `FeatureLinePoint`.
- You can get/set corridor region lock options while creating a Corridor though the API.
- Cant View is exposed, including command settings and style.
- Cant Label is exposed. You can now get/create the cant label object and access cant label's properties.
- You can get/set Cant Option and Rail Alignment Options for an Alignment.

Survey

The following changes have been made to the Survey API:

- The Survey project collection is now exposed (`SurveyProjectCollection` class) . It can be obtained from the `CivilApplication::SurveyProjects()` method. Users can now set/get the current working folder for survey projects, get the currently opened survey project object, get a survey project object from the project guid or path.
- The Survey project object is exposed (`SurveyProject` class). Users can now open and close the project, get the project's guid and name, and get the survey project's queries.
- SurveyQueries are exposed (`SurveyQueryCollection` class and `SurveyQuery` class). Users can now get the survey query object with the query guid and access the survey query's properties.
- Survey query definitions for TIN Surfaces are exposed, including AddPoint and AddFigure survey query definitions

(`SurveyQueriesAddPointDefinition`, `SurveyQueriesAddFigureDefinition`, `SurfaceOperationAddPointSurveyQuery`, and `SurfaceOperationAddFigureSurveyQuery` class). Users can now add data for a `TinSurface` with a survey query, and access the properties of a surface operation with a survey query.

.NET Namespace Changes

This release introduces a namespace restructuring that simplifies referencing objects. The "domain" part of of the namespace has been removed, so for existing projects, you will have to update your "using" statements to include the new namespace. As an example, previous releases exposes "Land" related classes and types in the `Autodesk.Civil.Land.DatabaseServices` namespace. In AutoCAD Civil 3D 2013, these classes and types are in the `Autodesk.Civil.DatabaseServices` namespace.

COM Changes

If you are using the COM API, you need to update the object version to 10.0 (from 9.0 used in AutoCAD Civil 3D 2012). The objects and interfaces exposed have remained the same, but you should reference the new libraries, which are installed by default to: "C:\Program Files\Common Files\Autodesk Shared\Civil Engineering 100".

In addition, interop DLLs are no longer registered as Primary Interop Assemblies (PIAs), and are deployed in the AutoCAD Civil 3D installation directory rather than the Global Assembly Cache (GAC). This means that these assemblies must now be added to Visual Studio projects using the Browse tab of the Add Reference dialog, rather than from the COM tab as was done previously.

To compile previously written projects against AutoCAD Civil 3D 2013, you will need to remove references to all interop assemblies from your project, and then re-add them using the Browse tab.

The assemblies required for COM interop are:

- Autodesk.AEC.Interop.Base
- Autodesk.AEC.Interop.UiBase
- Autodesk.AutoCAD.Interop
- Autodesk.AutoCAD.Interop.Common
- Autodesk.AECC.Interop.<domain>
- Autodesk.AEC.Interop.Ui<domain>

Where <domain> is one of the four Civil 3D COM domains: Land, Roadway, Pipe, or Survey.

We recommend that you set the "Embed Interop Types" property for each interop assembly to True, as this will embed all referenced types into your target assembly, and the referenced interop DLLs are therefore not required at runtime.

Also see the for detailed information about these new APIs, and items that have been deprecated in this release.

Legal Notices

AutoCAD Civil 3D 2013

Getting Started

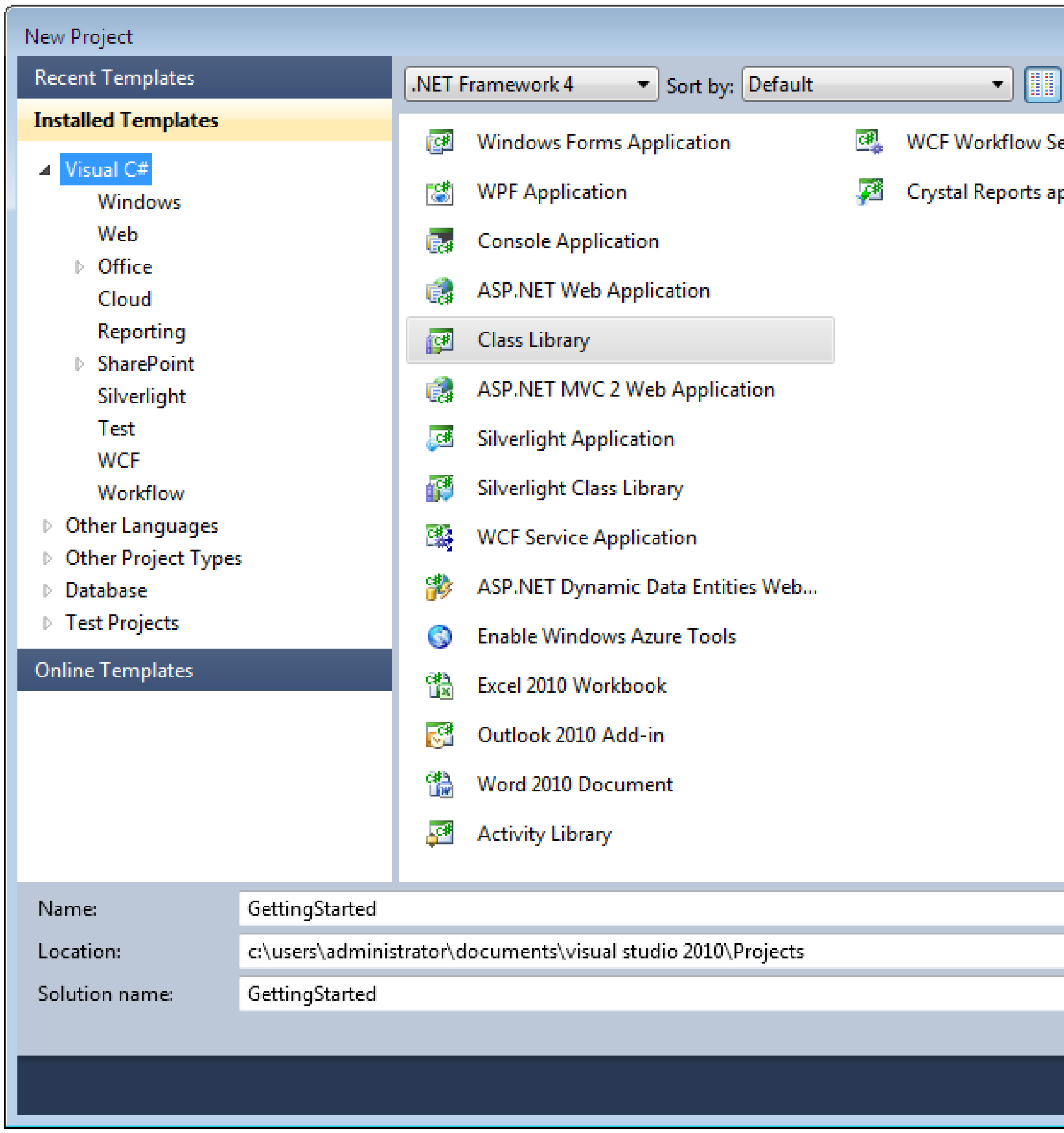
This chapter describes how to use VBA, and how to set up a new project for using the COM or .NET APIs for AutoCAD Civil 3D.

Setting up a .NET Project for AutoCAD Civil 3D

This section describes the basic steps to set up a .NET solution using Visual Studio and the AutoCAD Civil 3D managed classes. The steps are the similar whether you use Microsoft Visual C# .NET or Visual Basic .NET. The example below uses C# in Visual Studio 2010. Express (free) versions of Visual Studio may look slightly different, but can also be used.

To create a new project that uses the AutoCAD Civil 3D managed classes in Microsoft Visual Studio

1 In Visual Studio 2010, create a new class library solution and project.



Name: GettingStarted

Location: c:\users\administrator\documents\visual studio 2010\Projects

Solution name: GettingStarted

New Project Dialog in Visual Studio

2 Select **Project** menu ➤ **Add References**, or right-click **References** in the Solution Explorer and select **Add References**.

3 Browse to the install directory for AutoCAD Civil 3D, and select the base libraries *acdbmgd.dll*, *acmgd.dll*, *accoremgd.dll*, *AecBaseMgd.dll*, and *AeccDbMgd.dll*.

   NOTE

   These are the base AutoCAD and AutoCAD Civil 3D managed libraries. Your .NET assembly can use classes defined in additional libraries.

   To allow debugging and reduce the disk space requirements for your projects, select these libraries in the Visual Studio Solution Explorer, and set the **Copy Local** property to **False**.

4 Optionally, you can configure your project to start AutoCAD Civil 3D when you run the application from Visual Studio, which is useful for debugging.

   NOTE

   This option is not avaiable in Express (free) versions of Visual Studio.

   1 On the project **Properties** page, select the **Debug** panel.
   2 Under **Start Action**, choose **Start external program**, and enter the path to the *acad.exe* executable in the AutoCAD Civil 3D directory.
   3 Under **Start Options**, fill in the **Command line arguments**: `/ld "C:\Program Files\AutoCAD Civil 3D 2013\AecBase.dbx" /p "<<C3D_Imperial>>"`
   4 Under **Start Options**, fill in the **Working directory**, for example: `C:\Program Files\AutoCAD Civil 3D 2013\UserDataCache\`

5 Implement the `IExtensionApplication` interface in your main class. Add the `Autodesk.AutoCAD.Runtime` namespace (which is where this interface is defined), and `IExtensionApplication` after your class definition: Visual Studio will provide a code complete option to implement stubs for the interface. Your code should now look like this:

```
using System;
using Autodesk.AutoCAD.Runtime;

namespace GettingStarted
{
    public class Class1 : IExtensionApplication
    {
        #region IExtensionApplication Members

        public void Initialize()
        {
            throw new System.Exception("The method or
 operation is not implemented.");
        }

        public void Terminate()
        {
            throw new System.Exception("The method or
 operation is not implemented.");
        }

        #endregion
    }
}
```

You can remove or comment out the default content of these methods. `Initialize()` is called when your assembly is first loaded by a **NETLOAD** command in AutoCAD Civil 3D, and can be used for setting up resources, reading configuration files, and other initialization tasks. `Terminate()` is called when AutoCAD Civil 3D shuts down (there is no **NETUNLOAD** command to unload .NET assemblies), and can be used for cleanup to free resources.

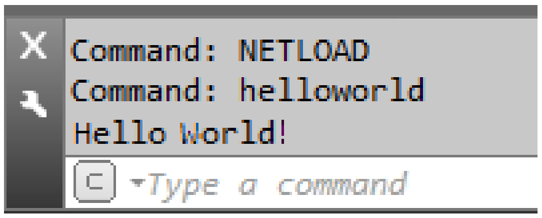
6 You are now ready to create a public method that is the target of a `CommandMethod` attribute. This attribute defines the AutoCAD Civil 3D command that invokes the method. For example:

```
[CommandMethod("HelloWorld")]
public void HelloWorld()
{

}
```

7 Let's make the method print out a "Hello World" message on the command line. Add the `Autodesk.AutoCAD.ApplicationServices` namespace, and add this line to the `HelloWorld()` method:

```
Application.DocumentManager.MdiActiveDocument.Editor.WriteMessage("\nHello
 World!\n");
```

You can now build the assembly and run it. Start AutoCAD Civil 3D, and type **NETLOAD** at the command line. In the **Choose .NET Assembly** dialog, browse to your assembly DLL (if you are using the project settings from step 1, this will be *GettingStarted.dll*). Type **HELLOWORLD** at the command line, and you will see the command output:

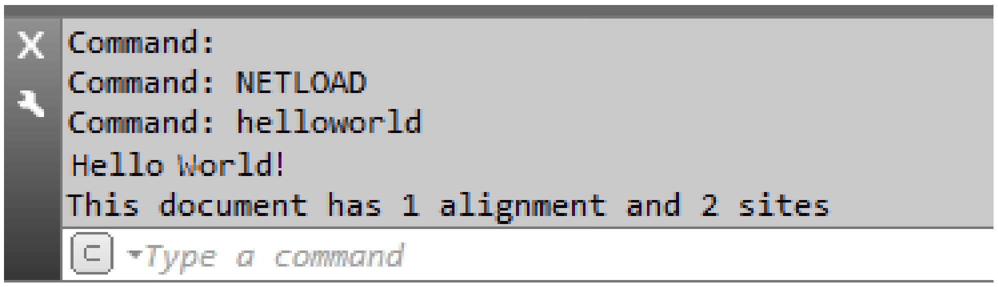


```
Command: NETLOAD
Command: helloworld
Hello World!
```

8 The previous step used functionality from the AutoCAD Application class. Let's include some functionality specific to the AutoCAD Civil 3D managed classes. First, add two more namespaces: `Autodesk.AutoCAD.DatabaseServices` and `Autodesk.Civil.ApplicationServices`. Then add these lines to obtain the current Civil document, get some basic information about it, and print the information out:

```
public void HelloWorld()
{
    CivilDocument doc =
Autodesk.Civil.ApplicationServices.CivilApplication.ActiveDocument;
    ObjectIdCollection alignments =
doc.GetAlignmentIds();
    ObjectIdCollection sites = doc.GetSiteIds();
    String docInfo = String.Format("\nHello
World!\nThis document has {0} alignments and {1}
sites.\n", alignments.Count, sites.Count);
   Application.DocumentManager.MdiActiveDocument.Editor.WriteMessage(docInfo);

}
```

Open or create a document in AutoCAD Civil 3D that contains alignments and sites. When you run the **HELLOWORLD** command now, you should see output similar to this:



```
Command:
Command: NETLOAD
Command: helloworld
Hello World!
This document has 1 alignment and 2 sites
```

For more samples, look in the AutoCAD Civil 3D*\samples\dotNet* directory.

Running Commands from the Toolbox

The recommended method to expose a AutoCAD Civil 3D extension to users is to add it to the **Toolbox** tab in **Toolspace**, by creating a toolbox macro. The **Toolbox** handles loading the .NET assembly or ARX DLL containing the commands.

There are two execution types that a toolbox macro can have:

1. CMD - the command name is sent to the command line to execute. This is the recommended execution type for both .NET and ARX commands.
2. .NET - a method name is located, via Reflection, in the assembly, and is executed directly. No attribute flags are read and the code is always run in application context. (A command executed from the command line runs in the drawing context by default). Therefore, code run as as a .NET execute type must always be a static method, and must handle its own document locking.

NOTE

It is safe to explicitly lock a document, even if the code might be run in document context.

Here is an example of how to handle document locking:

```
static void setPrecision()
{
    using (Autodesk.AutoCAD.ApplicationServices.DocumentLock
 locker =
Autodesk.AutoCAD.ApplicationServices.Application.DocumentManager.MdiActiveDocument.LockDocument())
```

```
    {
        // perform any document / database modifications
here
    CivilApplication.ActiveDocument.Settings.DrawingSettings.AmbientSettings.Station.Precision.Value
 = 2;
    }
}
```

Running Commands from the Toolbox

To create a toolbox macro for a compiled command using the Toolbox Editor

1. Click the **Toolbox** tab in Toolspace.
2. Click to open the **Toolbox Editor**.
3. Right-click **Miscellaneous Utilities** and click **New Category**.
4. Right-click the new category, and click **New Tool**.
5. Select the new tool, and enter its name.
6. For **Execute Type**, click the drop-down and select **CMD** or **.NET**.

   NOTE

   CMD is the recommended execution type in most cases, because you do not need to explicitly handle document locking. See the discussion above.

7. For **Execute File**, browse to the .NET assembly or ARX DLL that contains the command.
8. For **Macro Name**, enter:
   - Name of the command to run, if the execute type is **CMD**
   - Name of the method to run, if the execute type is **.NET**.
9. Optionally, enter a help file and help topic for the command.
10. Click ✓ to apply the changes and close the editor.

After a command has been set up, it can be run by right-clicking it and clicking **Execute**.

Migrating COM code to .NET

In the majority of cases, the .NET API mirrors the structure of the COM API, so porting code to .NET involves setting up a .NET project, copying the code lines, and renaming classes and methods to match the .NET names. If you are using C# instead of VB.NET, some additional changes to code structure are required. The following sections describe some of the differences in the two APIs.

Base Objects

The COM API requires that you access the AcadApplication object (via the `ThisDrawing.Application` object), get the interface object for the `AeccApplication` object, and from that get the active document. In the .NET API you import `Autodesk.Civil.ApplicationServices` namespace, and get the active document from the CivilApplication class:

```
g_oDocument = CivilApplication.ActiveDocument()
```

There is a single root document object, `CivilDocument`, instead of four domain-specific root objects for Land, Pipe, Roadway and Survey.

Transactions and ObjectIds

In the .NET API, code that reads and writes to root Civil documents need to use an `Autodesk.AutoCAD.DatabaseServices.TransactionManager` object to start and commit transactions. It's a best practice to manage a `Transaction` with a `Using` statement, which automatically disposes the Transaction at the end of the block; otherwise, the `Transaction` should be explicitly disposed of in the `Finally` section of a `Try-Finally` block. Here's an example of a `Transaction` in a `Using` block:

```
using (Transaction
trans=TransactionManager.StartTransaction())
{
  //operation here
  trans.Commit();
}
```

NOTE

See the section *Use Transactions with the Transaction Manager* in the *AutoCAD .NET Developer's Guide* for more information about using the `Transaction` object to open and modify drawing database objects.

In the .NET API, objects that you get from collections are, in most cases, type `ObjectId`, which have to be cast to their intended type using a `Transaction` object (returned by `TransactionManager.StartTransaction()`). Here's an example:

```
m_AligmentStyleId =
m_doc.Styles.AlignmentStyles.Item(sStyleName)
oAlignmentStyle = m_trans.GetObject(m_AligmentStyleId,
OpenMode.ForWrite) As AlignmentStyle
```

Styles

In the COM API, styles are held by the root document object. In the .NET API, they are located under `CivilDocument.Styles`, which is an object of type `StylesRoot` and contains style objects inherited from `StyleBase`. Getting and setting style attributes for `StyleBase` objects requires using a `GetDisplayStyle*()` method rather than a property. Here's an example from COM VBA:

```
oAlignmentStyle.ArrowDisplayStyle2d.Visible = False
oAlignmentStyle.ArrowDisplayStyle3d.Visible = False
' Display curves using violet.
oAlignmentStyle.CurveDisplayStyle2d.color = 200 ' violet
oAlignmentStyle.CurveDisplayStyle3d.color = 200 ' violet
oAlignmentStyle.CurveDisplayStyle2d.Visible = True
oAlignmentStyle.CurveDisplayStyle3d.Visible = True
```

This is the equivalent code in VB.NET:

```
oAlignmentStyle.GetDisplayStyleModel(AlignmentDisplayStyleType.Arrow).Visible
 = False
oAlignmentStyle.GetDisplayStylePlan(AlignmentDisplayStyleType.Arrow).Visible
 = False
' Display curves using violet.
oAlignmentStyle.GetDisplayStyleModel(AlignmentDisplayStyleType.Curve).Color
```

```
 = Autodesk.AutoCAD.Colors.Color.FromRgb(191, 0, 255) '
violet
oAlignmentStyle.GetDisplayStylePlan(AlignmentDisplayStyleType.Curve).Color
 = Autodesk.AutoCAD.Colors.Color.FromRgb(191, 0, 255) '
violet
oAlignmentStyle.GetDisplayStyleModel(AlignmentDisplayStyleType.Curve).Visible
 = True
oAlignmentStyle.GetDisplayStylePlan(AlignmentDisplayStyleType.Curve).Visible
 = True
```

Settings

In the COM API, settings are accessed through the `AeccDatabase::Settings` object, which contains properties representing the settings object hierarchy. In the .NET API, you use a method to retrieve specific settings objects, for example:

```
SettingsPipeNetwork oSettingsPipeNetwork =
doc.Settings.GetFeatureSettings<SettingsPipeNetwork>() as
 SettingsPipeNetwork;
```

Properties

In the COM API, properties are usually simple built-in types, such as `double`, or types such as `BSTR` that map to built-in VBA types such as `String`. In the .NET API, most properties are one of the `Property*` classes that implement the `IProperty` interface. For these properties, you get or set the `Value` of the property. For example, this code in COM:

```
oLabelStyleLineComponent.Visibility = True
```

Becomes this in .NET:

```
oLabelStyleLineComponent.General.Visible.Value = true;
```

NOTE

There are a few other changes here: the `Visibility` property is renamed `Visible`, which has moved to a sub-property of `LabelStyleLineComponent` called `General`.

Limitations and Using Interop

The .NET API does not expose all the functionality of AutoCAD Civil 3D, and it exposes less than the COM API. The following areas are not yet exposed in .NET:

- Sites and Parcels
- Sections
- Data Bands
- Some labels

In addition, there are some areas in implemented functionality that are not yet complete:

- Pipes: interference checks (except interference check styles)
- Corridors:
  - creating new corridors
  - adding baselines to corridors
  - creating or modifying corridor boundaries or masks
  - computing cut and fill
  - setting the CodeSetStyle

If you require this functionality in your .NET project, you can use the corresponding COM objects.

To use AutoCAD Civil 3D COM APIs from .NET

1. Create a .NET solution and project.
2. Select **Add Reference** from the Project menu or Solution Explorer.
3. On the Browse tab, browse to the Civil 3D install directory, and select the following COM interop DLLs, where <domain> is the Civil domain you want to use (Land, Roadway, Pipe, or Survey):
   - Autodesk.AEC.Interop.Base

- Autodesk.AEC.Interop.UiBase
- Autodesk.AutoCAD.Interop
- Autodesk.AutoCAD.Interop.Common
- Autodesk.AECC.Interop.<domain>
- Autodesk.AECC.Interop.Ui<domain>

4. Select the references above, and set the "Copy Local" property to true, as this will embed all referenced types into your target assembly, and the referenced interop DLLs are therefore not required at runtime.
5. Add the `Autodesk.AutoCAD.Interop` and `Autodesk.AECC.Interop.Ui<domain>` namespaces to your `using` or `Imports` statement.

NOTE

You may see warnings about types not being found in various Autodesk.AutoCAD.Interop namespaces (warning type 1684). To disable these warnings, enter **1684** under Supress Warnings on the Build tab of the project's properties.

Here is a C# example of getting a count of point groups and surfaces from a document using COM interop:

```
string m_sAcadProdID = "AutoCAD.Application";
string m_sAeccAppProgId =
"AeccXUiLand.AeccApplication.10.0";
...
private void useCom()
{
    //Construct AeccApplication object, Document and
Database objects
    m_oAcadApp =
(IAcadApplication)System.Runtime.InteropServices.Marshal.GetActiveObject(m_sAcadProdID);

    if (m_oAcadApp != null)
    {
        m_oAeccApp =
(IAeccApplication)m_oAcadApp.GetInterfaceObject(m_sAeccAppProgId);
        m_oAeccDoc =
(IAeccDocument)m_oAeccApp.ActiveDocument;
```

```
        // get the Database object via a late bind
        m_oAeccDb =
(Autodesk.AECC.Interop.Land.IAeccDatabase)m_oAeccDoc.GetType().GetProperty("Database").GetValue(m_oAeccDoc,
 null);
        long lCount = m_oAeccDb.PointGroups.Count;
        m_sMessage += "Number of PointGroups = " +
lCount.ToString() + "\n";
        lCount = m_oAeccDb.Surfaces.Count;
        m_sMessage += "Number of Surfaces = " +
lCount.ToString() + "\n\n";
        MessageBox.Show(m_sMessage);
        m_sMessage = "";

    }
}
```

For more interoprability examples, see the CSharpClient and VbDotNetClient sample projects located in *<Install directory>\Sample\AutoCAD Civil 3D\COM\*.

Root Objects and Common Concepts

This chapter explains how to work with the root objects required to access all other objects exposed by the AutoCAD Civil 3D .NET API: `CivilApplication` and `CivilDocument`, as well as how to work with collections. It also describes how to work with settings and label styles.

Root Objects

This section explains how to acquire references to the base objects, which are required for all applications using the .NET API. It also explains the uses of the application, document, and database objects and how to use collections, which are commonly used throughout the .NET API. To help developers who are already familiar with COM to migrate existing code to .NET, the differences between the two APIs are highlighted with notes.

Accessing Application and Document Objects

The root object in the AutoCAD Civil 3D .NET hierarchy is the `CivilApplication` object. It contains a reference to the currently active document, and information about the running product.

NOTE

Unlike the COM API, `CivilApplication` does not inherit from the AutoCAD object `Autodesk.AutoCAD.ApplicationServices.Application`. Therefore, if you need access to application-level methods and properties (such as the collection of all open documents, information about the main window, etc.), you need to access through the AutoCAD `Application` object. See the ObjectARX Managed Class Reference in the ObjectARX SDK for information about this class.

The active `CivilDocument` object is accessed by importing the `AutodeskCivil.ApplicationServices` namespace, and getting the `CivilApplication.ActiveDocument` property.

This example demonstrates the process of accessing the `CivilApplication` and `CivilDocument` objects:

```
using Autodesk.Civil.ApplicationServices;
namespace CivilSample {
    class CivilExample {
       CivilDocument doc = CivilApplication.ActiveDocument;
    }
}
```

Using Collections Within the Document Object

The document object not only contains collections of AutoCAD Civil 3D drawing elements (such as points and alignments) but also objects that modify those elements (such as styles and label styles). Collections in `CivilDocument` are ObjectID collections (`Autodesk.AutoCAD.DatabaseServices.ObjectIdCollection`) for most objects. Objects in these collections must be retrieved with a `Transaction.GetObject()`, and cast to their type before they can be used.

NOTE

In the COM API, document objects are contained in collections of objects that do not need to be cast.

`ObjectIdCollection` objects implement the `IList` interface, and can be enumerated or accessed by index. Here's an example of iterating through the Corridor collection with `foreach`, and retrieving and casting the resulting `ObjectId` to a `Corridor` to access its methods and properties:

```
public static void iterateCorridors () {
    CivilDocument doc = CivilApplication.ActiveDocument;
    using ( Transaction ts =
Application.DocumentManager.MdiActiveDocument.
           Database.TransactionManager.StartTransaction()
 ) {
        foreach ( ObjectId objId in doc.CorridorCollection
 ) {
            Corridor oCorridor = ts.GetObject(objId,
OpenMode.ForRead) as Corridor;
        Application.DocumentManager.MdiActiveDocument.Editor.WriteMessage("Corridor:
 {0}\nLargest possible triangle side: {1}\n",
                oCorridor.Name,
oCorridor.MaximumTriangleSideLength);
        }
    }
}
```

For more information about `ObjectIdCollection`s, see the .

This example creates a new point style:

```
ObjectId pointStyleID = doc.Styles.PointStyles.Add("Name");
// Now a new point style is added to the collection of
styles,
// and we can modify it by setting the properties
// of the oPointStyle object, which we get from the
Transaction ts:
PointStyle oPointStyle = ts.GetObject(pointStyleID,
OpenMode.ForWrite) as PointStyle;
oPointStyle.Elevation = 114.6;
// You must commit the transaction for the add / modify
operation
```

```
// to take effect
ts.Commit();
```

If you attempt to add a new element with properties that match an already existing element, try to access an item that does not exist, or remove an item that does not exist or is in use, an error will result. You should trap the error and respond accordingly.

The following sample demonstrates one method of dealing with such errors:

```
// Try to access the style named "Name"
try {
    // This raises an ArgumentException if the item doesn't
     // exist:
    ObjectId pointStyleId = doc.Styles.PointStyles["Name"];
     // do something with the point style...
} catch ( ArgumentException e ) {
     ed.WriteMessage(e.Message);
}
```

Accessing and Using the Database Object

`CivilDocument` class does not expose the underlying database associated with the document. However, you can access the database from the `AutoCAD Application.DocumentManager.MdiActiveDocument.Database` object. The `Database` object contains references to AutoCAD Civil 3D entities, as well as base AutoCAD entities. See the in the ObjectARX SDK for information.

Settings

This section explains the purpose and use of the document settings objects, and covers changing general and specific settings.

Accessing Drawing, Feature, and Command Settings

Settings apply at three levels in AutoCAD Civil 3D:

1 Drawing level: there are drawing-wide settings, such as units and zone, abbreviations, etc. There are also ambient settings, which affect a variety

of AutoCAD Civil 3D behaviors. While these settings also apply drawing-wide, they can be overridden at the feature or command level.

2 Feature (object) level: ambient settings override drawing level ambient settings for that feature only. There are also feature-specific settings, such as default styles.

3 Command level: ambient settings can be set on a command-by-command basis. These settings override both drawing level and feature level settings.

For more information on settings in general, see Understanding Settings in the .

A document's settings are accessed through the properties of the `SettingsRoot` object, which is obtained from the `Document.Settings` property. This object contains the `DrawingSettings` property (type `SettingsDrawing`), which contains all the top-level ambient settings for the document. It also has the `GetSettings()` method, which gets feature and command settings.

Drawing settings and general ambient settings are in the `Autodesk.Civil.Settings` namespace, while feature and command settings are in the namespace for the related feature. For example, alignment-related ambient and command settings are in the `Autodesk.Civil.Land.Settings` namespace.

The following sample shows how to access the angle settings for alignments:

```
SettingsAlignment alignmentSettings =
doc.Settings.GetSettings<SettingsAlignment>();
Autodesk.Civil.Settings.SettingsAmbient.SettingsAngle
angleSettings = alignmentSettings.Angle;
ed.WriteMessage(@"Alignment settings:\n  Precision: {0}\n
  Rounding: {1}
Unit: {2}\n  Drop Decimal: {3}\n  DropZeros: {4}\n ",
    angleSettings.Precision.Value,
angleSettings.Rounding.Value,
    angleSettings.Unit.Value,
angleSettings.DropDecimalForWholeNumbers.Value,
    angleSettings.DropLeadingZerosForDegrees.Value);
```

The command settings apply to commands, and correspond to the settings in the **Commands** folder for each item in the AutoCAD Civil 3D**Toolspace Settings Tab**. Each command setting has a corresponding class named `SettingsCmd`*CommandName*. For example, the settings class corresponding to the `CreateAlignmentLayout` command is `SettingsCmdCreateAlignmentLayout`. As with other types of settings, you use the `CivilDocument.Settings.GetSettings()` method to access command settings objects in the document.

The following snippet determines what the “Alignment Type Option” is for the `CreateAlignmentLayout` command:

```
SettingsCmdCreateAlignmentLayout alignLayoutCmdSettings =

doc.Settings.GetSettings<SettingsCmdCreateAlignmentLayout>();

ed.WriteMessage(@"Alignment Layout Command settings:
AlignmentType: {0}  ",
   alignLayoutCmdSettings.AlignmentTypeOption.AlignmentType.Value
   );
```

The result of this code returns the current command setting:

Getting command settings

Label Styles

This section explains common features of label styles. It covers creating a new label style object, defining a label style, and using property fields in label style text strings. Details specific to each construct are covered in the appropriate chapters.

Creating a Label Style Object

All types of annotation for AutoCAD Civil 3D elements are governed by label styles, which are objects of type `LabelStyle`. A label style can include any number of text labels, tick marks, lines, markers, and direction arrows.

The following example creates a new label style object that can be used with points:

```
CivilDocument doc = CivilApplication.ActiveDocument;
ObjectId labelStyleId;
labelStyleId =
doc.Styles.LabelStyles.PointLabelStyles.LabelStyles.Add
    ("New Point Label Style");
```

Defining a Label Style

A label style consists of collections of different features of a label, called “components”. The collection of these components is accessed with the `LabelStyle::GetComponents()` method, which takes the type (`LabelStyleComponentType`) of component to get. The component types are:

- `Text`
- `Line`
- `Block` - symbols
- `Tick` - for both major and minor tick marks
- `ReferenceText`
- `DirectionArrow`

- `TextForEach`

Not all of these may be valid, depending on the label style type. For example, adding a tick mark component to a label style meant for a point has no visible effect. Label styles also depend on graphical objects that may or may not be part of the current document. For example, if the style references a block that is not part of the current document, then the specified block or tick components are not shown.

To add a feature to a label style, add a new component to the corresponding collection using the `LabelStyle::AddComponent()` method. Then set the properties of that component to the appropriate values. Always make sure to set the `Visible` property to `true`.

```
try
{
    // Add a line to the collection of lines in our label
 style
   ObjectId lineComponentId = oLabelStyle.AddComponent("New
 Line Component", LabelStyleComponentType.Line);
    // Get the new component:
    ObjectIdCollection lineCompCol =
oLabelStyle.GetComponents(LabelStyleComponentType.Line);
    var newLineComponent = ts.GetObject(lineComponentId,
OpenMode.ForWrite) as LabelStyleLineComponent;
    // Now we can modify the component
    newLineComponent.General.Visible.Value = true;
    newLineComponent.Line.Color.Value =
Autodesk.AutoCAD.Colors.Color.FromColorIndex(Autodesk.AutoCAD.Colors.ColorMethod.ByAci,
 40); // orange-yellow
   newLineComponent.Line.Angle.Value = 2.094; // radians,
 = 120 deg
    // negative lengths are allowed - they mean the line
is drawn
    // in the opposite direction to the angle specified:
    newLineComponent.Line.Length.Value = -0.015;
   newLineComponent.Line.StartPointXOffset.Value = 0.005;
   newLineComponent.Line.StartPointYOffset.Value = -0.005;
}
// Thrown if component isn't valid, or name is duplicated
catch (System.ArgumentException e)
{
  Application.DocumentManager.MdiActiveDocument.Editor.WriteMessage("Error:
 {0}\n", e.Message);
}
```

When first created, the label style object is set according to the ambient settings. Because of this, a new label style object may already contain features. If you are creating a new label style object, be sure to check for such existing features or your style might contain unintended elements.

```
  // Check to see whether any text components already exist.

  // If not, add one.
  if
  (oLabelStyle.GetComponentsCount(LabelStyleComponentType.Text)
   == 0)
  {
      // Add a text component
      oLabelStyle.AddComponent("New Text ",
  LabelStyleComponentType.Text);
  }
  // Now modify the first one:
  ObjectIdCollection textCompCol =
  oLabelStyle.GetComponents(LabelStyleComponentType.Text);
  var newTextComponent = ts.GetObject(textCompCol[0],
  OpenMode.ForWrite) as LabelStyleTextComponent;
```

The ambient settings also define which units are used. If you are creating an application designed to work with different drawings, you should take ambient settings into account or labels may demonstrate unexpected behavior in each document.

Using Property Fields in Label Style Text

Text within a label is designated by the `LabelStyleTextComponent.Contents` property, a `PropertyString` value. Of course, text labels are most useful if they can provide some sort of information that is unique to each particular item being labeled. This is accomplished by specifying property fields within the string. These property fields are of the form "<[*Property name*(*modifier 1*|[..] *modifier n*)]>". Modifier values are optional and can be in any order. Any number of property fields can be combined with normal text in the `Contents` property.

In this example, a string component of a label is modified to show design speeds and station values for a point along an alignment:

```
var newTextComponent = ts.GetObject(textCompCol[0],
OpenMode.ForWrite) as LabelStyleTextComponent;
newTextComponent.Text.Contents.Value = "SPD=<[Design
Speed(P0|RN|AP|Sn)]>";
newTextComponent.Text.Contents.Value += "STA=<[Station
Value(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>";
```

Valid property fields for each element are listed in the appropriate chapter.

Sharing Styles Between Drawings

Label styles, like all style objects, can be shared between drawings. To do this, call the style's `ExportTo()` method, targeting the drawing you want to add the style to.

NOTE

You can also export collections of styles to another drawing by using the static `StyleBase::ExportTo()` method.

When exporting styles, you specify how conflicts are resolved using the `StyleConflictResolverType` enum. In this example, the first style in the `MajorStationLabelStyles` collection is exported from the active drawing to another open drawing named *Drawing1.dwg*:

```
[CommandMethod("ExportStyle")]
public void ExportStyle()
{
    CivilDocument doc = CivilApplication.ActiveDocument;
    Document AcadDoc =
Application.DocumentManager.MdiActiveDocument;
    Database destDb = null;
    // Find the database for "Drawing 1"
    foreach (Document d in Application.DocumentManager)
    {
        if (d.Name.Equals("Drawing1.dwg")) destDb =
d.Database;
    }
    // cancel if no matching drawing:
    if (destDb == null) return;
    using (Transaction ts =
AcadDoc.Database.TransactionManager.StartTransaction())
     {
         // Export style:
         ObjectId styleId =
doc.Styles.LabelStyles.AlignmentLabelStyles.MajorStationLabelStyles[0];
         LabelStyle oLabelStyle = ts.GetObject(styleId,
OpenMode.ForRead) as LabelStyle;
         oLabelStyle.ExportTo(destDb,
Autodesk.Civil.StyleConflictResolverType.Rename);
     }
}
```

NOTE

In certain situations attempts to abort the transaction will fail when calling `ExportTo()`. This is the case when all the following conditions are all true: multiple styles are being exported, there is a naming conflict between styles, and the `StyleConflictResolverType` is `StyleConflictResolverType.Override`.

Sample Programs

BatchEditLabelTextSample

<installation-directory>\Sample\Civil 3D API\DotNet\CSharp\BatchEditLabelTextSample\

This sample prompts the user to select multiple alignment labels, then it prompts for some input text, and replaces the original text for all selected labels.

CommandSettingsSample

<installation-directory>\Sample\Civil 3D API\DotNet\CSharp\CommandSettingsSample\

This sample demonstrates how to iterate through all the command settings defined in a drawing using Reflection. All commands are exported to an xml file.

CompareStyles

<installation-directory>\Sample\Civil 3D API\DotNet\CSharp\CompareStyles\

This sample illustrates how to compare styles defined in two drawings.

DraggedLabelSample

<installation-directory>\Sample\Civil 3D API\DotNet\CSharp\DraggedLabelSample\

This sample illustrates how to add a new label in a dragged state.

EditLabelStyleSample

<installation-directory>\Sample\Civil 3D API\DotNet\CSharp\EditLabelStyleSample\

This sample illustrates how to batch copy label styles from one label to multiple selected labels.

Surfaces

This chapter covers Civil 3D Surface objects, and how to work with them using the AutoCAD Civil 3D .NET API.

There are four classes of surface in Civil 3D:

- `TinSurface`
- `GridSurface`
- `TinVolumeSurface`
- `GridVolumeSurface`

The first two represent a single layer of terrain, while the second two represent a volume between two layers. All four derive from a generic Surface object, which exposes the common methods and properties shared by all surfaces.

Accessing Surfaces

There are many ways to access the surfaces objects in a drawing. All the surfaces contained by a Document can be obtained using the `CivilDocument.GetSurfaceIds()` method, which returns an `ObjectIdCollection`.

```
ObjectIdCollection SurfaceIds = doc.GetSurfaceIds();
foreach (ObjectId surfaceId in SurfaceIds)
```

```
{
    CivSurface oSurface =
surfaceId.GetObject(OpenMode.ForRead) as CivSurface;
    editor.WriteMessage("Surface: {0} \n  Type: {1}",
oSurface.Name, oSurface.GetType().ToString());

}
```

Note that there is also a Surface class in the `Autodesk.AutoCAD.DatabaseServices` namespace, which will conflict with `Autodesk.Civil.DatabaseServices.Surface` if you use both namespaces. In this case you can fully qualify the Surface object, or use a "using" alias directive to disambiguate the reference. For example:

```
using CivSuface = Autodesk.Civil.DatabaseServices.Surface;
```

And then use the alias like this:

```
CivSuface oSurface = surfaceId.GetObject(OpenMode.ForRead)
 as CivSuface;
```

You can also prompt a user to select a specific surface type, such as a TIN Surface, and then get the surface ID from the selection:

```
private ObjectId promptForTinSurface(String prompt)
{
    PromptEntityOptions options = new PromptEntityOptions(

        String.Format("\n{0}: ", prompt));
    options.SetRejectMessage(
        "\nThe selected object is not a TIN Surface.");
    options.AddAllowedClass(typeof(TinSurface), true);

    PromptEntityResult result = editor.GetEntity(options);

    if (result.Status == PromptStatus.OK)
    {
        // We have the correct object type
        return result.ObjectId;
    }
    return ObjectId.Null;   // Indicating error.
}
```

Surface Properties

The Surface object exposes general surface properties, which you can access using the various `GetGeneralProperties()` methods. Calculating and returning properties is a resource-intensive process, so you are encouraged to call this method once and re-use the returned object, instead of calling the method for each property. TIN and Grid surfaces have type-specific properties (returned by `GetTinProperties()` and `GetGridProperties()` respectively). Both Tin and Grid surfaces also implement a `GetTerrainProperties()` method.

This example gets general properties for the first surface in the database, and then depending on the surface type, it gets the Tin or Grid surface properties:

```
[CommandMethod("SurfaceProperties")]
public void SurfaceProperties()
{
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())

    {
        try
        {
            // Get the first surface in a document
           // "doc" is the CivilApplication.ActiveDocument

            ObjectId surfaceId = doc.GetSurfaceIds()[0];
            CivSurface oSurface =
surfaceId.GetObject(OpenMode.ForRead) as CivSurface;

            // print out general properties:
            GeneralSurfaceProperties genProps =
oSurface.GetGeneralProperties();
            String propsMsg = "\nGeneral Properties for "
 + oSurface.Name;
            propsMsg += "\n-------------------";
            propsMsg += "\nMin X: " +
genProps.MinimumCoordinateX;
            propsMsg += "\nMin Y: " +
genProps.MinimumCoordinateY;
            propsMsg += "\nMin Z: " +
genProps.MinimumElevation;
            propsMsg += "\nMax X: " +
genProps.MaximumCoordinateX;
```

```
            propsMsg += "\nMax Y: " +
genProps.MaximumCoordinateY;
            propsMsg += "\nMax Z: " +
genProps.MaximumElevation;
            propsMsg += "\nMean Elevation: " +
genProps.MeanElevation;
            propsMsg += "\nNumber of Points: " +
genProps.NumberOfPoints;
            propsMsg += "\n--";

            editor.WriteMessage(propsMsg);

            // Depending on the surface type, let's look
at grid or TIN properties:

            if (oSurface is TinSurface)
            {
                TinSurfaceProperties tinProps =
((TinSurface)oSurface).GetTinProperties();
                propsMsg = "\nTIN Surface Properties for
" + oSurface.Name;
                propsMsg += "\n------------------";
                propsMsg += "\nMin Triangle Area: " +
tinProps.MinimumTriangleArea;
                propsMsg += "\nMin Triangle Length: " +
tinProps.MinimumTriangleLength;
                propsMsg += "\nMax Triangle Area: " +
tinProps.MaximumTriangleArea;
                propsMsg += "\nMax Triangle Length: " +
tinProps.MaximumTriangleLength;
                propsMsg += "\nNumber of Triangles: " +
tinProps.NumberOfTriangles;
                propsMsg += "\n--";

                editor.WriteMessage(propsMsg);
            }
            else if (oSurface is GridSurface)
            {
                #region GetGridProperties
                GridSurfaceProperties gridProps =
((GridSurface)oSurface).GetGridProperties();
                propsMsg = "\\Grid Surface Properties for
 " + oSurface.Name;
```

```
                propsMsg += "\n------------------";
                propsMsg += "\n X Spacing: " +
gridProps.SpacingX;
                propsMsg += "\n Y Spacing: " +
gridProps.SpacingY;
                propsMsg += "\n Orientation: " +
gridProps.Orientation;
                propsMsg += "\n--";

                editor.WriteMessage(propsMsg);
                #endregion
            }

        }
        catch (System.Exception e) {
editor.WriteMessage(e.Message); }
    }
}
```

Creating Surfaces

`GridSurface` and `TinSurface` objects can be created from an imported file, or created as a new, empty surface to which surface data can be added later. A new `TinSurface` can also be created by cropping existing `TinSurface` objects.

NOTE Import from LandXML data is not supported in the .NET API at this time. You can use the COM API to import or export surface data to or from LandXML.

Most methods for creating empty or importing surfaces are similar in that they all have two overloads: one that specifies the database where the surface will be created (with the default `SurfaceStyle` applied), the other specifies a `SurfaceStyle` to apply, and adds the surface to the database that contains the `SurfaceStyle`.

Volume surfaces are created from two existing surfaces, the base (bottom) surface and a comparison surface.

Creating a TIN Surface from a TIN file

You can create a new TIN surface from a binary .tin file using the `TinSurface.CreateFromTin()` method. This method takes two arguments, the database for the drawing to which the TIN surface will be added, and the path to a .tin file, as a string.

```
[CommandMethod("CreateFromTIN")]
public void CreateFromTin()
{
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())

    {
        // Example TIN surface from Civil Tutorials:
       string tinFile = @"C:\Program Files\Autodesk\AutoCAD
 Civil 3D 2013\Help\Civil Tutorials\Corridor surface.tin";

        try
        {
            Database db =
Application.DocumentManager.MdiActiveDocument.Database;
            ObjectId tinSurfaceId =
TinSurface.CreateFromTin(db, tinFile);
            editor.WriteMessage("Import succeeded: {0} \n
 {1}", tinSurfaceId.ToString(), db.Filename);
        }
        catch (System.Exception e)
        {
            // handle bad file path
            editor.WriteMessage("Import failed: {0}",
e.Message);
        }

        // commit the transaction
        ts.Commit();
    }
}
```

Creating a TIN Surface using `TinSurface.Create()`

You can create an empty TIN surface and add it to the document's surface collection with the `TinSurface.Create()` method. This method has two overloads, one that specifies the `SurfaceStyle` to apply, while the other uses the default style.

In this example, we create a new TIN surface with a specified style, and then add some random point data.

```
[CommandMethod("CreateTINSurface")]
public void CreateTINSurface()
{
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())

    {
        string surfaceName = "ExampleTINSurface";
        // Select a style to use
        ObjectId surfaceStyleId =
doc.Styles.SurfaceStyles[3];

        // Create the surface
       ObjectId surfaceId = TinSurface.Create(surfaceName,
 surfaceStyleId);

        TinSurface surface =
surfaceId.GetObject(OpenMode.ForWrite) as TinSurface;

        // Add some random points
       Point3dCollection points = new Point3dCollection();

        Random generator = new Random();
        for (int i = 0; i < 10; i++)
        {
            double x = generator.NextDouble() * 250;
            double y = generator.NextDouble() * 250;
            double z = generator.NextDouble() * 100;
            points.Add(new Point3d(x, y, z));
        }

        surface.AddVertices(points);

        // commit the create action
        ts.Commit();
    }
}
```

Creating a Grid Surface from a DEM File

You can create a GridSurface from a Digital Elevation Model (DEM) file using the GridSurface.CreateFromDEM() method. There are two overloads of this method: one that applies the default style, while the other allows you to specify the SurfaceStyle to use. Both take the filename and path of a DEM file, as a string.

```
[CommandMethod("CreateFromDEM")]
public void CreateFromDEM()
{
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())

    {
        // Prompt user for a DEM file:
        // string demFile = @"C:\Program
Files\Autodesk\AutoCAD Civil 3D 2013\Help\Civil
Tutorials\Corridor surface.tin";
        PromptFileNameResult demResult =
editor.GetFileNameForOpen("Enter the path and name of the
 DEM file to import:");
        editor.WriteMessage("Importing: {0}",
demResult.StringResult);

        try
        {
            // surface style #3 is "slope banding" in the
 default template
            ObjectId surfaceStyleId =
doc.Styles.SurfaceStyles[3];
            ObjectId gridSurfaceId =
GridSurface.CreateFromDEM(demResult.StringResult,
surfaceStyleId);
           editor.WriteMessage("Import succeeded: {0} \n",
 gridSurfaceId.ToString());
        }
        catch (System.Exception e)
        {
            // handle bad file data or other errors
            editor.WriteMessage("Import failed: {0}",
e.Message);
        }
        // commit the transaction
        ts.Commit();
    }
}
```

Creating a GridSurface with `GridSurface.Create()`

You can create an empty `GridSurface` using the `GridSurface.Create()` method. There are two overloads of this method: one applies the default `SurfaceStyle`, while the other allows you to specify which `SurfaceStyle` to use. Both take the name of the new `GridSurface`, x and y spacing, and orientation. The units for x and y spacing and orientation are specified in the surface creation ambient settings (`SettingsCmdCreateSurface` Distance and Area properties).

`GridSurface` objects are defined on a regularly-spaced grid, and each location on the grid (represented by the GridLocation structure) has a row index and column index. The grid address (0,0) is at the bottom left corner of the grid.

The following example creates a new, empty `GridSurface` with 25′ x 25′ spacing, and then iterates through a 10 x 10 grid and adds a random elevation at each sample location:

```
[CommandMethod("CreateGridSurface")]
public void CreateGridSurface()
{
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())

    {
        string surfaceName = "ExGridSurface";
        // Select a surface style to use
        ObjectId surfaceStyleId =
doc.Styles.SurfaceStyles["Slope Banding (2D)"];

        // Create the surface with grid spacing of 25' x
25', orientation 0 degrees:
       ObjectId surfaceId = GridSurface.Create(surfaceName,
 25, 25, 0.0, surfaceStyleId);
        GridSurface surface =
surfaceId.GetObject(OpenMode.ForWrite) as GridSurface;
```

```
        // Add some random elevations
        Random m_Generator = new Random();
        for (int i = 0; i < 10; i++)
        {
            for (int j = 0; j < 10; j++)
            {
                double z = m_Generator.NextDouble() * 10;
               GridLocation loc = new GridLocation(i, j);

                surface.AddPoint(loc, z);
            }
        }

        // commit the create action
        ts.Commit();
    }
}
```

Creating a Volume Surface

A volume surface represents the difference or composite between two TIN or grid surface areas in a document. You can create a volume surface using the `Create()` method for either the `TinVolumeSuface` or `GridVolumeSurface` class.

In this example, the user is prompted to select the base and comparison surfaces, and then a new `TinVolumeSurface` is created from them. The implementation of `promptForTinSurface()` is left out for clarity.

```
CommandMethod("CreateTinVolumeSurface")]
public void CreateTinVolumeSurface()
{
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())

    {
        string surfaceName = "ExampleVolumeSurface";
        // Prompt user to select surfaces to use
        // promptForTinSurface uses Editor.GetEntity() to
 select a TIN surface
        ObjectId baseId = promptForTinSurface("Select the
 base surface");
       ObjectId comparisonId = promptForTinSurface("Select
 the comparison surface");
```

```
        try
        {
            // Create the surface
            ObjectId surfaceId =
TinVolumeSurface.Create(surfaceName, baseId, comparisonId);

            TinVolumeSurface surface =
surfaceId.GetObject(OpenMode.ForWrite) as TinVolumeSurface;

        }

        catch (System.Exception e)
        {
            editor.WriteMessage("Surface create failed:
{0}", e.Message);
        }

        // commit the create action
        ts.Commit();
    }
}
```

Working with Surfaces

This section describes the various methods for adding and editing surface data. This includes adding a boundary, adding information to an existing surface from a DEM file, and using snapshots to improve surface performance.

Adding a Boundary

A boundary is a closed polygon that affects the visibility of the triangles inside it.

All boundaries applied to a surface are stored in the `Surface.BoundariesDefinition` collection. The boundary itself is defined by an AutoCAD entity such as a closed polyline or polygon. The height of the entity plays no part in how surface triangles are clipped, so you can use 2D or 3D entities. This entity can also contain curves, but the boundary always consists of lines. How these lines are tessellated is defined by the mid-ordinate distance, which is the maximum distance between a curve and the lines that are generated to approximate it.

You can add boundaries to a surface with its `BoundariesDefinition.AddBoundaries()` method. There are three overloads of this method that take one of these to define the new boundaries:

1 an `ObjectIdCollection` containing an existing polyline, polygon, or parcel
2 a `Point2dCollection`
3 a `Point3dCollection`

This method also specifies the boundary type (data clip, outer, hide, or show), whether non-destructive breaklines should be used, and the mid-ordinate distance value, which determines how lines are tessellated from curves.

In this example, the user is prompted to select a TIN surface and a polyline, and the polyline is added to the surface's boundaries collection. Note that the surface must be re-built after the boundary is added. The re-build icon in the Civil 3D GUI is not displayed when a surface's boundaries are modified using the .NET API.

```
[CommandMethod("AddSurfaceBoundary")]
public void AddSurfaceBoundary()
{
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())

    {
        // Prompt the user to select a surface and a
polyline
        ObjectId surfaceId = promptForEntity("Select the
surface to add a boundary to", typeof(TinSurface));
       ObjectId polyId = promptForEntity("Select the object
 to use as a boundary", typeof(Polyline));

        // The boundary or boundaries must be added to an
 ObjectIdCollection for the AddBoundaries method:
        ObjectId[] boundaries = { polyId };
        TinSurface oSurface =
surfaceId.GetObject(OpenMode.ForWrite) as TinSurface;

        try
        {
           oSurface.BoundariesDefinition.AddBoundaries(new
 ObjectIdCollection(boundaries), 100,
Autodesk.Civil.Land.SurfaceBoundaryType.Outer, true);
            oSurface.Rebuild();
        }

        catch (System.Exception e)
        {
          editor.WriteMessage("Failed to add the boundary:
 {0}", e.Message);
        }

        // commit the transaction
        ts.Commit();
    }
}
```

Adding Data from DEM Files

Any number of DEM files can be added to existing grid and TIN surfaces. When a DEM file is added to the `GridSurface.DEMFilesDefinition` or `TinSurface.DEMFilesDefinition` collection, its information is converted to an evenly spaced lattice of triangles that is added to the surface.

Improving Performance by Using Snapshots

A surface is made up of all the operations that modifiy the surface's triangles. If you rebuild the surface, re-performing all these operations can be slow. Snapshots can improve performance by recording the current state of all the triangles in a surface. Subsequent rebuilds start from the data of the snapshot, thus saving time by not performing complicated calculations that have already been done once. Surface objects have `CreateSnapshot()`, `RebuildSnapshot()`, and `RemoveSnapshot()` methods. Both `CreateSnapshot()` and `RebuildSnapshot()` will overwrite an existing snapshot.

NOTE

`RebuildSnapshot()` will cause an error if the snapshot does not exist. `CreateSnapshot()` and `RebuildSnapshot()` can also cause errors if the surface is out-of-date. You can check whether the Surface has a snapshot already with the `Surface.HasSnapshot` property. For example:

```
if (oSurface.HasSnapshot)
{
    oSurface.RemoveSnapshot();
}
oSurface.CreateSnapshot();
oSurface.RebuildSnapshot();
```

Working with TIN Surfaces

This section covers the various methods and properties for examining or modifying existing TIN surfaces, including adding new point data, adding breaklines, and adding contours.

Extracting Contours

The `ExtractContour()` and `ExtracBorder()` methods exposed in the COM API are not yet available in the .NET API.

Adding Point Data to a TIN Surface

There are two techniques for adding points that are unique to TIN surfaces:

1. Using point files
2. Using point groups

The `TinSurface.PointFilesDefinition` property contains the names of text files that contain point information. These text files must consist only of lines containing the point number, easting, northing, and elevation delineated by spaces. Except for comment lines beginning with "#", any other information will result in an error. Unlike TIN or LandXML files, text files do not contain a list of faces - the points are automatically joined into a series of triangles based on the document settings.

The method `PointFilesDefinition.AddPointFile()` takes the path to a point file, and the `ObjectId` of a `PointFileFormat` object. This object is obtained from the Database’s `PointFileFormatCollection` using the same string used to describe the format in the Civil 3D GUI:

This is an example of adding a PENZD format point file to an existing surface. The file in this example is from the Civil 3D tutorials directory.

```
[CommandMethod("SurfacePointFile")]
public void SurfacePointFile()
{
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())

    {
        // Select the first Surface in the document
        ObjectId surfaceId = doc.GetSurfaceIds()[0];
        TinSurface oSurface =
surfaceId.GetObject(OpenMode.ForRead) as TinSurface;

        try
        {
            // add points from a point file to the surface

            // this is the location of an example PENZD
file from the C3D tutorials, the actual path may
            // differ based on the OS
            string penzdFile = @"C:\Program
Files\Autodesk\AutoCAD Civil 3D 2013\Help\Civil
Tutorials\EG-Surface-PENZD (space delimited).txt";

            // get the point file format object, required
 for import:
            PointFileFormatCollection ptFileFormats =
PointFileFormatCollection.GetPointFileFormats(HostApplicationServices.WorkingDatabase);

            ObjectId ptFormatId = ptFileFormats["PENZD
(space delimited)"];


oSurface.PointFilesDefinition.AddPointFile(penzdFile,
ptFormatId);
        }

        catch (System.Exception e)
        {
            editor.WriteMessage("Failed: {0}", e.Message);

        }
```

```
            // commit the transaction
            ts.Commit();
        }
    }
```

Adding Points Using Point Groups

Although the `TinSurface` class exposes a point group collection (as the `SurfaceDefinitionPointGroups` property), the API doesn't support adding point groups to the collection at this time.

Smoothing a TIN Surface

Surface smoothing adds points at system-determined elevations using Natural Neighbor Interpolation (NNI) or Kriging methods, which results in smoothed contours with no overlapping. See the for more information about the two supported smoothing methods.

`TinSurface` objects expose these two smoothing operations with the `SmoothSurfaceByNNI()` and `SmoothSurfaceByKriging()` methods.

Setting up a smoothing operation takes a couple of steps:

1. Create a `SurfacePointOutputOptions` object.
2. Set the `OutputLocations` property (enumerated by `SurfacePointOutputLocationsType`) to specify the output locations. The other options you need to set on `SurfacePointOutputOptions` depend on what is specified for this setting:

    1. `EdgeMidPoints` – specifies the Edges property, an array of `TinSurfaceEdge` objects representing edges on the surface.
    2. `RandomPoints` – specifies the number of points (`RandomPointsNumber`) and output regions (`OutputRegions`, a Point3dCollection)
    3. `Centroids` – specifies the `OutputRegions` property.
    4. `GridBased` – sets the `OutputRegions` property, grid spacing (`GridSpacingX` and `GridSpacingY`), and grid orientation (`GridOrientation`).

3 If you are using the Kriging method, you need to also create a `KrigingMethodOptions` object to set the options for this method:

   1 `SemivariogramModel` property – set to one of the models enumerated by `KrigingSemivariogramType`.

   2 `SampleVertices` property – set to the collection of vertices to which to smooth (for example, you can use the `TinSurface.GetVerticesInsidePolylines()` to get this collection).

   3 Optionally set `NuggetEffect`, `VariogramParamA` and `VariogramParamC` depending on the model selected.

4 Pass the options to `SmoothSurfaceByNNI()` or `SmoothSurfaceByKriging()`. These methods return a `SurfaceOperationSmooth` object that includes the number of output points in the operation.

This example illustrates setting up and using the `SmoothSurfaceByNNI()` method, using the Centroids output location:

```
[CommandMethod("SmoothTinSurface")]
public void SmoothTinSurface()
{
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())

    {
        try
        {
            // Select a TIN Surface:
            ObjectId surfaceId = promptForEntity("Select
a TIN surface to smooth\n", typeof(TinSurface));
            TinSurface oSurface =
surfaceId.GetObject(OpenMode.ForWrite) as TinSurface;

            // Select a polyline to define the output
region:
            ObjectId polylineId = promptForEntity("Select
 a polyline to define the output region\n",
typeof(Polyline));
            Point3dCollection points = new
Point3dCollection();
            Polyline polyline =
polylineId.GetObject(OpenMode.ForRead) as Polyline;

            // Convert the polyline into a collection of
points:
            int count = polyline.NumberOfVertices;
            for (int i = 0; i < count; i++)
            {
                points.Add(polyline.GetPoint3dAt(i));
            }

            // Set the options:
            SurfacePointOutputOptions output = new
SurfacePointOutputOptions();
            output.OutputLocations =
SurfacePointOutputLocationsType.Centroids;
           output.OutputRegions = new Point3dCollection[]
 { points };

            SurfaceOperationSmooth op =
oSurface.SmoothSurfaceByNNI(output);

            editor.WriteMessage("Output Points: {0}\n",
op.OutPutPoints.Count);

            // Commit the transaction
            ts.Commit();
        }
        catch (System.Exception e) {
editor.WriteMessage(e.Message); }
    }
}
```

Adding A Breakline to a TIN Surface

Breaklines are used to shape the triangulation of a TIN surface. Each TIN surface has a collection of breaklines contained in the `TinSurface.BreaklinesDefinition` property, which is a `SurfaceDefinitionBreaklines` object. There are different kinds of breaklines, and each is created in a slightly different way.

NOTE For more information about breakline types, see the

The `SurfaceDefinitionBreaklines` class allows you to add standard, non-destructive, and proximity breaklines in similar ways. Each breakline type has its own `Add*()` method (for example, `AddStandardBreaklines` for standard breaklines), and each method has three versions, depending on the type of object you are creating a breakline from. You can add breaklines from a `Point2dCollection`, a `Point3dCollection`, or an `ObjectIdCollection` that contains one or more 3D lines, grading feature lines, splines, or 3D polylines. Each type of breakline requires a specified mid-ordinate distance parameter, which determines how curves are tessellated.

A **standard breakline** consists of an array of 3D lines or polylines. Each line endpoint becomes a point in the surface and surface triangles around the breakline are redone. The `AddStandardBreaklines()` method requires that you specify the maximum distance, weeding distance and weeding angle, in addition to the mid-ordinate distance. The `maximumDistance` parameter corresponds to the Supplementing Distance in the AutoCAD Civil 3D GUI, while `weedingDistance` and `weedingAngle` correspond to the weeding distance and angle, respectively.

A **proximity breakline** does not add new points to a surface. Instead, the nearest surface point to each breakline endpoint is used. The triangles that make up a surface are then recomputed making sure those points are connected.

A **non-destructive breakline** does not remove any triangle edges. It places new points along the breakline at each intersection with a triangle edge and new triangles are computed.

This example illustrates how to add standard, non-destructive and proximity breaklines:

```
[CommandMethod("SurfaceBreaklines")]
public void SurfaceBreaklines()
{
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())

    {
        // Prompt the user to select a TIN surface and a
polyline, and create a breakline from the polyline

        ObjectId surfaceId = promptForEntity("Select a TIN
 surface to add a breakline to", typeof(TinSurface));
        ObjectId lineId = promptForEntity("Select a 3D
polyline to use as the breakline", typeof(Polyline3d));
        TinSurface oSurface =
surfaceId.GetObject(OpenMode.ForWrite) as TinSurface;
        ObjectId[] lines = { lineId };
        PromptKeywordOptions pKeyOpts = new
PromptKeywordOptions("");
        pKeyOpts.Message = "\nEnter the type of breakline
 to create: ";
        pKeyOpts.Keywords.Add("Standard");
        pKeyOpts.Keywords.Add("Non-Destructive");
        pKeyOpts.Keywords.Add("Proximity");
        pKeyOpts.Keywords.Default = "Standard";
        pKeyOpts.AllowNone = true;
       PromptResult pKeyRes = editor.GetKeywords(pKeyOpts);


        try
        {
            switch (pKeyRes.StringResult)
            {
                case "Non-Destructive":

oSurface.BreaklinesDefinition.AddNonDestructiveBreaklines(new
 ObjectIdCollection(lines), 1);
                    break;
                case "Proximity":

oSurface.BreaklinesDefinition.AddProximityBreaklines(new
ObjectIdCollection(lines), 1);
                    break;
                case "Standard":
                default:

oSurface.BreaklinesDefinition.AddStandardBreaklines(new
ObjectIdCollection(lines), 10, 5, 5, 0);
                    break;
            }
        }

        catch (System.Exception e)
        {
            editor.WriteMessage("Operation failed: {0}",
e.Message);
        }

        // commit the transaction
        ts.Commit();
        }
    }
```

Adding a Wall Breakline

A wall breakline is used when the height of the surface on one side of the breakline is different than the other side. The `AddWallBreaklines()` method creates two breaklines, one for the top of the wall and one for the bottom. However, you cannot have a perfectly vertical wall in a TIN surface. The first breakline is placed along the path specified, and the second breakline is very slightly offset to one side and raised or lowered by a relative elevation. There are two versions of `AddWallBreaklines()`: one takes a `WallBreaklineCreation` structure that sets one elevation for all vertices in the breakline, while `WallBreaklineCreationEx` specifies an elevation for each individual vertex. The `IsRightOffset` property indicates in which direction the wall at each entity endpoint is offset. If set to true, the offset is to the right as you walk along the breakline from the start point to the end.

Importing Breaklines from a File

You can import breaklines from a file in .FLT format, using `SurfaceDefinitionBreaklines.ImportBreaklineFromFile()`. When you import the file, all breaklines in the FLT file are copied into the surface as Add Breakline operations, and the link to the file is not maintained. The option available on the GUI to maintain a link to the file is not available via the API.

This sample shows how to import breaklines from a file named *eg1.flt*, and to get the first newly created breakline:

```
[CommandMethod("ImportBreaklines")]
public void ImportBreaklines()
{
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())

    {
        // Prompt the user to select a TIN surface and a
polyline, and create a breakline from the polyline

        ObjectId surfaceId = promptForEntity("Select a TIN
 surface to add a breakline to", typeof(TinSurface));
```

```
        TinSurface oSurface =
surfaceId.GetObject(OpenMode.ForWrite) as TinSurface;
        string breaklines = "eg1.flt";

oSurface.BreaklinesDefinition.ImportBreaklinesFromFile(breaklines);


        // commit the transaction
        ts.Commit();
    }
}
```

Adding Contours to a TIN Surface

A contour is an open or closed entity that describes the altitude of the surface along the entity. Contours must have a constant altitude. The z value of the first point of the entity is used as the altitude of entire entity, no matter what is specified in the following points. Contours also have settings that can adjust the number of points added to the surface - when you create a contour, you specify a weeding distance, a weeding angle, and a distance parameter. Points in the contour are removed if the distance between the points before and after is less than the weeding distance and if the angle between the lines before and after is less than the weeding angle. Each line segment is split into equal sections with a length no greater than the `maximumDistance` parameter. Any curves in the entity are also tessellated according to the mid-ordinate distance, just as with breaklines. The `maximumDistance` value has precedence over the weeding values, so it is possible that the final contour will have line segments smaller than the weeding parameters specify.

Contours can be added from a `Point2dCollection`, `Point3dCollection`, or an `ObjectIdCollection` containing polylines. You can optionally specify options for minimizing flat areas in a surface by passing a `SurfaceMinimizeFlatAreaOptions` object as a parameter to `SurfaceDefinitionContours.AddContours()`. For more information about the ways you can minimize flat areas, see “Minimizing Flat Areas in a Surface” in the Civil 3D user’s Guide.

The following sample demonstrates adding a contour to a surface from a polyline:

```
[CommandMethod("CreateContour")]
public void CreateContour()
{
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())

    {
        // Prompt the user to select a TIN surface and a
polyline, and create a contour from the polyline
        ObjectId surfaceId = promptForEntity("Select a TIN
 surface to add a contour to", typeof(TinSurface));
       ObjectId polyId = promptForEntity("Select a polyline
 to create a contour from", typeof(Polyline));
        TinSurface oSurface =
surfaceId.GetObject(OpenMode.ForWrite) as TinSurface;

        ObjectId[] contours = {polyId};

        oSurface.ContoursDefinition.AddContours(new
ObjectIdCollection(contours), 1, 85.5, 55.5, 0);

        // commit the transaction
        ts.Commit();
    }
}
```

Extracting Contours

Contours can be extracted as AutoCAD drawing objects from surfaces (both TIN and Grid) using versions of `ExtractContours()`, `ExtractContoursAt()`, `ExtractMajorContours()`, and `ExtractMinorContours()`. These methods are defined by the `ITerrainSurface` interface, and implemented by all Surface classes.

The four methods are very similar, but accomplish different things:

- `ExtracContours()` - this method extracts contours at a specified elevation interval, starting at the surface's lowest elevation.
- `ExtractContoursAt()` - this method extracts all contours at a single specified elevation.
- `ExtractMajorContours()` - this method extracts only major contours from a Surface.
- `ExtractMinorContours()` - this method extracts only minor contours from a Surface.

The `ExtractContours()` method has four versions, the simplest taking an interval parameter, and another taking an elevation range and interval. There are also versions of these two methods that take additional smoothing parameters to smooth the extracted polylines. The extracted contours are lightweight AutoCAD Polyline objects, and the method returns an ObjectIdCollection containing the IDs of all extracted objects. The objects are independent of the surface and can be manipulated without affecting the underlying surface.

This example creates a random surface, and then extracts contours in a couple of ways.

```
// Setup: creates a new, random surface
//
TinSurface surface = CreateRandomSurface("Example Surface");

// Extract contours and print information about them:
ObjectIdCollection contours;
double contourInterval = 50.0;
contours = surface.ExtractContours(contourInterval);
write("# of extracted contours: " + contours.Count + "\n");
int totalVertices = 0;
for (int i = 0; i < contours.Count; i++)
{
    ObjectId contourId = contours[i];

    // Contours are lightweight Polyline objects:
   Polyline contour = contourId.GetObject(OpenMode.ForRead)
 as Polyline;
    write(String.Format("Contour #{0} length:{1}, # of
vertices:{2}\n",
        i, contour.Length, contour.NumberOfVertices));
    totalVertices += contour.NumberOfVertices;
}

// Extract contours with smoothing:
contours = surface.ExtractContours(contourInterval,
ContourSmoothingType.AddVertices, 10);
int totalVerticesSmoothed = 0;
foreach (ObjectId contourId in contours)
{
   Polyline contour = contourId.GetObject(OpenMode.ForRead)
 as Polyline;
    totalVerticesSmoothed += contour.NumberOfVertices;
}

// Compare smoothing by adding vertices:
write(String.Format("Effects of smoothing:\n  total vertices
 no smoothing: {0}\n  total vertices with smoothing: {1}\n",

    totalVertices, totalVerticesSmoothed));

// Extract contours in a range:
double startRange = 130.0;
double endRange = 190.0;
contours = surface.ExtractContours(contourInterval,
startRange, endRange);

write("# of extracted contours in range: " + contours.Count
 + "\n");

// You can also extract contours in a range with smoothing:
// contours = surface.ExtractContours(contourInterval,
startRange, endRange,
//    ContourSmoothingType.SplineCurve, 10);
```

Surface Styles

This section describes how to create and apply styles to surface objects.

Creating and Changing a Style

Surface styles are stored in the `CivilDocument.Styles.SurfaceStyles` collection. To create a new style, call the `SurfaceStyleCollection.Add()` method with the name of your new style. The new style is created according to the document's ambient settings.

In addition to the properties inherited from `StyleBase`, a surface style consists of different objects governing the appearance of boundaries, contours, direction analysis, elevation analysis, grids, points, slope arrows, triangles, and watershed analysis. Usually a single style only displays some of these objects. When initially created, a style is set according to the document's ambient settings and may show some unwanted style elements. Always set the visibility properties of all style elements to ensure the style behaves as you expect.

Assigning a Style to a Surface

To assign a style to a surface, set the Surface object's `StyleId` property to the `ObjectId` of a valid `SurfaceStyle` object.

This example illustrates creating a new style, changing its settings, and then assigning it to the first surface in the document. Only the plan display settings are changed for brevity, though you would normally also change the model display settings as well.

```
 [CommandMethod("SurfaceStyle")]
public void SurfaceStyle()
{
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())

    {
        // create a new style called 'example style':
        ObjectId styleId =
doc.Styles.SurfaceStyles.Add("example style");

        // modify the style:
        SurfaceStyle surfaceStyle =
styleId.GetObject(OpenMode.ForWrite) as SurfaceStyle;

        // display surface triangles

surfaceStyle.GetDisplayStylePlan(SurfaceDisplayStyleType.Triangles).Visible
 = true;

surfaceStyle.GetDisplayStyleModel(SurfaceDisplayStyleType.Triangles).Visible
 = true;

        // display boundaries, exterior only:

surfaceStyle.GetDisplayStylePlan(SurfaceDisplayStyleType.Boundary).Visible
 = true;
       surfaceStyle.BoundaryStyle.DisplayExteriorBoundaries
 = true;
       surfaceStyle.BoundaryStyle.DisplayInteriorBoundaries
 = false;

        // display major contours:
```

```
surfaceStyle.GetDisplayStylePlan(SurfaceDisplayStyleType.MajorContour).Visible
 = true;

        // turn off display of other items:

surfaceStyle.GetDisplayStylePlan(SurfaceDisplayStyleType.MinorContour).Visible
 = false;

surfaceStyle.GetDisplayStylePlan(SurfaceDisplayStyleType.UserContours).Visible
 = false;

surfaceStyle.GetDisplayStylePlan(SurfaceDisplayStyleType.Directions).Visible
 = false;

surfaceStyle.GetDisplayStylePlan(SurfaceDisplayStyleType.Elevations).Visible
 = false;

surfaceStyle.GetDisplayStylePlan(SurfaceDisplayStyleType.Slopes).Visible
 = false;

surfaceStyle.GetDisplayStylePlan(SurfaceDisplayStyleType.SlopeArrows).Visible
 = false;

surfaceStyle.GetDisplayStylePlan(SurfaceDisplayStyleType.Watersheds).Visible
 = false;

        // do the same for all model display settings as
well


        // assign the style to the first surface in the
document:
        CivSurface surf =
doc.GetSurfaceIds()[0].GetObject(OpenMode.ForWrite) as
CivSurface;
        surf.StyleId = styleId;

        // commit the transaction
        ts.Commit();
    }
}
```

Surface Analysis

This section shows you how to access surface analysis data using the .NET API.

All surface analysis data is accessed with the `Surface.Analysis` property, which exposes data for contour, user-defined contour, direction, elevation, slope arrow, slope, and watershed analyses.

Creating an Elevation Analysis

An elevation analysis creates a 2-dimensional projection of a surface and then adds bands of color indicating ranges of altitude. Calling `Surface.Analysis.GetElevationData()` returns an array of `SurfaceAnalysisElevationData` objects, one for each elevation region created by the analysis, or an empty array if no analysis exists. Each elevation region represents a portion of the surface's total elevation. The collection lets you modify the color, minimum elevation, and maximum elevation of each region.

Note that each time a surface's elevation analysis is generated in the GUI, AutoCAD Civil 3D discards all existing elevation regions for the surface and creates a new collection of regions. Changes to the previous collection of `SurfaceAnalysisElevationData` objects are discarded.

The .NET API does not have an equivalent to the COM API `SurfaceAnalysisElevation.CalculateElevationRegions()` method, but you can implement one that does the same thing. This example shows one implementation, and the implemented method being used by a command:

```
/// <summary>
/// Calculates elevation regions for a given surface, and
 returns an array that can be passed
/// to Surface.Analysis.SetElevationData()
/// </summary>
/// <param name="surface">A Civil 3D Surface object</param>
/// <param name="steps">The number of elevation steps to
calculate</param>
/// <param name="startColor">The index of the start color.
  Each subsequent color index is incremeted by 2.</param>
/// <returns>An array of SurfaceAnalysisElevationData
objects.</returns>
private SurfaceAnalysisElevationData[]
CalculateElevationRegions(Autodesk.Civil.Land.DatabaseServices.Surface
 surface, int steps, short startColor)
{
    // calculate increments based on # of steps:
    double minEle =
surface.GetGeneralProperties().MinimumElevation;
    double maxEle =
surface.GetGeneralProperties().MaximumElevation;
    double incr = (maxEle - minEle) / steps;

    SurfaceAnalysisElevationData[] newData = new
SurfaceAnalysisElevationData[steps];
    for (int i = 0; i < steps; i++)
    {
        Color newColor =
Color.FromColorIndex(ColorMethod.ByLayer, (short)(100 + (i
 * 2)));
       newData[i] = new SurfaceAnalysisElevationData(minEle
 + (incr * i), minEle + (incr * (i + 1)), newColor);
    }

    return newData;
}

/// <summary>
/// Illustrates performing an elevation analysis
/// </summary>
[CommandMethod("SurfaceAnalysis")]
public void SurfaceAnalysis()
{
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())

    {
        // Select first TIN Surface
        ObjectId surfaceId = doc.GetSurfaceIds()[0];
        TinSurface oSurface =
surfaceId.GetObject(OpenMode.ForWrite) as TinSurface;

        // get existing analysis, if any:
        SurfaceAnalysisElevationData[] analysisData =
oSurface.Analysis.GetElevationData();
        editor.WriteMessage("Existing Analysis length:
{0}\n", analysisData.Length);
```

```
        SurfaceAnalysisElevationData[] newData =
CalculateElevationRegions(oSurface, 10, 100);

        oSurface.Analysis.SetElevationData(newData);

        // commit the transaction
        ts.Commit();
    }
}
```

Many elevation analysis features can be modified through the surface style. For example, you can use a number of pre-set color schemes (as defined in the `ColorSchemeType` enumeration).

Accessing a Watershed Analysis

A watershed analysis predicts how water will flow over and off a surface. The analysis data is managed by an object of type `SurfaceAnalysisWatershedDataCollection` returned by the `Surface.Analysis.GetWatershedData()` method.

The .NET API does not implement an equivalent to the COM API `AeccSurfaceAnalysisWatershed.CalculateWatersheds()` method, but you can use the `SurfaceAnalysis.GetWatershedData()` method to access watershed data from an existing analysis, and change properties (such as `AreaColor`) of watershed regions.

Each item in the `SurfaceAnalysisWatershedDataCollection` represents a watershed region. Depending on the nature of the drain target, each watershed region is a different type specified by the `WatershedType` enumeration. (For more information about watershed region types, see “Types of Watersheds” in the). Other properties, such as the region color, hatch pattern, description, and visibility, are all accessible.

This example illustrates reading the properties of an existing watershed analysis:

```
[CommandMethod("SurfaceWatershedAnalysis")]
public void SurfaceWatershedAnalysis()
{
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())
```

```
    {
        // Select first TIN Surface
        ObjectId surfaceId = doc.GetSurfaceIds()[0];
        CivSurface oSurface =
surfaceId.GetObject(OpenMode.ForRead) as CivSurface;

       SurfaceAnalysisWatershedDataCollection analysisData
 = oSurface.Analysis.GetWatershedData();
        editor.WriteMessage("Number of watershed regions:
 {0}\n", analysisData.Count);
       foreach (SurfaceAnalysisWatershedData watershedData
 in analysisData)
        {
            editor.WriteMessage("Data item AreaId: {0} \n"

                + "Description: {1}\n"
                + "Type: {2}\n"
                + "Drains into areas: {3}\n"
                + "Visible? {4}\n",
                watershedData.AreaID,
watershedData.Description, watershedData.Type,
              String.Join(", ", watershedData.DrainsInto),
 watershedData.Visible);
        }

        // commit the transaction
        ts.Commit();
    }
}
```

Calculating Bounded Volumes

The .NET API exposes the Civil 3D Bounded Volumes Utility as the `GetBoundedVolumes()` method for the `Surface` class, which means that bounded volumes can be calculated for both TIN and Grid surfaces. This method takes a `Point3dCollection` containing points that define the vertices of a polygon area, and an optional elevation datum. If you do not supply an elevation datum, the method uses 0.0. The first and last point in the vertices collection must be the same; that is, the polygon must be closed. The method returns a `SurfaceVolumeInfo` object that includes values for net volume, cut volume, and fill volume.

In this example, a sample TIN surface is created, a polygon inside the surface is defined, and both versions of the `GetBoundedVolumes()` method is called on the surface.

```
// Create a sample surface
ObjectId surfaceId = TinSurface.Create(_acaddoc.Database,
 "Example Surface");
TinSurface surface = surfaceId.GetObject(OpenMode.ForWrite)
 as TinSurface;

// Generates 100 random points between 0,100:
Point3dGenerator p3dgen = new Point3dGenerator();
Point3dCollection locations = p3dgen.AsPoint3dCollection();
surface.AddVertices(locations);

// Create a region that is a polygon inside the surface.
// The first and last point must be the same to create a
closed polygon.
//
Point3dCollection polygon = new Point3dCollection();
polygon.Add(new Point3d(20, 20, 20));
polygon.Add(new Point3d(20, 80, 15));
polygon.Add(new Point3d(80, 40, 25));
polygon.Add(new Point3d(20, 20, 20));
double elevation = 30.5;

SurfaceVolumeInfo surfaceVolumeInfo =
surface.GetBoundedVolumes(polygon, elevation);
write(String.Format("Surface volume info:\n  Cut volume:
{0}\n Fill volume: {1}\n Net volume: {2}\n",
    surfaceVolumeInfo.Cut, surfaceVolumeInfo.Fill,
surfaceVolumeInfo.Net));

// If you do not specify an elevation, 0.0 is used:
//
surfaceVolumeInfo = surface.GetBoundedVolumes(polygon);
write(String.Format("Surface volume info:\n  Cut volume:
{0}\n Fill volume: {1}\n Net volume: {2}\n",
    surfaceVolumeInfo.Cut, surfaceVolumeInfo.Fill,
surfaceVolumeInfo.Net));
```

Alignments

This chapter covers creating and using Alignments, Stations, and Alignment styles using the AutoCAD Civil 3D .NET API.

Basic Alignment Operations

Creating an Alignment

Alignments are usually created without being associated with an existing site. Each `CivilDocument` object has its own collection of alignments not associated with a site accessed with the `GetSitelessAlignmentIds()` method. There is also a collection of all alignments (siteless and associated with a site) accessed with `GetAlignmentIds()` method. Alignments can be moved into a site with the `Alignment.CopyToSite()` method. A siteless alignment can be copied from a sited alignment using `Alignment.CopyToSite()`, and passing `ObjectId.Null` or "" as the site.

Creating a New Alignment

The `Alignment` class provides versions of the `Create()` method to create a new Alignment object from a polyline, or without geometry data. There are two overloads for creating an alignment without geometry data. Both take a reference to the document object, and the name of the new alignment. One takes ObjectIds for the site to associate the alignment with (pass `ObjectId.Null` to create a siteless alignment), the layer to create the alignment on, the style to apply to the alignment, and the label set style to use. The other overload takes strings naming these items. Here's a simple example that creates a siteless alignment without geometry data:

```
// Uses an existing Alignment Style named "Basic" and Label
 Set Style named "All Labels" (for example, from
// the _AutoCAD Civil 3D (Imperial) NCS.dwt template.  This
 call will fail if the named styles
// don't exist.
// Uses layer 0, and no site (ObjectId.Null)
ObjectId testAlignmentID = Alignment.Create(doc, "New
Alignment", ObjectId.Null, "0", "Basic", "All Labels");
```

There are two overloads of the `create()` method for creating alignments from polylines. The first takes a reference to the `CivilDocument` object, a `PolylineOptions` object (which contains the ID of the polyline to create an alignment from), a name for the new alignment, the `ObjectID` of a layer to draw to, the `ObjectID` of an alignment style, and the `ObjectID` of a label set object, and returns the `OjectID` of the new Alignment. The second overload takes the same parameters, except the layer, alignment style, and label set are specified by name instead of `ObjectID`.

This code creates an alignment from a 2D polyline, using existing styles:

```
[CommandMethod("CreateAlignment")]
public void CreateAlignment()
{
    doc = CivilApplication.ActiveDocument;
    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;

    // Ask the user to select a polyline to convert to an
 alignment
    PromptEntityOptions opt = new
PromptEntityOptions("\nSelect a polyline to convert to an
 Alignment");
    opt.SetRejectMessage("\nObject must be a polyline.");
    opt.AddAllowedClass(typeof(Polyline), false);
    PromptEntityResult res = ed.GetEntity(opt);

    // create some polyline options for creating the new
alignment
    PolylineOptions plops = new PolylineOptions();
    plops.AddCurvesBetweenTangents = true;
    plops.EraseExistingEntities = true;
    plops.PlineId = res.ObjectId;

   // uses an existing Alignment Style and Label Set Style
 named "Basic" (for example, from
   // the Civil 3D (Imperial) NCS Base.dwt template.  This
 call will fail if the named styles
    // don't exist.
   ObjectId testAlignmentID = Alignment.Create(doc, plops,
 "New Alignment", "0", "Standard", "Standard");
}
```

Creating an Alignment Offset From Another Alignment

Alignments can also be created based on the layout of existing alignments. The `Alignment::CreateOffsetAlignment()` method creates a new alignment with a constant offset and adds it to the same parent site as the original alignment. The new alignment has the same name (followed by a number in parenthesis) and the same style as the original, but it does not inherit any station labels, station equations, or design speeds from the original alignment.

```
[CommandMethod("CreateOffsetAlignment")]
public void CreateOffsetAlignment()
{
    doc = CivilApplication.ActiveDocument;
    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())
    {
        // Ask the user to select an alignment to create
a new offset alignment from
        PromptEntityOptions opt = new
PromptEntityOptions("\nSelect an Alignment");
        opt.SetRejectMessage("\nObject must be an
alignment.");
        opt.AddAllowedClass(typeof(Alignment), false);
        ObjectId alignID = ed.GetEntity(opt).ObjectId;
        Alignment align = ts.GetObject(alignID,
OpenMode.ForRead) as Alignment;

        // Creates a new alignment with an offset of 10:
        ObjectId offsetAlignmentID =
align.CreateOffsetAlignment(10.0);
    }
}
```

Defining an Alignment Path Using Entities

An alignment is made up of a series of *entities*, which are individual lines, curves, and spirals that make up the path of an alignment. A collection of entities is held in the `Alignment::Entities` property, which is an

AlignmentEntityCollection object. This collection has a wide array of methods for creating new entities.

Here's a short code snippet that illustrates one of the methods for adding a FixedCurve entitiy to an alignment's Entities collection:

```
Int32 previousEntityId = 0;
Point3d startPoint = new Point3d(8800.7906, 13098.1946,
0.0000);
Point3d middlePoint = new Point3d(8841.9624, 13108.6382,
0.0000);
Point3d endPoint = new Point3d(8874.2664, 13089.3333,
0.0000);
AlignmentArc retVal =
myAlignment.Entities.AddFixedCurve(previousEntityId,
startPoint, middlePoint, endPoint);
```

Determining Entities Within an Alignment

Each of the entities in the `Alignment::Entities` collection is a type derived from the `Alignment::AlignmentEntity` class. By checking the `AlignmentEntity.EntityType` property, you can determine the specific type of each entity and cast the reference to the correct type.

The following sample loops through all entities in an alignment, determines the type of entity, and prints one of its properties.

```
[CommandMethod("EntityProperties")]
public void EntityProperties()
{
    doc = CivilApplication.ActiveDocument;
    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())
    {
        // Ask the user to select an alignment to get info
 about
        PromptEntityOptions opt = new
PromptEntityOptions("\nSelect an Alignment");
        opt.SetRejectMessage("\nObject must be an
alignment.");
```

```
        opt.AddAllowedClass(typeof(Alignment), false);
        ObjectId alignID = ed.GetEntity(opt).ObjectId;
        Alignment align = ts.GetObject(alignID,
OpenMode.ForRead) as Alignment;

        int i = 0;
        // iterate through each Entity and check its type
        foreach (AlignmentEntity myAe in align.Entities){
            i++;
            String msg = "";
            switch (myAe.EntityType)
            {
                case AlignmentEntityType.Arc:
                    AlignmentArc myArc = myAe as
AlignmentArc;
                    msg = String.Format("Entity{0} is an
Arc, length: {1}\n", i, myArc.Length);
                    break;

                case AlignmentEntityType.Spiral:
                    AlignmentSpiral mySpiral = myAe as
AlignmentSpiral;
                    msg = String.Format("Entity{0} is a
Spiral, length: {1}\n", i, mySpiral.Length);
                    break;
                // we could detect other entity types as
well, such as
               // Tangent, SpiralCurve, SpiralSpiral, etc.
                default:
                    msg = String.Format("Entity{0} is not
 a spiral or arc.\n", i);
                    break;

            }
            // write out the Entity information
            ed.WriteMessage(msg);
        }
    }
}
```

Each entity has an identification number contained in its `AlignmentEntity.EntityId` property. Each entity knows the numbers of the entities before and after it in the alignment, and you can access specific entities by identification number through the `AlignmentEntityCollection.EntityAtId()` method.

Stations

Modifying Stations with Station Equations

A station is a point along an alignment a certain distance from the start of the alignment. By default the station at the start point of an alignment is 0 and increases contiguously through its length. This can be changed by using *station equations*, which can renumber stations along an alignment. A station equation is an object of type `StationEquation` which contains a location along the alignment, a new station number basis, and a flag describing whether station values should increase or decrease from that location on. A collection of these station equations is contained in the `Alignment::StationEquations` property.

The following code adds a station equation to an alignment, starting at a point 80 units from the beginning of the alignment, and increasing in value:

```
StationEquation myStationEquation =
myAlignment.StationEquations.Add(80, 0,
StationEquationType.Increasing);
```

NOTE

Some functions, such as `Alignment::DesignSpeedCollection::GetDesignSpeed()`, require a "raw" station value without regard to modifications made by station equations.

Creating Station Sets

Alignment stations are usually labeled at regular intervals. You can compute the number, location, and geometry of stations taken at regular spacings by using the `Alignment::GetStationSet()` method. Overloads of this method return a collection of stations based on the type of station requested, and optionally major and minor intervals.

```
// Get all the potential stations with major interval =
100, and minor interval = 20
// Print out the raw station number, type, and location
Station[] myStations = myAlignment.GetStationSet(
StationType.All,100,20);
ed.WriteMessage("Number of possible stations: {0}\n",
myStations.Length);
foreach (Station myStation in myStations){

        ed.WriteMessage("Station {0} is type {1} and at
{2}\n", myStation.RawStation, myStation.StnType.ToString(),
 myStation.Location.ToString());
}
```

Specifying Design Speeds

You can assign design speeds along the length of an alignment to aid in the future design of a roadway based on the alignment. The collection of speeds along an alignment are contained in the `Alignment::DesignSpeeds` property. Each item in the collection is an object of type `DesignSpeed`, which contains a raw station value, a speed to be used from that station on until the next specified design speed or the end of the alignment, the design speed number, and an optional string comment.

```
// Starting at station 0 + 00.00
DesignSpeed myDesignSpeed = myAlignment.DesignSpeeds.Add(0,
 45);
myDesignSpeed.Comment = "Straigtaway";
// Starting at station 4 + 30.00
myDesignSpeed = myAlignment.DesignSpeeds.Add(430, 30);
myDesignSpeed.Comment = "Start of curve";
// Starting at station 14 + 27.131 to the end
myDesignSpeed = myAlignment.DesignSpeeds.Add(1427.131, 35);
myDesignSpeed.Comment = "End of curve";
// make alignment design speed-based:
myAlignment.UseDesignSpeed = true;
```

Finding the Location of a Station

You can find the point coordinates (northing and easting) of a station and offset on an alignment using the `Alignment::PointLocation()` method. The simplest version of the method takes a station and offset, and returns a northing and easting (as ref parameters).

Another version of this method takes a station, offset, and tolerance, and returns a northing, easting, and bearing (as ref parameters). The tolerance determines on which alignment entity the point is returned. If the tolerance is greater than the desired station minus the station at the alignment entity transition, the point will be reported on that entity. For example, consider an alignment made up of a tangent (length 240) and curve (length 260). Looking for the location of station 400, with tolerance = 0, will find a point on the curve. However, a tolerance of 200 will cause the method to report a point on the tangent, because 400 - 240 < 200.

Superelevation

Another setting that can be applied to certain stations of an alignment is the superelevation, used to adjust the angle of roadway section components for corridors based on the alignment. The inside and outside shoulders and road surfaces can be adjusted for both the left and right sides of the road.

NOTE

The Superelevation feature was substantially changed in AutoCAD Civil 3D 2013, and the API has also changed. The `Alignment::SuperelevationData` property and `Alignment::SuperElevationAtStation()` method have been removed. Existing code should be updated to use the new API, and access superelevation via `Alignment::SuperelevationCurves`, `Alignment::SuperelevationCriticalStations`, and `SuperelevationCriticalStationCollection::GetCriticalStationAt()`.

The superelevation data for an alignment is divided into discrete curves, called superelevation curves, and each superelevation curve contains transition regions where superelevation transitions from normal roadway to full superelevation and back (with separate regions for transition in, and transition out of superelvation). These regions are defined by “critical stations”, or stations where there is a transition in the roadway cross-section. The collection of superelevation curves for an alignment is accessed with the `Alignment::SuperelevationCurves` property, while all critical stations for all curves is accessed with the `Alignment::SuperelevationCriticalStations` property. The `SuperelevationCurves` collection is empty if superelevation curves have not been added to the alignment (either manually, or calculated by the Superelevation Wizard in the user interface). The `SuperelevationCriticalStations` collection contains default entites for the start and end stations of the Alignment if no superelevation data has been calculated for the curves.

An individual `SuperelevationCriticalStation` can be accessed through the `Alignment::SuperelevationCriticalStations::GetCriticalStationAt()` method.

In this code snippet, the collection of superelevation curves for an alignment is iterated, and information about each critical station in each curve is printed out. Note that you must specify the cross segment type to get either the slope or smoothing length, but you may not know which segment types are valid for the critical station. In this snippet, the code attempts to get all segment types, and silently catches the `InvalidOperationException` exception for invalid types.

```
[CommandMethod("GetSECurves")]
public void GetSECurves()
{
    doc = CivilApplication.ActiveDocument;
    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())
    {
        // get first alignment:
        ObjectId alignID = doc.GetAlignmentIds()[0];
        Alignment myAlignment = ts.GetObject(alignID,
OpenMode.ForRead) as Alignment;
        if (myAlignment.SuperelevationCurves.Count < 1)
        {
            ed.WriteMessage("You must calculate
superelevation data.\n");
            return;
        }
        foreach (SuperelevationCurve sec in
myAlignment.SuperelevationCurves)
        {
            ed.WriteMessage("Name: {0}\n", sec.Name);
            ed.WriteMessage("  Start: {0} End: {1}\n",
sec.StartStation, sec.EndStation);
            foreach (SuperelevationCriticalStation sest in
 sec.CriticalStations)
            {
                ed.WriteMessage("   Critical station: {0}
 {1} {2}\n", sest.TransitionRegionType,
                    sest.Station, sest.StationType);
                // try to get the slope:
                foreach (int i in
Enum.GetValues(typeof(SuperelevationCrossSegmentType)))
                {
                    try
                    {
                        // if this succeeds, we know the
segment type:
                        double slope =
sest.GetSlope((SuperelevationCrossSegmentType)i, false);
                        ed.WriteMessage("   Slope: {0}
Segment type:
{1}\n",slope,Enum.GetName(typeof(SuperelevationCrossSegmentType),i));
                    }
                        // silently fail:
                    catch (InvalidOperationException e) {
 }
                }
            }
        }
    }
}
```

Alignment Style

Creating an Alignment Style

Styles govern many aspects of how alignments are drawn, including direction arrows and curves, spirals, and lines within an alignment. All alignment styles are contained in the `CivilDocument.Styles.AlignmentStyles` collection. Alignment styles must be added to this collection before being used by an alignment object. A style is normally assigned to an alignment when it is first created, but it can also be assigned to an existing alignment through the `Alignment.StyleId` property.

```
[CommandMethod("SetAlignStyle")]
public void SetAlignStyle()
{
    doc = CivilApplication.ActiveDocument;
    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())
    {
        // Ask the user to select an alignment
        PromptEntityOptions opt = new
PromptEntityOptions("\nSelect an Alignment");
        opt.SetRejectMessage("\nObject must be an
alignment.\n");
        opt.AddAllowedClass(typeof(Alignment), false);
        ObjectId alignID = ed.GetEntity(opt).ObjectId;
        Alignment myAlignment = ts.GetObject(alignID,
OpenMode.ForWrite) as Alignment;
        ObjectId styleID =
doc.Styles.AlignmentStyles.Add("Sample alignment style");
        AlignmentStyle myAlignmentStyle =
ts.GetObject(styleID, OpenMode.ForWrite) as AlignmentStyle;
        // don't show direction arrows
     myAlignmentStyle.GetDisplayStyleModel(AlignmentDisplayStyleType.Arrow).Visible
 = false;
     myAlignmentStyle.GetDisplayStylePlan(AlignmentDisplayStyleType.Arrow).Visible
 = false;
        // show curves in violet
     myAlignmentStyle.GetDisplayStyleModel(AlignmentDisplayStyleType.Curve).Color
 = Color.FromColorIndex(ColorMethod.ByAci, 200);
     myAlignmentStyle.GetDisplayStylePlan(AlignmentDisplayStyleType.Curve).Color
 = Color.FromColorIndex(ColorMethod.ByAci, 200);
        // show straight sections in blue
     myAlignmentStyle.GetDisplayStyleModel(AlignmentDisplayStyleType.Line).Color
 = Color.FromColorIndex(ColorMethod.ByAci, 160);
     myAlignmentStyle.GetDisplayStylePlan(AlignmentDisplayStyleType.Line).Color
 = Color.FromColorIndex(ColorMethod.ByAci, 160);
        // assign style to an existing alignment
        myAlignment.StyleId = myAlignmentStyle.Id;
        ts.Commit();
        }
    }
```

Alignment Label Styles

The style of text labels and graphical markers displayed along an alignment are set by specifying a LabelSet (by name or ObjectID) when the alignment is first created with one of the `Alignment::Create()` methods, or by assigning the label set object to the `CivilDocument.Styles.LabelSetStyles.AlignmentLabelSetStyles` property. The `AlignmentLabelSetStyles` collection consists of separate sets of styles to be placed at major stations, minor stations, and where the alignment geometry, design speed, or station equations change.

Alignment labels are described in the `AlignmentLabelSetStyle` collection, which is a collection of `AlignmentLabelSetItem` objects. Labels at major stations are described by `AlignmentLabelSetItem` objects with a `LabelStyleType` property of `LabelStyleType.AlignmentMajorStation`. Minor station labels are described by `AlignmentLabelSetItem` objects with a `LabelStyleType` property of `LabelStyleType.AlignmentMinorStation`. Each `AlignmentLabelSetItem` object has a related `LabelStyle` object (which you can get or set with the `LabelStyleId` and `LabelStyleName` properties) and a number of properties describing the limits of the labels and the interval between labels along the alignment. When a new `AlignmentLabelSetItem` is created for a minor station label (using `BaseLabelSetStyle.Add()`), it must reference a parent major station label `AlignmentLabelSetItem` object.

Labels may be placed at the endpoints of each alignment entity. Such labels are controlled through the `AlignmentLabelSetItem.GetLabeledAlignmentGeometryPoints()` and `AlignmentLabelSetItem.GetLabeledAlignmentGeometryPoints()` methods. These methods also access labels at each change in alignment design speeds and station equations. The get method returns a Dictionary hash object: `Dictionary<AlignmentPointType, bool>`, specifiying the location of the geometry point, and the bool indicates whether the point is labeled.

Label text for all label styles at alignment stations is controlled by a `LabelStyle` object's Text component, which is set by the `LabelStyle.SetComponent()`

method. The following list of property fields indicates valid values for the Text component:

Valid property fields for LabelStyleComponentType.Text Contents
<[Station Value(Uft|FS|P0|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Raw Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Northing(Uft|P4|RN|AP|Sn|OF)]>
<[Easting(Uft|P4|RN|AP|Sn|OF)]>
<[Design Speed(P3|RN|AP|Sn|OF)]>
<[Instantaneous Direction(Udeg|FDMSdSp|MB|P4|RN|DSn|CU|AP|OF)]>
<[Perpendicular Direction(Udeg|FDMSdSp|MB|P4|RN|DSn|CU|AP|OF)]>
<[Alignment Name(CP)]>
<[Alignment Description(CP)]>
<[Alignment Length(Uft|P3|RN|AP|Sn|OF)]>
<[Alignment Start Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Alignment End Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>

Label styles for minor stations, geometry points, design speeds, and station equations can also use the following property fields:

<[Offset From Major Station(Uft|P3|RN|AP|Sn|OF)]>	Minor stations
<[Geometry Point Text(CP)]>	Geometry points
<[Geometry Point Entity Before Data(CP)]>	Geometry points

<[Geometry Point Entity After Data(CP)]>	Geometry points
<[Design Speed Before(P3|RN|AP|Sn|OF)]>	Design speeds
<[Station Ahead(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>	Station equations
<[Station Back(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>	Station equations
<[Increase/Decrease(CP)]>	Station equations

Label styles are described in detail in the chapter 2 section Label Styles (page 27).

Sample Programs

AlignmentSample

<installation-directory>\Sample\Civil 3D API\DotNet\VB.NET\AlignmentSample

Some of the sample code from this chapter can be found in context in the AlignmentSample project. This sample shows how to create an alignment, add entities to an alignment, create an alignment label style set, get a site, and create an alignment style.

Sprial2 Demo

<installation-directory>\Sample\Civil 3D API\DotNet\CSharp\Spiral2 Demo

Demonstrates how to get simple and complex alignment information.

Profiles

This chapter describes the process for creating and using profiles with the AutoCAD Civil 3D .NET API. For information about using Profiles with the COM API, see Profiles (page 297)

Profiles

Profiles are the vertical analogue to alignments. Together, an alignment and a profile represent a 3D path.

Creating a Profile From a Surface

A profile is an object consisting of elevations along an alignment. Each alignment contains a collection of profiles which you can access by the `Alignment.GetProfileIds()` method. The `Profile.CreateFromSurface()` method creates a new profile and derives its elevation information from the specified surface along the path of the alignment.

```
// Illustrates creating a new profile from a surface
[CommandMethod("ProfileFromSurface")]
public void ProfileFromSurface()
{
    doc = CivilApplication.ActiveDocument;
    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())
    {
        // Ask the user to select an alignment
        PromptEntityOptions opt = new
PromptEntityOptions("\nSelect an Alignment");
        opt.SetRejectMessage("\nObject must be an
alignment.\n");
        opt.AddAllowedClass(typeof(Alignment), true);
        ObjectId alignID = ed.GetEntity(opt).ObjectId;
        // get layer id from the alignment
        Alignment oAlignment = ts.GetObject(alignID,
OpenMode.ForRead) as Alignment;
        ObjectId layerId = oAlignment.LayerId;
        // get first surface in the document
        ObjectId surfaceId = doc.GetSurfaceIds()[0];
        // get first style in the document
        ObjectId styleId = doc.Styles.ProfileStyles[0];
        // get the first label set style in the document
        ObjectId labelSetId =
doc.Styles.LabelSetStyles.ProfileLabelSetStyles[0];
        try
        {
            ObjectId profileId =
Profile.CreateFromSurface("My Profile", alignID, surfaceId,
 layerId, styleId, labelSetId);
            ts.Commit();
        }
        catch (Autodesk.AutoCAD.Runtime.Exception e)
        {
            ed.WriteMessage(e.Message);
        }

    }
}
```

Creating a Profile Using Entities

The various `Profile.CreateByLayout()` overloaded methods create a new profile with no elevation information. The vertical shape of a profile can then be specified using entities. Entities are geometric elements - tangents or symmetric parabolas. The collection of all entities that make up a profile are contained in the `Profile.Entities` collection. The `ProfileEntityCollection` class also contains all the methods for creating new entities.

This sample creates a new profile along the alignment “oAlignment” and then adds three entities to define the profile shape. Two straight entities are added at each end and a symmetric parabola is added in the center to join them and represent the sag of a valley.

```
// Illustrates creating a new profile without elevation
data, then adding the elevation
// via the entities collection

[CommandMethod("CreateProfileNoSurface")]
public void CreateProfileNoSurface()
{
    doc = CivilApplication.ActiveDocument;

    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())
    {
        // Ask the user to select an alignment
        PromptEntityOptions opt = new
PromptEntityOptions("\nSelect an Alignment");
        opt.SetRejectMessage("\nObject must be an
alignment.\n");
        opt.AddAllowedClass(typeof(Alignment), false);
        ObjectId alignID = ed.GetEntity(opt).ObjectId;

        Alignment oAlignment = ts.GetObject(alignID,
OpenMode.ForRead) as Alignment;

        // use the same layer as the alignment
        ObjectId layerId = oAlignment.LayerId;
        // get the standard style and label set
        // these calls will fail on templates without a
style named "Standard"
        ObjectId styleId =
doc.Styles.ProfileStyles["Standard"];
        ObjectId labelSetId =
doc.Styles.LabelSetStyles.ProfileLabelSetStyles["Standard"];

        ObjectId oProfileId = Profile.CreateByLayout("My
Profile", alignID, layerId, styleId, labelSetId);

        // Now add the entities that define the profile.

        Profile oProfile = ts.GetObject(oProfileId,
OpenMode.ForRead) as Profile;

        Point3d startPoint = new
Point3d(oAlignment.StartingStation, -40, 0);
        Point3d endPoint = new Point3d(758.2, -70, 0);
        ProfileTangent oTangent1 =
oProfile.Entities.AddFixedTangent(startPoint, endPoint);

        startPoint = new Point3d(1508.2, -60.0, 0);
        endPoint = new Point3d(oAlignment.EndingStation,
-4.0, 0);
        ProfileTangent oTangent2
=oProfile.Entities.AddFixedTangent(startPoint, endPoint);
```

```
        oProfile.Entities.AddFreeSymmetricParabolaByLength(oTangent1.EntityId,
 oTangent2.EntityId, VerticalCurveType.Sag, 900.1, true);

            ts.Commit();
        }

    }
```

Editing Points of Vertical Intersection

The point where two adjacent tangents would cross (whether they actually cross or not) is called the "point of vertical intersection", or "PVI." This location can be useful for editing the geometry of a profile because this one point controls the slopes of both tangents and any curve connecting them. The collection of all PVIs in a profile is contained in the `Profile.PVIs` property. This collection lets you access, add, and remove PVIs from a profile, which can change the position and number of entities that make up the profile. Individual PVIs (type `ProfilePVI`) do not have a name or id, but are instead identified by a particular station and elevation. The collection methods `ProfilePVICollection.GetPVIAt` and `ProfilePVICollection.RemoveAt` either access or delete the PVI closest to the station and elevation parameters so you do not need the exact location of the PVI you want to modify.

This sample identifies the PVI closest to a specified point. It then adds a new PVI to the profile created in the Creating a Profile Using Entities (page 299) topic and adjusts its elevation.

```
 [CommandMethod("EditPVI")]
public void EditPVI()
{
    doc = CivilApplication.ActiveDocument;

    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())
    {
       // get first profile of first alignment in document
        ObjectId alignID = doc.GetAlignmentIds()[0];
        Alignment oAlignment = ts.GetObject(alignID,
OpenMode.ForRead) as Alignment;
        Profile oProfile =
ts.GetObject(oAlignment.GetProfileIds()[0],
OpenMode.ForRead) as Profile;

        // check to make sure we have a profile:
        if (oProfile == null)
        {
            ed.WriteMessage("Must have at least one
alignment with one profile");
            return;
        }
        //  Find the PVI close to station 1000 elevation
-70.
        ProfilePVI oProfilePVI =
oProfile.PVIs.GetPVIAt(1000, -70);
        ed.WriteMessage("PVI closest to station 1000 is at
 station: {0}", oProfilePVI.Station);
        // Add another PVI and slightly adjust its
elevation.
        oProfilePVI = oProfile.PVIs.AddPVI(607.4, -64.3);
        oProfilePVI.Elevation -= 2.0;

        ts.Commit();
    }
}
```

Creating a Profile Style

The profile style, an object of type `ProfileStyle`, defines the visual display of profiles. The collection of all such styles in a document are stored in the `CivilDocument.Styles.ProfileStyles` collection. The style contains properties of type `DisplayStyle` which govern the display of arrows showing alignment direction and of the lines, line extensions, curves, parabolic curve extensions, symmetrical parabolas and asymmetrical parabolas that make up a profile. The properties of a new profile style are defined by the document's ambient settings.

```
// Illustrates creating a new profile style
[CommandMethod("CreateProfileStyle")]
public void CreateProfileStyle()
{
    doc = CivilApplication.ActiveDocument;
    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())
    {
        ObjectId profileStyleId =
doc.Styles.ProfileStyles.Add("New Profile Style");
        ProfileStyle oProfileStyle =
ts.GetObject(profileStyleId, OpenMode.ForRead) as
ProfileStyle;
        oProfileStyle.GetDisplayStyleProfile(ProfileDisplayStyleProfileType.Arrow).Visible
 = true;
        // set to yellow:
        oProfileStyle.GetDisplayStyleProfile(ProfileDisplayStyleProfileType.Line).Color
 = Color.FromColorIndex(ColorMethod.ByAci, 50);
        oProfileStyle.GetDisplayStyleProfile(ProfileDisplayStyleProfileType.Line).Visible
 = true;
        // grey
        oProfileStyle.GetDisplayStyleProfile(ProfileDisplayStyleProfileType.LineExtension).Color
 = Color.FromColorIndex(ColorMethod.ByAci, 251);
        oProfileStyle.GetDisplayStyleProfile(ProfileDisplayStyleProfileType.LineExtension).Visible
 = true;
        // green
        oProfileStyle.GetDisplayStyleProfile(ProfileDisplayStyleProfileType.Curve).Color
 = Color.FromColorIndex(ColorMethod.ByAci, 80);
        oProfileStyle.GetDisplayStyleProfile(ProfileDisplayStyleProfileType.Curve).Visible
 = true;
        // grey
        oProfileStyle.GetDisplayStyleProfile(ProfileDisplayStyleProfileType.ParabolicCurveExtension).Color
 = Color.FromColorIndex(ColorMethod.ByAci, 251);
        oProfileStyle.GetDisplayStyleProfile(ProfileDisplayStyleProfileType.ParabolicCurveExtension).Visible
 = true;
        // green
        oProfileStyle.GetDisplayStyleProfile(ProfileDisplayStyleProfileType.SymmetricalParabola).Color
 = Color.FromColorIndex(ColorMethod.ByAci, 81);
        oProfileStyle.GetDisplayStyleProfile(ProfileDisplayStyleProfileType.SymmetricalParabola).Visible
 = true;
        // green
        oProfileStyle.GetDisplayStyleProfile(ProfileDisplayStyleProfileType.AsymmetricalParabola).Color
 = Color.FromColorIndex(ColorMethod.ByAci, 83);
        oProfileStyle.GetDisplayStyleProfile(ProfileDisplayStyleProfileType.AsymmetricalParabola).Visible
 = true;
```

```
        // properties for 3D should also be set
    }
}
```

Profile Views

This section describes the creation and display of profile views. A profile view is a graph displaying the elevation of a profile along the length of the related alignment.

Creating a Profile View

The `ProfileView` class has two versions of the `ProfileView::Create()` method for adding new ProfileView objects to a drawing. Each method overload takes a reference to the `CivilDrawing` object, the name of the new profile view, and a `Point3d` location in the drawing where the profile view is inserted. They also both take a band set style and alignment; one version takes these arguments as ObjectIds, while the other takes them as strings.

This example demonstrates creating a new ProfileView:

```
[CommandMethod("CreateProfileView")]
public void CreateProfileView()
{
    doc = CivilApplication.ActiveDocument;
    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())
    {
        // Ask the user to select an alignment
        PromptEntityOptions opt = new
PromptEntityOptions("\nSelect an Alignment");
        opt.SetRejectMessage("\nObject must be an
alignment.\n");
        opt.AddAllowedClass(typeof(Alignment), false);
        ObjectId alignID = ed.GetEntity(opt).ObjectId;
        // Create insertion point:
        Point3d ptInsert = new Point3d(100, 100, 0);
        // Get profile view band set style ID:
        ObjectId pfrVBSStyleId =
doc.Styles.ProfileViewBandSetStyles["Standard"];
        // If this doesn't exist, get the first style in
the collection
        if (pfrVBSStyleId == null) pfrVBSStyleId =
doc.Styles.ProfileViewBandSetStyles[0];
        ObjectId ProfileViewId = ProfileView.Create(doc,
"New Profile View", pfrVBSStyleId, alignID, ptInsert);
        ts.Commit();
    }
}
```

Creating Profile View Styles

The profile view style, an object of type `ProfileViewStyle`, governs all aspects of how the graph axes, text, and titles are drawn. Within `ProfileViewStyle` are objects dealing with the top, bottom, left, and right axes; lines at geometric locations within profiles; and with the graph as a whole. All profile view styles in the document are stored in the `CivilDocument.ProfileViewStyles` collection. New styles are created using the collection's `Add` method with the name of the new style.

```
ObjectId profileViewStyleId =
doc.Styles.ProfileViewStyles.Add("New Profile View Style");
ProfileViewStyle oProfileViewStyle =
ts.GetObject(profileViewStyleId, OpenMode.ForRead) as
ProfileViewStyle;
```

Setting Profile View Styles

The profile view style object consists of separate objects for each of the four axes, one object for the graph overall, and an `DisplayStyle` object for grid lines displayed at horizontal geometry points (accessed with the `GetDisplayStylePlan()` method). The axis styles and graph style also contain subobjects for specifying the style of tick marks and titles.

Setting the Axis Style

All axis styles are based on the `AxisStyle` class. The axis style object controls the display style of the axis itself, tick marks and text placed along the axis, and a text annotation describing the axis's purpose. The annotation text, location, and size is set through the `AxisStyle.TitleStyle` property, an object of type `AxisTitleStyle`. The annotation text can use any of the following property fields:

Valid property fields for AxisTitleStyle.Text
<[Profile View Minimum Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[Profile View Maximum Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[Profile View Start Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Profile View End Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>

Axis Tick Marks

Within each axis style are properties for specifying the tick marks placed along the axis. Both major tick marks and minor tick marks are represented by objects of type `AxisTickStyle`. The `AxisTickStyle` class manages the location, size, and visual style of tick marks through its `Interval`, `Size` and other properties. Note that while most style properties use drawing units, the `Interval` property uses the actual ground units of the surface. The `AxisTickStyle` object also determines the text that is displayed at each tick, including the following property fields:

Valid property fields for TickStyle.Text	Axis
<[Station Value(Uft|FS|P0|RN|AP|Sn|TP|B2|EN|W0|OF)]>	horizontal
<[Raw Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>	horizontal
<[Graph View Abscissa Value(Uft|P4|RN|AP|Sn|OF)]>	horizontal
<[Profile View Point Elevation(Uft|P1|RN|AP|Sn|OF)]>	vertical

Valid property fields for TickStyle.Text	Axis
<[Graph View Ordinate Value(Uft|P3|RN|AP|Sn|OF)]>	vertical

Setting the Graph Style

The graph is managed by objects of type `GraphStyle` and `GridStyle`. These objects can be used to change the scale, title, and grid of the graph.

The grid is controlled by the `ProfileViewStyle.GridStyle` property, an object of type `GridStyle`. The grid style sets the amount of empty space above and below the extents of the section through the `GridStyle.GridPaddingAbove` and `GridStyle.GridPaddingBottom` properties. The grid style also manages the line styles of major and minor vertical and horizontal gridlines with the `DisplayStyle.PlotStyle` property accessed by the `GetDisplayStylePlan()` method.

Graph Title

The title of the graph is controlled by the `GraphStyle.TitleStyle` property, an object of type `GraphTitleStyle`. The title style object can adjust the position, style, and border of the title. The text of the title can include any of the following property fields:

Valid property fields for GraphTitleStyle.Text
<[Graph View Name(CP)]>
<[Parent Alignment(CP)]>
<[Drawing Scale(P4|RN|AP|OF)]>
<[Graph View Vertical Scale(P4|RN|AP|OF)]>
<[Graph View Vertical Exaggeration(P4|RN|AP|OF)]>
<[Profile View Start Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>

Valid property fields for GraphTitleStyle.Text
<[Profile View End Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Profile View Minimum Elevation(Uft|P2|RN|AP|Sn|OF)]>
<[Profile View Maximum Elevation(Uft|P3|RN|AP|Sn|OF)]>

Working With Hatch Areas

Hatch areas are a feature of profile views that apply a style to areas of cut and fill to highlight them. In addition to cut and fill, a hatch area can highlight areas of intersection between any two defined profiles.

The hatching feature for `ProfileView` objects is exposed by the `HatchAreas` property. This is a collection of all `ProfileHatchArea` objects defined for the `ProfileView`, which can be used to access or add additional hatch areas.

Each `ProfileHatchArea` has a set of criteria (`ProfileCriteria` objects) that specify the profile that defines the upper or lower boundary for the hatch area. The criteria also references a `ShapeStyle` object that defines how the hatch area is styled in the profile view.

This code sample illustrates how to access the hatch areas for a profile view, and prints out some information about each `ProfileHatchArea` object's criteria:

```
[CommandMethod("ProfileHatching")]
public void ProfileHatching () {
    doc = CivilApplication.ActiveDocument;
    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;
    using ( Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction()
 ) {
        // Ask the user to select a profile view
        PromptEntityOptions opt = new
PromptEntityOptions("\nSelect a profile view");
        opt.SetRejectMessage("\nObject must be a profile
view.\n");
        opt.AddAllowedClass(typeof(ProfileView), false);
```

```
       ObjectId profileViewID = ed.GetEntity(opt).ObjectId;
        ProfileView oProfileView =
ts.GetObject(profileViewID, OpenMode.ForRead) as
ProfileView;
        ed.WriteMessage("\nHatch areas defined in this
profile view: \n");
        foreach ( ProfileHatchArea oProfileHatchArea in
oProfileView.HatchAreas ) {
            ed.WriteMessage(" Hatch area: " +
oProfileHatchArea.Name + " shape style: " +
oProfileHatchArea.ShapeStyleName + "\n");
            foreach ( ProfileCriteria oProfileCriteria in
 oProfileHatchArea.Criteria ) {
               ed.WriteMessage(string.Format("   Criteria:
 type: {0} profile: {1}\n",
oProfileCriteria.BoundaryType.ToString(),
oProfileCriteria.ProfileName) );
            }
        }
    }
}
```

Profile View Style Example

This example takes an existing profile view style and modifies its top axis and title:

```
[CommandMethod("ModProfileViewStyle")]
public void ModProfileViewStyle()
{
    doc = CivilApplication.ActiveDocument;

    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.Database.TransactionManager.StartTransaction())
    {
       // Get the first style in the document's collection
 of styles

        ObjectId profileViewStyleId =
doc.Styles.ProfileViewStyles[0];
        ProfileViewStyle oProfileViewStyle =
ts.GetObject(profileViewStyleId, OpenMode.ForRead) as
ProfileViewStyle;

        // Adjust the top axis.  Put station information
here, with the title
        // at the far left.
     oProfileViewStyle.GetDisplayStylePlan(ProfileViewDisplayStyleType.TopAxis).Visible
 = true;
       oProfileViewStyle.TopAxis.MajorTickStyle.LabelText
 = "<[Station Value(Um|FD|P1)]> m";
        oProfileViewStyle.TopAxis.MajorTickStyle.Interval
 = 164.041995;
        oProfileViewStyle.TopAxis.TitleStyle.OffsetX =
0.13;
       oProfileViewStyle.TopAxis.TitleStyle.OffsetY = 0.0;
        oProfileViewStyle.TopAxis.TitleStyle.Text =
"Meters";
        oProfileViewStyle.TopAxis.TitleStyle.Location =
Autodesk.Civil.DatabaseServices.Styles.AxisTitleLocationType.TopOrLeft;

        // Adjust the title to show the alignment name
        oProfileViewStyle.GraphStyle.TitleStyle.Text =
"Profile View of:<[Parent Alignment(CP)]>";

        ts.Commit();
    }
}
```

Sample Programs

Profile Sample

This sample is located in *<install directory>\Sample\Civil 3D API\DotNet\VB.NET\ProfileSample\*. It illustrates:

- How to create a profile
- Creating profile styles and profile views

Pipe Networks

This chapter describes working with pipe networks with the AutoCAD Civil 3D .NET API.

Base Objects

This section explains how to get the base objects required for using the pipe network API classes.

Accessing Pipe Network-Specific Base Objects

Applications that access pipe networks do so through the `CivilDocument` object. This is different from the COM API, in which pipe network functionality is accessed through the separate `AeccPipeDocument` instead of `AeccDocument`. In the .NET API, the `CivilDocument` object contains collections of pipe network-related items, such as pipe networks, pipe styles, and interference checks.

Pipe-Specific Ambient Settings

Ambient settings allow you to get and set the units and default property settings of pipe network objects as well as access the catalog of all pipe and structure parts held in the document. Ambient settings for a pipe document are obtained from the `CivilDocument.Settings.GetSettings()` method, which returns an object inherited from `SettingsAmbient`.

Among the classes that inherit from `SettingsAmbient` are `SettingsPipe`, `SettingsPipeNetwork`, and `SettingsStructure`. Each of these has properties that describe the default units of measurement for interference, pipe, and structure objects. The `PipeSettingsRoot.PipeNetworkSettings` property contains the name of the default styles for pipe and structure objects as well as the default label placement, units, and naming conventions for pipe networks as a whole.

```
public void ShowPipeRules()
{
    CivilDocument doc = CivilApplication.ActiveDocument;
    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;
    SettingsPipeNetwork oSettingsPipeNetwork =
doc.Settings.GetSettings<SettingsPipeNetwork>() as
SettingsPipeNetwork;
    ed.WriteMessage("Using pipe rules: {0}\n",
oSettingsPipeNetwork.Rules.Pipe.Value);
    //  Set the default units used for pipes in this
document.
    oSettingsPipeNetwork.Angle.Unit.Value =
Autodesk.Civil.AngleUnitType.Radian;
    oSettingsPipeNetwork.Coordinate.Unit.Value =
Autodesk.Civil.LinearUnitType.Foot;
    oSettingsPipeNetwork.Distance.Unit.Value =
Autodesk.Civil.LinearUnitType.Foot;
}
```

Listing and Adding Dynamic Part Properties

Each type of pipe and structure has many unique attributes (such as size, geometry, design, and composition) that cannot be stored in the standard pipe and structure properties. To give each part appropriate attributes, pipe and structure objects have sets of dynamic properties. A single property is represented by an `PartDataField` object. Data fields are held in collections of type `PartDataRecord`. You can reach these collections through the `PartData` property of the `Part` class, from which `Pipe` and `Structure` objects inherit. Each data field contains an internal variable name, a text description of the value, a global context used to identify the field, data type, and the data value itself, as well as other properties.

This sample enumerates all the data fields contained in a pipe object "oPipe" and displays information from each field.

```
// Get PartDataRecord for first pipe in the network
ObjectId pipeId =  oNetwork.GetPipeIds()[0];
Pipe oPipe = ts.GetObject(pipeId, OpenMode.ForRead) as
Pipe;
PartDataField[] oDataFields =
oPipe.PartData.GetAllDataFields();
ed.WriteMessage("Additional info for pipe: {0}\n",
oPipe.Name);
foreach (PartDataField oPartDataField in oDataFields)
```

```
{
    ed.WriteMessage("Name: {0}, Description: {1},  DataType:
 {2}, Value: {3}\n",
         oPartDataField.Name,
         oPartDataField.Description,
         oPartDataField.DataType,
         oPartDataField.Value);
}
```

Dynamic properties created with the NetworkCatalogDef class are not yet supported by the .NET API.

Retrieving the Parts List

`CivilDocument.Styles.PartsListSet` contains a read-only collection of all the lists of part types available in the document. Each list is an object of type `PartList`, a read-only collection of PartsList ObjectIds. A part family represents a broad category of parts, and is identified by a GUID (Globally Unique Identification) value. A part family can only contain parts from one domain - either pipes or structures but not both. Part families contain a read-only collection of part filters (`PartSizeFilter`), which are the particular sizes of parts. A part filter is defined by its `PartSizeFilter.PartDataRecord` property, a collection of fields describing various aspects of the part.

This sample prints the complete listing of all parts in a document.

```
[CommandMethod("PrintParts")]
public void PrintParts()
{
    CivilDocument doc = CivilApplication.ActiveDocument;
    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.
        Database.TransactionManager.StartTransaction())
    {
        // SettingsPipeNetwork oSettingsPipeNetwork =
doc.Settings.GetSettings<SettingsPipeNetwork>() as
SettingsPipeNetwork;
        PartsListCollection oPartListCollection =
doc.Styles.PartsListSet;
       ed.WriteMessage("Number of parts lists in document:
```

```
 {0}\n", oPartListCollection.Count);
        foreach (ObjectId objId in oPartListCollection)
        {
            PartsList oPartsList = ts.GetObject(objId,
OpenMode.ForWrite) as PartsList;
            ed.WriteMessage("PARTS LIST:
{0}\n----------------\n", oPartsList.Name);
            // From the part list, looking at only those
part families
            // that are pipes, print all the individual
parts, plus
            // some information about each part.
            ObjectIdCollection pipeFamilyCollection =
oPartsList.GetPartFamilyIdsByDomain(DomainType.Pipe);
            ed.WriteMessage("  Pipes\n  =====\n");
            foreach (ObjectId objIdPfa in
pipeFamilyCollection)
            {
                PartFamily oPartFamily =
ts.GetObject(objIdPfa, OpenMode.ForWrite) as PartFamily;
                if (oPartFamily.Domain == DomainType.Pipe)
                {
                    ed.WriteMessage("  Family: {0}\n",
oPartFamily.Name);
                    SizeFilterRecord oSizeFilterRecord =
oPartFamily.PartSizeFilter;
                    SizeFilterField SweptShape =
oSizeFilterRecord.GetParamByContextAndIndex(PartContextType.SweptShape,
 0);
                    SizeFilterField MinCurveRadius =
oSizeFilterRecord.GetParamByContextAndIndex(PartContextType.MinCurveRadius,
 0);
                    //SizeFilterField
StructPipeWallThickness;
                    SizeFilterField FlowAnalysisManning =

oSizeFilterRecord.GetParamByContextAndIndex(PartContextType.FlowAnalysisManning,
 0);
                    SizeFilterField m_Material =
oSizeFilterRecord.GetParamByContextAndIndex(PartContextType.Material,
 0);
                    // SizeFilterField PipeInnerDiameter
=
oSizeFilterRecord.GetParamByContextAndIndex(PartContextType.PipeInnerDiameter,
 0);
                    ed.WriteMessage("  {0}: {1}, {2}: {3},
 {4}: {5} {6}: {7}\n",
                        SweptShape.Description,
SweptShape.Value,
                        MinCurveRadius.Description,
MinCurveRadius.Value,
                        FlowAnalysisManning.Description,
FlowAnalysisManning.Value,
                        m_Material.Description,
m_Material.Value
                        );
                }
            }
            // From the part list, looking at only those
part families
          // that are structures, print all the individual
 parts.
            ed.WriteMessage("  Structures\n  =====\n");
            foreach (ObjectId objIdPfa in
pipeFamilyCollection)
            {
                PartFamily oPartFamily =
ts.GetObject(objIdPfa, OpenMode.ForWrite) as PartFamily;
                if (oPartFamily.Domain ==
DomainType.Structure)
                {
                    ed.WriteMessage("  Family: {0}\n",
oPartFamily.Name);
                }
            }
        }
    }
}
```

Creating a Pipe Network

A pipe network is a set of interconnected or related parts. The collection of all pipe networks is returned by the `CivilDocument.GetPipeNetworkIds()` method. A pipe network, an object of type `Network`, contains the collection of pipes and the collection of structures which make up the network. `Network` also contains the method `FindShortestNetworkPath()` for determining the path between two network parts.

The `Network.ReferenceSurfaceId` is used primarily for Pipe Rules. For example, you can have a rule that places the structure rim at a specified elevation from the surface.

```
Public Function CreatePipeNetwork() As Boolean
    Dim trans As Transaction = tm.StartTransaction()
    Dim oPipeNetworkIds As ObjectIdCollection
    Dim oNetworkId As ObjectId
    Dim oNetwork As Network
    oNetworkId = Network.Create(g_oDocument, NETWORK_NAME)
    ' get the network
    Try
        oNetwork = trans.GetObject(oNetworkId,
OpenMode.ForWrite)
    Catch
        CreatePipeNetwork = False
        Exit Function
    End Try
    '
    'Add pipe and Structure
    ' Get the Networks collections
    oPipeNetworkIds = g_oDocument.GetPipeNetworkIds()
    If (oPipeNetworkIds Is Nothing) Then
        MsgBox("There is no PipeNetwork Collection." +
Convert.ToChar(10))
        ed.WriteMessage("There is no PipeNetwork
Collection." + Convert.ToChar(10))
        CreatePipeNetwork = False
        Exit Function
    End If
    Dim oPartsListId As ObjectId =
g_oDocument.Styles.PartsListSet(PARTS_LIST_NAME) 'Standard
 PartsList
    Dim oPartsList As PartsList =
trans.GetObject(oPartsListId, OpenMode.ForWrite)
    Dim oidPipe As ObjectId = oPartsList("Concrete Pipe
SI")
    Dim opfPipe As PartFamily = trans.GetObject(oidPipe,
OpenMode.ForWrite)
    Dim psizePipe As ObjectId = opfPipe(0)
    Dim line As LineSegment3d = New LineSegment3d(New
Point3d(30, 9, 0), New Point3d(33, 7, 0))
    Dim oidNewPipe As ObjectId = ObjectId.Null
    oNetwork.AddLinePipe(oidPipe, psizePipe, line,
oidNewPipe, True)
    Dim oidStructure As ObjectId = oPartsList("CMP
Rectangular End Section SI")
    Dim opfStructure As PartFamily =
trans.GetObject(oidStructure, OpenMode.ForWrite)
    Dim psizeStructure As ObjectId = opfStructure(0)
    Dim startPoint As Point3d = New Point3d(30, 9, 0)
    Dim endPoint As Point3d = New Point3d(33, 7, 0)
    Dim oidNewStructure As ObjectId = ObjectId.Null
    oNetwork.AddStructure(oidStructure, psizeStructure,
startPoint, 0, oidNewStructure, True)
    oNetwork.AddStructure(oidStructure, psizeStructure,
endPoint, 0, oidNewStructure, True)
    ed.WriteMessage("PipeNetwork created" +
Convert.ToChar(10))
    trans.Commit()
    CreatePipeNetwork = True
End Function ' CreatePipeNetwork
```

Pipes

This section explains the creation and use of pipes. Pipes represent the conduits within a pipe network.

Creating Pipes

Pipe objects represent the conduits of the pipe network. Pipes are created using the pipe network's methods for creating either straight or curved pipes, `AddLinePipe()` and `AddCurvePipe()`. Both methods require you to specify a particular part family (using the ObjectId of a family) and a particular part size filter object as well as the geometry of the pipe.

This sample creates a straight pipe between two hard-coded points using the first pipe family and pipe size filter it can find in the part list:

```
[CommandMethod("AddPipe")]
```

```
public void AddPipe()
{
    CivilDocument doc = CivilApplication.ActiveDocument;
    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.
        Database.TransactionManager.StartTransaction())
    {
        ObjectIdCollection oIdCollection =
doc.GetPipeNetworkIds();
        // Get the first network in the document
        ObjectId objId = oIdCollection[0];
        Network oNetwork = ts.GetObject(objId,
OpenMode.ForWrite) as Network;
        ed.WriteMessage("Pipe Network: {0}\n",
oNetwork.Name);
        // Go through the list of part types and select
the first pipe found
        ObjectId pid = oNetwork.PartsListId;
       PartsList pl = ts.GetObject(pid, OpenMode.ForWrite)
 as PartsList;
        ObjectId oid = pl["Concrete Pipe"];
        PartFamily pfa = ts.GetObject(oid,
OpenMode.ForWrite) as PartFamily;
        ObjectId psize = pfa[0];
        LineSegment3d line = new LineSegment3d(new
Point3d(30, 9, 0), new Point3d(33, 7, 0));
        ObjectIdCollection col = oNetwork.GetPipeIds();
        ObjectId oidNewPipe = ObjectId.Null;
        oNetwork.AddLinePipe(oid, psize, line, ref
oidNewPipe, false);
        Pipe oNewPipe = ts.GetObject(oidNewPipe,
OpenMode.ForRead) as Pipe;
        ed.WriteMessage("Pipe created: {0}\n",
oNewPipe.DisplayName);
        ts.Commit();
    }

}
```

Using Pipes

To make a new pipe a meaningful part of a pipe network, it must be connected to structures or other pipes using the `Pipe.ConnectToStructure()` or `Pipe.ConnectToPipe()` methods, or structures must be connected to it using the `Structure.ConnectToPipe()` method. Connecting pipes together directly creates a new virtual `Structure` object to serve as the joint. If a pipe end is connected to a structure, it must be disconnected before attempting to connect it to a different structure. After a pipe has been connected to a network, you can determine the structures at either end by using the `StartStructureId` and `EndStructureId` properties. There are methods and properties for setting and determining the flow direction, getting all types of physical measurements, and for accessing collections of user-defined properties for custom descriptions of the pipe.

Creating Pipe Styles

A pipe style controls the visual appearance of pipes in a document. All pipe style objects in a document are stored in the `CivilDocument.PipeStyles` collection. Pipe styles have four display methods and three hatch methods for controlling general appearance attributes and three properties for controlling display attributes that are specific to pipes. The methods `GetDisplayStyleProfile|Section|Plan()`, and `GetHatchStyleProfile()` all take a parameter describing the feature being modified, and return a reference to the `DisplayStyle` or `HatchDisplayStyle` object controlling common display attributes, such as line styles and color. The methods `GetDisplayStyleModel()`, `GetHatchStylePlan()`, and `GetHatchStyleSection()` do not take a component parameter.

The properties `PlanOption` and `ProfileOption` set the size of the inner wall, outer wall, and end lines according to either the physical properties of the pipe, custom sizes using drawing units, or a certain percentage of its previous drawing size. The `HatchOption` property sets the area of the pipe covered by any hatching used. A pipe object is given a style by assigning the `Pipe.Style` property to a `PipeStyle` object.

This sample attempts to create a new pipe style object and set some of its properties. If a style already exists with the same name, it sets the properties on the existing style:

```
Public Function CreatePipeStyle(ByVal sStyleName As String)
```

```
 As PipeStyle
    Dim oPipeStyleId As ObjectId
    Dim oPipeStyle As PipeStyle
    Dim trans As Transaction = tm.StartTransaction()
    Try
        oPipeStyleId =
g_oDocument.Styles.PipeStyles.Add(sStyleName)
    Catch
    End Try
    If (oPipeStyleId = ObjectId.Null) Then
        Try
            oPipeStyleId =
g_oDocument.Styles.PipeStyles.Item(sStyleName)
        Catch
        End Try
        If (oPipeStyleId = ObjectId.Null) Then
            MsgBox("Could not create or use a pipe style
with the name:" & sStyleName)
            CreatePipeStyle = Nothing
            Exit Function
        End If
    End If
    oPipeStyle = trans.GetObject(oPipeStyleId,
OpenMode.ForWrite)
    ' Set the display size of the pipes in plan view.  We
 will
    ' use absolute drawing units for the inside, outside,
 and
    ' ends of each pipe.
    ' enter a value greater than or equal to 0.000mm and
less than or equal to 1000.000mm
    oPipeStyle.PlanOption.InnerDiameter = 0.0021
    oPipeStyle.PlanOption.OuterDiameter = 0.0024
    ' Indicate that we will use our own measurements for
the inside
    ' and outside of the pipe, and not base drawing on the
 actual
    ' type of pipe.
    oPipeStyle.PlanOption.WallSizeType =
PipeWallSizeType.UserDefinedWallSize
    ' Inidcate what kind of custom sizing to use.
    oPipeStyle.PlanOption.WallSizeOptions =
PipeUserDefinedType.UseDrawingScale
```

```
    oPipeStyle.PlanOption.EndLineSize = 0.0021
    ' Indicate that we will use our own measurements for
the end
    'line of the pipe, and not base drawing on the actual
 type
    ' of pipe.
    oPipeStyle.PlanOption.EndSizeType =
PipeEndSizeType.UserDefinedEndSize
    ' Inidcate what kind of custom sizing to use.
    oPipeStyle.PlanOption.EndSizeOptions =
PipeUserDefinedType.UseDrawingScale
    '
    ' Modify the colors of pipes using this style, as shown

    'in plan view.
   oPipeStyle.GetDisplayStylePlan(PipeDisplayStylePlanType.OutsideWalls).Color
 = Color.FromRgb(255, 191, 0)  ' orange, ColorIndex = 40
   oPipeStyle.GetDisplayStylePlan(PipeDisplayStylePlanType.InsideWalls).Color
 = Color.FromRgb(191, 0, 255) ' violet, ColorIndex = 200
   oPipeStyle.GetDisplayStylePlan(PipeDisplayStylePlanType.EndLine).Color
 = Color.FromRgb(191, 0, 255) ' violet, ColorIndex = 200
    '
    ' Set the hatch style for pipes using this style, as
shown
    'in plan view.
    oPipeStyle.GetHatchStylePlan().Pattern = "DOTS"
    oPipeStyle.GetHatchStylePlan().HatchType =
Autodesk.Civil.DatabaseServices.Styles.HatchType.PreDefined
    oPipeStyle.GetHatchStylePlan().UseAngleOfObject = False
    oPipeStyle.GetHatchStylePlan().ScaleFactor = 9.0#
   oPipeStyle.GetDisplayStylePlan(PipeDisplayStylePlanType.Hatch).Color
 = Color.FromRgb(0, 255, 191) ' turquose, ColorIndex = 120
   oPipeStyle.GetDisplayStylePlan(PipeDisplayStylePlanType.Hatch).Visible
 = True
    oPipeStyle.PlanOption.HatchOptions =
PipeHatchType.HatchToInnerWalls
    trans.Commit()
    ed.WriteMessage("Create PipeStyle succeeded." +
Convert.ToChar(10))
    CreatePipeStyle = oPipeStyle
End Function ' CreatePipeStyle
```

Creating Pipe Label Styles

The collection of all pipe label styles in a document is found in the `CivilDocument.Styles.PipeLabelStyles` property, which is a `LabelStylesPipeRoot` object. This object lets you get and set cross section label styles, plan / profile view label styles, and default label styles for pipes.

NOTE

The label style of a particular pipe cannot be set using the .NET API.

Structures

This section describes the creation and use of structures. Structures are the connectors within a pipe network.

Creating Structures

Structures represent physical objects such as manholes, catch basins, and headwalls. Logically, structures are used as connections between pipes at pipe endpoints. In cases where two pipes connect directly, an `Structure` object not representing any physical object is still created to serve as the joint. Any number of pipes can connect with a structure. Structures are represented by objects of type `Structure`, which are created by using the `AddStructure()` method of `Network`.

See the code sample in Creating a Pipe Network (page 96) for an example of how to call this method.

Using Structures

To make the new structure a meaningful part of a pipe network, it must be connected to pipes in the network using the `Structure.ConnectToPipe()` method or pipes must be connected to it using the `Pipe.ConnectToStructure()` method. After a structure has been connected to a network, you can determine the pipes connected to it by using the `ConnectedPipe` property, which is a read-only collection of network parts. There are also methods and properties for setting and determining all types of physical measurements for the structure and for accessing collections of user-defined properties for custom descriptions of the structure.

Creating Structure Styles

A structure style controls the visual appearance of structures in a document. All structure style objects are stored in the `CivilDocument.Styles.StructureStyles` property. Structure styles have four methods for controlling general appearance attributes and three properties for controlling display attributes that are specific to structures. The methods `GetDisplayStylePlan|Profile|Section()` and `GetHatchStyleProfile()` all take a parameter describing the feature being modified and return a reference to the `DisplayStyle` or `HatchDisplayStyle` object controlling common display attributes such as line styles and color. The properties `PlanOption`, `ProfileOption`, `SectionOption`, and `ModelOption` set the display size of the structure and whether the structure is shown as a model of the physical object or only symbolically. A structure object is given a style by assigning the `Structure.StyleId` or `Structure.StyleName` property to a `StructureStyle` object.

This sample attempts to create a new structure style object and set some of its properties. If the style already exists, it changes the existing style:

```
Public Function CreateStructureStyle(ByVal sStyleName As
String) As StructureStyle
    Dim oStructureStyle As StructureStyle
    Dim oStructureStyleId As ObjectId
    Dim trans As Transaction = tm.StartTransaction()
    Try
        oStructureStyleId =
g_oDocument.Styles.StructureStyles.Add(sStyleName)
    Catch
    End Try
    If (oStructureStyleId = ObjectId.Null) Then
        Try
            oStructureStyleId =
g_oDocument.Styles.StructureStyles.Item(sStyleName)
        Catch
        End Try
        If (oStructureStyleId = ObjectId.Null) Then
            MsgBox("Could not create or use a structure
style with the name:" & sStyleName)
```

```
            CreateStructureStyle = Nothing
            Exit Function
        End If
    End If
    oStructureStyle = trans.GetObject(oStructureStyleId,
OpenMode.ForWrite)
   oStructureStyle.GetDisplayStylePlan(StructureDisplayStylePlanType.Structure).Color
 = Color.FromRgb(255, 191, 0) ' orange
   oStructureStyle.GetDisplayStylePlan(StructureDisplayStylePlanType.Structure).Visible
 = True
   oStructureStyle.PlanOption.MaskConnectedObjects = False
    oStructureStyle.PlanOption.SizeType =
StructureSizeOptionsType.UseDrawingScale
    oStructureStyle.PlanOption.Size = 0.0035
   oStructureStyle.GetDisplayStyleSection(StructureDisplayStylePlanType.Structure).Visible
 = False
   oStructureStyle.GetDisplayStyleSection(StructureDisplayStylePlanType.StructureHatch).Visible
 = False
   oStructureStyle.GetDisplayStylePlan(StructureDisplayStylePlanType.StructureHatch).Visible
 = False
   oStructureStyle.GetDisplayStyleProfile(StructureDisplayStylePlanType.Structure).Visible
 = False
   oStructureStyle.GetDisplayStyleProfile(StructureDisplayStylePlanType.StructureHatch).Visible
 = False
    trans.Commit()
    ed.WriteMessage("Create StructureStyle Successful." +
 Convert.ToChar(10))
    CreateStructureStyle = oStructureStyle
End Function ' CreateStructureStyle
```

Creating Structure Label Styles

The collection of all structure label styles in a document is found in the `CivilDocument.Styles.LabelStyles.StructureLabelStyles` property, which is a `LabelStylesStructureRoot` object. .

NOTE

The label style of a particular structure cannot be set using the .NET API.

Interference Checks

This section explains how to generate and examine an interference check. An interference check is used to determine when pipe network parts are either intersecting or are too close together.

Performing an Interference Check

This functionality is not yet supported by the .NET API.

Listing the Interferences

This functionality is not yet supported by the .NET API.

Interference Check Styles

Either a symbol or a model of the actual intersection region can be drawn at each interference location. The display of these intersections is controlled by an `InterferenceStyle` object. The collection of all interference style objects in the document are stored in the `CivilDocument.Styles.InterferenceStyles` collection.

There are three different styles of interference displays you can chose from. First, you can display a 3D model of the intersection region. This is done by setting the `ModelOptions` style property to `InterferenceModelType.TrueSolid`. The `GetDisplayStyleModel()` method returns an object of type `DisplayStyle` which controls the visible appearance of the model such as color and line types. Make sure the `DisplayStyle.Visible` property is set to `True`.

Another possibility is to draw a 3D sphere at the location of the intersection. This is done by setting the `ModelOptions` style property to `InterferenceModelType.Sphere`. If the `ModelSizeType` property is set to `InterferenceModelSizeType.SolidExtents`, then the sphere is automatically sized to just circumscribe the region of intersection (that is, it is the smallest sphere that still fits the model of the intersection region). You can set the size of the sphere by setting the `ModelSizeType` property to `InterferenceModelSizeType.UserSpecified`, setting the `ModelSizeOptions` property to use either absolute units or drawing units, and setting the corresponding `AbsoluteModelSize` or `DrawingScaleModelSize` property to the desired value. Again, the `DisplayStyle` object returned by `GetDisplayStyleModel()` controls the visual features such as color and line type.

The third option is to place a symbol at the location of intersection. Set the `GetDisplayStylePlan(InterferenceDisplayStyleType.Symbol).Visible` property to `True` to make symbols visible. The style property `MarkerStyle`, an object of type `MarkerStyle`, controls all aspects of how the symbol is drawn.

This sample creates a new interference style object that displays an X symbol with a superimposed circle at points of intersection:

```
public void InterfStyle()
{
    CivilDocument doc = CivilApplication.ActiveDocument;
    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;
    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.
       Database.TransactionManager.StartTransaction())
    {
        ObjectId intStyleId;
        intStyleId =
doc.Styles.InterferenceStyles.Add("Interference style 01");
        InterferenceStyle oIntStyle =
ts.GetObject(intStyleId, OpenMode.ForWrite) as
InterferenceStyle;
        // Draw a symbol of a violet X with circle with a
 specified
        // drawing size at the points of intersection.
     oIntStyle.GetDisplayStylePlan(InterferenceDisplayStyleType.Symbol).Visible
 = true;
        ObjectId markerStyleId = oIntStyle.MarkerStyle;
        MarkerStyle oMarkerStyle =
ts.GetObject(markerStyleId, OpenMode.ForWrite) as
MarkerStyle;
        oMarkerStyle.MarkerType =
MarkerDisplayType.UseCustomMarker;
        oMarkerStyle.CustomMarkerStyle =
CustomMarkerType.CustomMarkerX;
        oMarkerStyle.CustomMarkerSuperimposeStyle =
CustomMarkerSuperimposeType.Circle;
        oMarkerStyle.MarkerDisplayStylePlan.Color =
Color.FromColorIndex(ColorMethod.ByAci, 200);
       oMarkerStyle.MarkerDisplayStylePlan.Visible = true;
        oMarkerStyle.SizeType =
MarkerSizeType.AbsoluteUnits;
        oMarkerStyle.MarkerSize = 5.5;
        // Hide any model display at intersection points.
      oIntStyle.GetDisplayStyleModel(InterferenceDisplayStyleType.Solid).Visible
 = false;
        ts.Commit();
    }

}
```

Sample Program

PipeSample

<installation-directory>\Sample\Civil 3D API\DotNet\VB.NET\PipeSample

Some of the sample code from this chapter can be found in context in the PipeSample project. This sample creates a simple pipe network, creates pipe, structure and interference styles, creates a parts list, and prints a hierarchy of part types available in a document, separated into pipe and structure domains.

Corridors

This chapter covers creating and managing corridor objects using the AutoCAD Civil 3D .NET API.

Root Objects

Accessing Corridor-Specific Base Objects

The .NET API does not use separate root objects for getting roadway-related objects. Unlike the COM API, you only need to use the CivilApplication and CivilDocument classes to access corridor root objects.

Ambient Settings

Ambient settings allow you to get and set the unit and default property settings of roadway objects. Ambient settings for a corridor are accessed with the `SettingsCorridor` object returned by `CivilDocument.Settings.GetFeatureSettings()` method.

Corridor Ambient Settings

The corridor ambient settings object allows you to set the default name formats and default styles for corridor-related objects. The name templates allow you to set how new corridors, corridor surfaces, profiles from feature lines, or alignments from feature lines are named. Each format can use elements from the following property fields:

Valid property fields for SettingsCorridor.SettingsNameFormat.Corridor
<[Corridor First Assembly(CP)]>
<[Corridor First Baseline(CP)]>
<[Corridor First Profile(CP)]>
<[Next Counter(CP)]>

... for SettingsCorridor.SettingsNameFormat.CorridorSurface
<[Corridor Name(CP)]>
<[Next Corridor Surface Counter(CP)]>

...for SettingsCorridor.SettingsNameformat.ProfileFromFeatureLine
<[Next Counter(CP)]>

... for SettingsCorridor.SettingsNameFormat.AlignmentFromFeatureLine
<[Corridor Baseline Name(CP)]>

... for SettingsCorridor.SettingsNameFormat.AlignmentFromFeatureLine
<[Corridor Feature Code(CP)]>
<[Corridor Name(CP)]>
<[Next Counter(CP)]>
<[Profile Type]>

This sample sets the corridor name format:

```
// Get the Corridor ambient settings root object
CivilDocument doc = CivilApplication.ActiveDocument;
Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;
SettingsCorridor oCorridorSettings =
doc.Settings.GetFeatureSettings<SettingsCorridor>() as
SettingsCorridor;
// Set the template so new corridors are named "Corridor"
// followed by a unique number followed by the name of the
// corridor's first assembly in parenthesis.
oCorridorSettings.NameFormat.Corridor.Value = "Corridor
<[Next Counter(CP)]>(<[Corridor First Assembly(CP)]>)";
```

Default styles are set through the `SettingsCorridor.StyleSettings` property. The styles for corridor alignments, alignment labels, code sets, surfaces, feature lines, profiles, profile labels, and slope pattern are accessed through a series of string properties.

This sample sets the style of alignments in a corridor to the first alignment style in the document's collection of styles:

```
using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.
    Database.TransactionManager.StartTransaction())
{
    // Get the name of the first alignment style in the
collection.
    ObjectId alignId = doc.Styles.AlignmentStyles[0];
    Alignment oAlignment = ts.GetObject(alignId,
OpenMode.ForRead) as Alignment;
    // Assign the name to alignment style property.
    oCorridorSettings.Styles.Alignment.Value =
oAlignment.Name;
}
```

Assembly Ambient Settings

The assembly ambient settings object allows you to set the default name formats and default styles for assemblies. The name formats allow you to set how new assemblies, offset assemblies, and assembly groups are named. Each format can use elements from the following property fields:

Valid property fields for SettingsAssembly.NameFormat.Assembly
<[Next Counter(CP)]>

... for SettingsAssembly.NameFormat.Offset
<[Corridor Name(CP)]>

...for SettingsAssembly.NameFormat.Group
<[Next Counter(CP)]>

Subassembly Ambient Settings

The subassembly ambient settings object allows you to set the default name formats and default styles for subassembly objects. The name formats allow you to set how subassemblies created from entities and subassemblies created from macros are named. Each format can use elements from the following property fields:

... for SettingsSubassembly.SettingsNameFormat.CreateFromEntities
<[Macro Short Name(CP)]>
<[Next Counter(CP)]>
<[Subassembly Local Name(CP)]>

... for SettingsSubassembly.SettingsNameFormat.CreateFromEntities
<[Subassembly Side]>

... for SettingsSubassembly.SettingsNameFormat.CreateFromMacro
<[Macro Short Name(CP)]>
<[Next Counter(CP)]>
<[Subassembly Local Name(CP)]>
<[Subassembly Side]>

NOTE

The name of the default code style set cannot be set with the .NET API.

Corridors

Corridor Concepts

A corridor represents a path, such as a road, trail, railroad, or airport runway. The geometry of a corridor is defined by a horizontal alignment and a profile. Together, these form the baseline - the centerline of the 3D path of the corridor. Along the length of the baselines are a series of assemblies which define the cross-sectional shape of the alignment. Common points in each assembly are connected to form feature lines. Together the assemblies and feature lines form the 3D shape of a corridor. A corridor also has one or more surfaces which can be compared against an existing ground surface to determine the amount of cut or fill required.

Listing Corridors

The collection of all corridors in a document are held in the `CivilDocument.CorridorCollection` property.

The following sample displays the name and the largest possible triangle side of every corridor in a document:

```
public static void ListCorridors()
{
    CivilDocument doc = CivilApplication.ActiveDocument;
    Editor ed =
Application.DocumentManager.MdiActiveDocument.Editor;

    using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.
        Database.TransactionManager.StartTransaction())
    {
        foreach (ObjectId objId in doc.CorridorCollection)
        {
            Corridor myCorridor = ts.GetObject(objId,
OpenMode.ForRead) as Corridor;
          ed.WriteMessage("Corridor: {0}\nLargest possible
 triangle side: {1}\n",
                myCorridor.Name,
myCorridor.MaximumTriangleSideLength);
        }
    }

}
```

Creating Corridors

You cannot add new Corridors to a document using the .NET API.

Baselines

A baseline represents the centerline of the path of a corridor. It is based on an alignment (the horizontal component of the path) and a profile (the vertical component of the path). A corridor can contain more than one baseline if the corridor is modeling a complicated shape, such as an intersection. A baseline is made up of one or more baseline regions. Each region has its own assembly (its own cross section), so a corridor can have different shapes at different locations along its length.

Listing Baselines in a Corridor

The collection of all baselines in a corridor are contained in the `Corridor.Baselines` property, which is type `BaselineCollection`.

The following sample displays information about the underlying alignment and profile for every baseline in a corridor:

```
foreach (Baseline oBaseline in oCorridor.Baselines)
{
    Alignment oAlign = ts.GetObject(oBaseline.AlignmentId,
 OpenMode.ForRead) as Alignment;
    Profile oProfile = ts.GetObject(oBaseline.ProfileId,
OpenMode.ForRead) as Profile;
    ed.WriteMessage(@"Baseline information -
      Alignment     : {0}
      Profile       : {1}
      Start station : {2}
      End station   : {3}",
      oAlign.Name,
      oProfile.Name,
      oBaseline.StartStation,
      oBaseline.EndStation);
}
```

Adding a Baseline to a Corridor

Adding baselines to a Corridor is not supported in the .NET API.

Listing Baseline Regions

The collection of all the regions of a baseline are contained in the `Baseline.BaselineRegions` property.

The AutoCAD Civil 3D API does not include methods for creating new baseline regions, or manipulating existing regions.

The following sample displays the start and end station for every baseline region in a baseline:

```
foreach (BaselineRegion oBaselineRegion in
oBaseline.BaselineRegions)
{
    ed.WriteMessage(@"Baseline region information -
  Start station : {0}
  End station   : {1}\n",
  oBaselineRegion.StartStation,
  oBaselineRegion.EndStation);

}
```

Accessing and Modifying Baseline Stations

Assembly cross sections are placed at regular intervals along a baseline. The list of all stations where assemblies are located along a baseline can be retrieved using the `Baseline.SortedStations()` method, while all stations along a baseline region can be retrieved using the `BaselineRegion.SortedStations()` method.

```
double[] stations = oBaselineRegion.SortedStations();
ed.WriteMessage("Baseline Region stations: \n");
foreach (double station in stations){
    ed.WriteMessage("\tStation: {0}\n", station);
}
```

New stations can be added to baseline regions using the `AddStation()` method. Existing stations can be deleted using the `DeleteStation` method. `DeleteStation` includes an optional `tolerance` parameter, letting you specify a station within a range. You can list all of the stations added to a baseline region with the `BaselineRegion.GetAdditionalStation` method. `BaselineRegion.ClearAdditionalStations` removes all added stations within a baseline region and leaves only the original stations created at regular intervals.

```
// Add an assembly to the middle of the baseline region
double newStation = oBaselineRegion.StartStation +
    ((oBaselineRegion.EndStation -
oBaselineRegion.StartStation) / 2);
oBaselineRegion.AddStation(newStation, "New Station");
ed.WriteMessage("Added New Station: {0}", newStation);
```

```
// Remove the station located at the beginning of the
baseline region:
oBaselineRegion.DeleteStation(oBaselineRegion.StartStation);
```

Listing Offset Baselines

Within a baseline region, it is possible to have secondary baselines that are offset from the main baseline. The collection of these offset baselines are contained in the `BaselineRegion.OffsetBaselines` property. The collection contains two kinds of baselines derived from the `BaseBaseline` class. One is the hardcoded offset baseline (an instances of the `HardcodedOffsetBaseline` class) which is a constant distance from the main baseline for the entire length of the offset baseline. The other is the offset baseline (an instance of the `OffsetBaseline` class), which is a variable distance from the main baseline.

NOTE

The AutoCAD Civil 3D .NET API does not include methods for creating new offset baselines or hardcoded offset baselines.

This code examines each offset baseline within a baseline region:

```
foreach (BaseBaseline ob in oBaselineRegion.OffsetBaselines)
{
    ed.WriteMessage("Offset baseline: \n");
    switch (ob.BaselineType)
    {
        case CorridorBaselineType.OffsetBaseline:
            OffsetBaseline offb = (OffsetBaseline)ob;
            ed.WriteMessage("Offset baseline, station {0}
 to {1}\n",
                offb.StartStationOnMainBaseline,
offb.EndStationOnMainBaseline);
            ed.WriteMessage(" is offset by: {0} horizontal
 and {1} vertical at start\n",
          offb.GetOffsetElevationFromMainBaselineStation(offb.StartStationOnMainBaseline).X,
          offb.GetOffsetElevationFromMainBaselineStation(offb.StartStationOnMainBaseline).Y);
            ed.WriteMessage(" is offset by: {0} horizontal
 and {1} vertical at end\n",
          offb.GetOffsetElevationFromMainBaselineStation(offb.EndStationOnMainBaseline).X,
          offb.GetOffsetElevationFromMainBaselineStation(offb.EndStationOnMainBaseline).Y);
```

```
            break;

          case CorridorBaselineType.HardcodedOffsetBaseline:
              HardcodedOffsetBaseline hob =
(HardcodedOffsetBaseline)ob;
             ed.WriteMessage("Hardcoded offset baseline {0}
 \n",
                  hob.Name);
             ed.WriteMessage(" is offset by: {0} horizontal
 and {1} vertical\n",
                  hob.OffsetElevationFromMainBaseline.X,
                  hob.OffsetElevationFromMainBaseline.Y);
              break;



          default:
              break;
      }
  }
```

Assemblies and Subassemblies

An assembly is a pattern for the cross section of a corridor at a particular station. An assembly consists of a connected set of subassemblies, each of which are linked to a centerpoint or to other subassemblies. A subassembly consists of a series of shapes, links, and points. When an assembly is used to define the cross-section of a corridor, a series of applied assemblies (an object of type `AppliedAssembly`) is added to the corridor. Each applied assembly consists of a collection of applied subassemblies, which in turn consist of shapes, links, and points that have been positioned relative to a specific station along the corridor baseline (`CalculatedShape`, `CalculatedLink`, and `CalculatedPoint` respectively). An applied assembly also has direct access to all the calculated shapes, links, and points of its constituent applied subassemblies.

NOTE

The AutoCAD Civil 3D .NET API does not include methods for creating or modifying assemblies.

Listing Applied Assemblies in a Baseline Region

The collection of all applied assemblies used in a baseline region are contained in the `BaselineRegion.AppliedAssemblies` property.

The following sample displays information about the construction of an assembly for every assembly in a baseline region:

```
// List the applied assemblies in the baseline region
foreach (AppliedAssembly oAppliedAssembly in
oBaselineRegion.AppliedAssemblies)
{
    ed.WriteMessage("Applied Assembly, num shapes: {0},
num links: {1}, num points: {2}\n",
        oAppliedAssembly.Shapes.Count,
oAppliedAssembly.Links.Count,
oAppliedAssembly.Points.Count);


}
```

An `AppliedAssembly` object does not contain its baseline station position. Instead, each calculated point contains a property for determining its position with a baseline station, offset, and elevation called `CalculatedPoint.StationOffsetElevationToBaseline`. Each calculated shape contains a collection of all links that form the shape, and each calculated link contains a collection of all points that define the link. Finally, each shape, link, and point contain an array of all corridor codes that apply to that element.

This sample retrieves all calculated points in an applied assembly and prints their locations:

```
foreach (CalculatedPoint oPoint in oAppliedAssembly.Points)
{
    ed.WriteMessage("Point position: Station: {0}, Offset:
 {1}, Elevation: {2}\n",
        oPoint.StationOffsetElevationToBaseline.X,
        oPoint.StationOffsetElevationToBaseline.Y,
        oPoint.StationOffsetElevationToBaseline.Z);

}
```

Getting Applied Subassembly Information

An applied subassembly consists of a series of calculated shapes, links, and points, represented by objects of type `CalculatedShape`, `CalculatedLink`, and `CalculatedPoint` respectivly.

```
foreach (AppliedSubassembly oSubassembly in oASC)
{
    ed.WriteMessage("Applied subassembly: Station to
baseline: {0}, Offset to baseline: {1}, Elevation to
baseline: {2}\n",
       oSubassembly.OriginStationOffsetElevationToBaseline.X,
       oSubassembly.OriginStationOffsetElevationToBaseline.Y,
       oSubassembly.OriginStationOffsetElevationToBaseline.Z);
}
```

Applied subassemblies also contain an ObjectId reference to the archetype subassembly (of type `Subassembly`) in the subassembly database.

```
// Get information about the subassembly template:
ObjectId oID = oAppliedSubassembly.SubassemblyId;
Subassembly oSubassembly = ts.GetObject(oID,
OpenMode.ForRead) as Subassembly;
ed.WriteMessage("Subassembly name: {0}\n",
oSubassembly.Name);
```

Feature Lines

Feature lines are formed by connecting related points in each assembly along the length of a corridor baseline. These lines represent some aspect of the roadway, such as a sidewalk edge or one side of a corridor surface. Points become related by sharing a common *code*, a string property usually describing the corridor feature.

Each baseline has two sets of feature lines, one for lines that are positioned along the main baseline and one for lines that are positioned along any of the offset baselines.

NOTE

Creating feature lines from polylines is not supported in the .NET API. However, you can use the COM API `IAeccLandFeatureLine::AddFromPolyline()` method.

Listing Feature Lines Along a Baseline

The set of all feature lines along a main baseline are held in the `Baseline.MainBaselineFeatureLines` property, an object of type `BaselineFeatureLines`. This object contains information about all the feature lines, such as a list of all codes used. The `BaselineFeatureLines.FeatureLinesCol` property is a collection of feature line collections. Each feature line (an object of type `FeatureLine`) contains the code string used to create the feature line and a collection of all feature line points.

This sample lists all the feature line collections and feature lines along the main baseline. It also lists the code and every point location for each feature line.

```
// Get all the feature lines:
foreach (FeatureLineCollection oFeatureLineCollection in
oBaseline.MainBaselineFeatureLines.FeatureLineCollectionMap)
{
    ed.WriteMessage("Feature Line Collection\n# Lines in
collection: {0}\n",
        oFeatureLineCollection.Count);
    foreach (FeatureLine oFeatureLine in
oFeatureLineCollection)
    {
        ed.WriteMessage("Feature line code: {0}\n",
oFeatureLine.CodeName);
        // print out all point locations on the feature
line
        foreach (FeatureLinePoint oFeatureLinePoint in
oFeatureLine.FeatureLinePoints)
        {
            ed.WriteMessage("Point: {0},{1},{2}\n",
                oFeatureLinePoint.XYZ.X,
                oFeatureLinePoint.XYZ.Y,
                oFeatureLinePoint.XYZ.Z );
        }
    }
}
```

Listing Feature Lines Along Offset Baselines

As there can be many offset baselines in a single main baseline, the list of all feature lines along all offset baselines contains an extra layer. The `Baseline.OffsetBaselineFeatureLinesCol` property contains a collection of `BaselineFeatureLines` objects. These `BaselineFeatureLines` objects not only contain the feature lines just as for the main baseline, but also contain properties identifying which offset baseline each group of feature lines belong to.

This sample shows how to modify the previous sample for feature lines along offset baselines:

```
// Get all the offset feature lines:
foreach (BaselineFeatureLines oBaselineFeaturelines in
oBaseline.OffsetBaselineFeatureLinesCol)
{
    foreach (FeatureLineCollection oFeatureLineCollection
 in oBaselineFeaturelines.FeatureLineCollectionMap)
    {
        ed.WriteMessage("Feature Line Collection\n# Lines
 in collection: {0}\n",
        oFeatureLineCollection.Count);
        foreach (FeatureLine oFeatureLine in
oFeatureLineCollection)
        {
            ed.WriteMessage("Feature line code: {0}\n",
oFeatureLine.CodeName);
           // print out all point locations on the feature
 line
            foreach (FeatureLinePoint oFeatureLinePoint in
 oFeatureLine.FeatureLinePoints)
            {
                ed.WriteMessage("Point: {0},{1},{2}\n",
                    oFeatureLinePoint.XYZ.X,
                    oFeatureLinePoint.XYZ.Y,
                    oFeatureLinePoint.XYZ.Z);
            }
            }

        }
    }
```

Each offset baseline and hardcoded offset baseline also has direct access to the feature lines related to itself. The `BaselineFeatureLines` collection is accessed through the `RelatedOffsetBaselineFeatureLines` property in both types of offset baselines.

Corridor Surfaces

Corridor surfaces can represent the base upon which the corridor is constructed, the top of the finished roadway, or other aspects of the corridor. Such surfaces are represented by the `Surface` class and by the unrelated `CorridorSurface` class. `CorridorSurface` objects contain corridor-specific information about the surfaces, such as which feature line, point, and link codes were used to create it.

Listing Corridor Surfaces

The collection of all corridor surfaces for each corridor is held in the the `Corridor.CorridorSurfaces` property. Each corridor surface contains the boundary of the surface and a list of all point, link, and feature line codes used in the construction of the surface. Corridor surfaces also contain read-only references to the surface style ID and section style ID used in drawing the surface.

NOTE

The AutoCAD Civil 3D .NET API does not include methods for creating new corridor surfaces or modifying existing corridor surfaces.

This sample lists all the corridor surfaces within a corridor and specifies the point codes that make up each surface:

```
// List surfaces
foreach (CorridorSurface oCorridorSurface in
oCorridor.CorridorSurfaces)
{
    ed.WriteMessage("Corridor surface: {0}\n",
oCorridorSurface.Name);

    // Get the point codes for the surface.
    String[] oPointCodes = oCorridorSurface.PointCodes();

    ed.WriteMessage("Surface point codes:\n");
    foreach (String s in oPointCodes)
    {
        ed.WriteMessage("{0}\n", s);
    }

}
```

Listing Surface Boundaries

Two different objects are used to define the limits of a corridor surface: boundaries and masks. A boundary is a polygon representing the outer edge of a surface or the inside edge of a hole in a surface. A mask is a polygon representing the part of the surface that can be displayed. The collection of all the boundaries of a surface are stored in the `CorridorSurface.Boundaries` property and the collection of all masks are stored in the `CorridorSurface.Masks` property.

Boundaries (of type `CorridorSurfaceBoundary`) and masks (of type `CorridorSurfaceMask`) are both derived from the same base class (`CorridorSurfaceBaseMask`) and both have similar methods and properties. The array of points making up the border polygon is retrieved by calling the `PolygonPoints()` method. If the border was originally defined by selecting segments of feature lines, the collection of all such feature line components are contained in the `FeatureLineComponents` property.

NOTE

The AutoCAD Civil 3D .NET API does not include methods for creating or modifying corridor boundaries or masks.

This sample loops through all the boundaries of a corridor surface and displays information about each:

```
// List boundaries
foreach (CorridorSurfaceBoundary oCorridorSurfaceBoundary
 in oCorridorSurface.Boundaries)
{
    if (oCorridorSurfaceBoundary.BoundaryType ==
CorridorSurfaceBoundaryType.InsideBoundary)
        ed.WriteMessage("Inner Boundary: ");
    else
        ed.WriteMessage("Outer Boundary: ");

    ed.WriteMessage(oCorridorSurfaceBoundary.Name);

    // Get the points of the boundary polygon
    Point3d[] oPoints =
oCorridorSurfaceBoundary.PolygonPoints();
    ed.WriteMessage("\nNumber of points: {0}\n",
oPoints.Length);
    // Print the location of the first point. Usually
corridors
    // have a large number of boundary points, so we will
 not
    // bother printing all of them.
    ed.WriteMessage("Point 1: {0},{1},{2}\n",
        oPoints[0][0],
        oPoints[0][1],
        oPoints[0][2]);

    // Display information about each feature
    // line component in this surface boundary.
    ed.WriteMessage("Feature line components \n Count:
{0}\n",
       oCorridorSurfaceBoundary.FeatureLineComponents.Count);

    foreach (FeatureLineComponent oFeatureLineComponent in
 oCorridorSurfaceBoundary.FeatureLineComponents)
    {
        ed.WriteMessage("Code: {0}, Start station: {1},
End station: {2}\n",
            oFeatureLineComponent.FeatureLine.CodeName,
            oFeatureLineComponent.StartStation,
            oFeatureLineComponent.EndStation);
    }
}
```

Computing Cut and Fill

The .NET API doesn notexpose surface functionality, so it isn't possible to calculate cut and fill volumes by comparing surfaces. However, you can perform this task with the COM API. See for more information.

Styles

These style objects control the visual appearance of applied assemblies.

Assembly Style

The collection of all assembly style objects are found in the `CivilDocument.Styles.AssemblyStyles` property. The assembly style object contains properties for adjusting the marker types for the assembly attachment points, and each of the standard `MarkerType` properties. While you can create new styles and edit existing styles, you cannot assign a style to an existing assembly using the AutoCAD Civil 3D .NET API.

```
using (Transaction ts =
Application.DocumentManager.MdiActiveDocument.
    Database.TransactionManager.StartTransaction())
{
    ObjectId objId =
doc.Styles.AssemblyStyles.Add("Style1");
    AssemblyStyle oAssemblyStyle = ts.GetObject(objId,
OpenMode.ForWrite) as AssemblyStyle;
    objId = oAssemblyStyle.MarkerStyleAtMainBaselineId;
    MarkerStyle oMarker = ts.GetObject(objId,
OpenMode.ForWrite) as MarkerStyle;
    oMarker.CustomMarkerStyle =
CustomMarkerType.CustomMarkerX;
    oMarker.MarkerDisplayStylePlan.Color =
Color.FromColorIndex(ColorMethod.ByAci, 10);
    oMarker.MarkerDisplayStylePlan.Visible = true;

    ts.Commit();
}
```

Link Style

The collection of all link style objects are found in the `CivilDocument.Styles.LinkStyles` property. This style object contains properties for adjusting the visual display of assembly and subassembly links.

NOTE

Link style objects are not used directly with link objects, but are instead used with roadway style sets.

```
// Add a new link style to the document:
objId = doc.Styles.LinkStyles.Add("Style2");
LinkStyle oLinkStyle = ts.GetObject(objId,
OpenMode.ForWrite) as LinkStyle;
oLinkStyle.LinkDisplayStylePlan.Color =
Color.FromColorIndex(ColorMethod.ByAci, 80);
oLinkStyle.LinkDisplayStylePlan.Visible = true;

ts.Commit();
```

Shape Style

The collection of all shape style objects are found in the `CivilDocument.ShapeStyles` property. This style object contains properties for adjusting the visual display of assembly and subassembly shapes, including the outline and the inside area.

NOTE

Shape style objects are not used directly with shape objects, but are instead used with roadway style sets.

```
// Create a new shape style and change it so that it has
// an orange border and a yellow hatch fill.
objId = doc.Styles.ShapeStyles.Add("Style3");
ShapeStyle oShapeStyle = ts.GetObject(objId,
OpenMode.ForWrite) as ShapeStyle;
// 50 = yellow
oShapeStyle.AreaFillDisplayStylePlan.Color =
Color.FromColorIndex(ColorMethod.ByAci, 50);
oShapeStyle.AreaFillDisplayStylePlan.Visible = true;
oShapeStyle.AreaFillHatchDisplayStylePlan.HatchType =
HatchType.PreDefined;
oShapeStyle.AreaFillHatchDisplayStylePlan.Pattern = "LINE";
// 30 = orange
oShapeStyle.BorderDisplayStylePlan.Color =
Color.FromColorIndex(ColorMethod.ByAci, 30);
oShapeStyle.BorderDisplayStylePlan.Visible = true;

ts.Commit();
```

Roadway Style Sets

The visual display of applied assemblies is defined by roadway style sets, which are a set of shape styles and link styles assigned to shapes and links that use specified code strings. The collection of all style sets are found in the `CivilDocument.Styles.CodeSetStyles` property. A style set is itself a collection of `CodeSetStyleItem` objects. Each style set item has a `CodeSetStyleItem.CodeStyle` property that can reference either an existing shape style object or link shape object. New style set items are added to a style set though the `CodeSetStyleCollection.Add()` method which takes parameters describing the kind of style object, the code string, and the style object itself.

NOTE

You cannot set the CodeSetStyle to be the currently used style with the .NET API. However, you can get the ObjectId for the currently used style by calling CodeSetStyle.GetCurrentStyleSetId().

```
// Create a new style set using our previously created
styles.
objId = doc.Styles.ShapeStyles.Add("Style Set 1");
CodeSetStyle oCodeSetStyle = ts.GetObject(objId,
OpenMode.ForWrite) as CodeSetStyle;
oCodeSetStyle.Add("TOP", doc.Styles.LinkStyles["Style2"]);
oCodeSetStyle.Add("BASE", doc.Styles.ShapeStyles["Style3"]);

ts.Commit();
```

Points

This chapter covers creating and using Coordinated Geometry (COGO) Points, exposed by the `CogoPoint` class, with the .NET API. It describes accessing the collection of all points in the `CivilDocument`, adding and removing points, assigning User Defined Properties (UDPs) to points, and working with `PointGroup` objects to organize points. It also describes creating and applying point styles and label styles to points.

Using the Points Collection

All points in a document are held in a CogoPointCollection object accessed through the CivilDocument.CogoPoints property. In addition to the common collection properties and methods, this collection also exposes methods for working with large numbers of points at once. For example, points can be added to the collection either individually, or from a Point3dCollection.

The following sample adds a collection of randomly generated points to the document's point collection, and then accesses each point in the collection directly to calculate the average elevation:

```
[CommandMethod("C3DSAMPLES", "AverageCogoElevation",
CommandFlags.Modal)]
public void AverageCogoElevation()
{
    using (Transaction tr = startTransaction())
    {

        // _civildoc is the active CivilDocument instance.

        CogoPointCollection cogoPoints =
_civildoc.CogoPoints;
        Point3d[] points = { new Point3d(4927, 3887, 150),
 new Point3d(5101, 3660, 250), new Point3d(5144, 3743, 350)
 };
        Point3dCollection locations = new
Point3dCollection(points);

        cogoPoints.Add(locations);

        // Compute the average elevation of all the points
 in a document.
        double avgElevation = 0;
        foreach (ObjectId pointId in cogoPoints)
        {
            CogoPoint cogoPoint =
pointId.GetObject(OpenMode.ForRead) as CogoPoint;
            avgElevation += cogoPoint.Elevation;
        }

        avgElevation /= cogoPoints.Count;
        _editor.WriteMessage("Average elevation: {0}
\nNumber of points: {1}", avgElevation, cogoPoints.Count);

        tr.Commit();
    }
}
```

Using Points

Coordinated Geometry Points (COGO points) are more complex than AutoCAD point nodes, which have only coordinate data. A `CogoPoint` object, in addition to a location, also has properties such as a unique ID number, name, raw (field) description, and full (expanded) description. The point number is unique, and is automatically assigned when the point is created. You can change the point number either by setting the `PointNumber` property directly, or by using the `Renumber()` method. Setting the property directly will throw an exception if another point exists with the specified value, while the `Renumber()` method will use the settings for resolving point numbering conflicts to choose another point number. You can also set the `PointNumber` property for multiple points using the `CogoPointCollection.SetPointNumber()` method.

The full description property is read-only once a point is created.

The `CogoPoint` object's `Location` property is read-only. However, you can read and change the local position using the `Easting`, `Northing` and `Elevation` properties. The point's location can also be specified by using the `Grideasting` and `GridNorthing` properties or the `Latitude` and `Longitude` properties, depending on the coordinate and transformation settings of the drawing.

A `CogoPoint` object is either a drawing point or a project point. Project points have the `isProjectPoint` property set to true, and have additional project information contained by the `IsCheckedOut` and `ProjectVersion` properties.

Attempting to get or set these properties for points that have `isProjectPoint == false` raises an exception, so it is a good idea to check that property first.

`CogoPoint` objects also have several read-only override properties that contain values that have been overridden by `PointGroup` settings. These include `ElevationOverride`, `FullDescriptionOverride`, `LabelStyleIdOverride`, `RawDescriptionOverride`, and `StyleIdOverride`.

This sample adds a new point to the document's collection of points and sets some of its properties.

```
[CommandMethod("C3DSAMPLES", "CreatePoint",
CommandFlags.Modal)]
public void CreatePoint()
{
    using (Transaction tr = startTransaction())
    {
        // _civildoc is the active CivilDocument instance.

        Point3d location = new Point3d(4958, 4079, 200);
        CogoPointCollection cogoPoints =
_civildoc.CogoPoints;
        ObjectId pointId = cogoPoints.Add(location);
        CogoPoint cogoPoint =
pointId.GetObject(OpenMode.ForWrite) as CogoPoint;
        cogoPoint.PointName = "point1";
        cogoPoint.RawDescription = "Point description";

        tr.Commit();
    }
}
```

Bulk Editing Points

The `CogoPointCollection` class provides several methods for changing the properties of multiple points with a single action. Most of the `Set<Property>()` methods have three versions:

1 Set the property for a single `CogoPoint` (identified by `ObjectId`) in the collection .

2 Set the property for all points in a list to a single value .

3 Set the property for all points in a list to a value in a corresponding list of values.

The list of points can be a sub-collection of points, or you can pass the `CivilDocument.CogoPoints` collection to process all points in the document.

Here is an example of setting `Elevation`, `RawDescription` and `DescriptionFormat` properties for all points in the drawing using the bulk editing methods.

```
// Change a couple of properties using bulk editing methods
cogoPoints.SetElevationByOffset(cogoPoints, 3.00);
cogoPoints.SetRawDescription(cogoPoints, "NEW_DESC");
// Sets Full Description = Raw Description:
cogoPoints.SetDescriptionFormat(cogoPoints, "$*");
```

Point User-Defined Properties

User-defined properties (UDPs) allow users to attach additional data to points. (UDPs can also be assigned to Parcels). UDPs are organized into groups called UDP Classifications, which in turn are assigned to `PointGroups`. UDPs can also be “unclassified”. When a `UDPClassification` is associated with a `PointGroup` object (using its `UseCustomClassification()` method), all the UDP definitions in the `UDPClassification` are assigned to each point in the `PointGroup`. UDP values are unique for each point, and can be changed individually. Each point gets a UDP’s default value (if `UseDefault` is true for the `UDP`) upon assignment.

Each instance of a UDP is a class derived from the `UDP` base class, with additional properties depending on the data type. `UDPDouble` and `UDPInteger` types have upper and lower bound properties, to express a range. `UDPEnumeration` types have a `GetEnumerationValues()` method to get an array of all defined values.

To create a `UDP`, you must first create and populate an `AttributeTypeInfo<type>` object (there is one for each UDP type), and pass it to the `CreateUDP()` method for an existing `UDPClassification`. UDPs have a GUID as well as a name, and the name+GUID combination is guaranteed to be a unique identifier. The `CreateUDP()` method takes an optional GUID parameter, so that you can create UDPs with a specific name+GUID combination. This allows you to create identical UDPs in multiple drawings.

For more information about user-defined properties and classifications, see User-Defined Property Classifications in the AutoCAD Civil 3D User Guide.

This sample creates a new user-defined property classification for points called “Example”, and then adds a new user-defined property with upper and lower bounds and a default value:

```
[CommandMethod("C3DSAMPLES", "UDPExample",
CommandFlags.Modal)]
public void UDPExample()
{
    using (Transaction tr = startTransaction())
    {
        // _civildoc is the active CivilDocument instance.

        UDPClassification udpClassification =
_civildoc.PointUDPClassifications.Add("Example");
        AttributeTypeInfoInt attributeTypeInfoInt = new
AttributeTypeInfoInt("Int UDP");
        attributeTypeInfoInt.DefaultValue = 15;
        attributeTypeInfoInt.UpperBoundValue = 20;
        attributeTypeInfoInt.LowerBoundValue = 10;
        UDP udp =
udpClassification.CreateUDP(attributeTypeInfoInt);

        // assign a point group
        ObjectId pointGroupId =
_civildoc.PointGroups.AllPointGroupsId;
        PointGroup pointGroup =
pointGroupId.GetObject(OpenMode.ForWrite) as PointGroup;
        pointGroup.UseCustomClassification("Example");


        tr.Commit();
    }
}
```

This example illustrates accessing the collection of all Point UDPClassifications in a document, and then reading each UDP in each UDPClassification.

```
[CommandMethod("C3DSAMPLES", "ListUDPs",
CommandFlags.Modal)]
public void ListUDPs()
{
    using (Transaction tr = startTransaction())
    {
        // _civildoc is the active CivilDocument instance.
        foreach (UDPClassification udpClassification in
_civildoc.PointUDPClassifications)
        {
            _editor.WriteMessage("\n\nUDP Classification:
 {0}\n", udpClassification.Name);
            foreach (UDP udp in udpClassification.UDPs)
            {
                _editor.WriteMessage(" * UDP name: {0}
guid: {1}, description: {2}, default value: {3}, use
default? {4}\n",
                    udp.Name, udp.Guid, udp.Description,
udp.DefaultValue, udp.UseDefaultValue);

                _editor.WriteMessage("\tUDP type: {0}\n",
 udp.GetType().ToString());

                var udpType = udp.GetType().Name;
                switch (udpType)
                {
                    // Booleans and String types do not
define extra properties
                    case "UDPBoolean":
                    case "UDPString":
                        break;

                    case "UDPInteger":
                        UDPInteger udpInteger =
(UDPInteger)udp;
                      _editor.WriteMessage("\tUpper value:
 {0}, inclusive? {1}, Lower bound value: {2}, inclusive?
{3}\n",
                            udpInteger.UpperBoundValue,
udpInteger.UpperBoundInclusive, udpInteger.LowerBoundValue,
 udpInteger.LowerBoundInclusive);
                        break;

                    case "UDPDouble":
                        UDPDouble udpDouble =
(UDPDouble)udp;
                      _editor.WriteMessage("\tUpper value:
 {0}, inclusive? {1}, Lower bound value: {2}, inclusive?
{3}\n",
                            udpDouble.UpperBoundValue,
udpDouble.UpperBoundInclusive, udpDouble.LowerBoundValue,
 udpDouble.LowerBoundInclusive);
                        break;

                    case "UDPEnumeration":
                        UDPEnumeration udpEnumeration =
(UDPEnumeration)udp;
                       _editor.WriteMessage("\tEnumeration
 values: {0}\n", udpEnumeration.GetEnumValues());
                        break;
                }
            }
        }

        tr.Commit();
    }
}
```

Point Groups

A point group is a collection that defines a subset of the points in a document. Points may be grouped for a number of reasons, such as points that share common characteristics or are used to perform a common task (for example, define a surface). A collection of all point groups in a drawing is held in a document's `CivilDocument.PointGroups` property. Add a new point group by using the `CivilDocument.PointGroups.Add()` method and specifying a unique identifying string name. The `ObjectId` for a new, empty point group is returned.

```
// _civildoc is the active CivilDocument instance.
ObjectId pointGroupId = _civildoc.PointGroups.Add("Example
 Point Group");
PointGroup pointGroup =
pointGroupId.GetObject(OpenMode.ForRead) as PointGroup;
```

Using Point Groups

Once a point group has been created, you can perform actions upon all the points in that group in a single operation. You can override point elevations, descriptions, styles, and label styles.

```
// Check to see if a particular point exists in the
PointGroup
ObjectId pointId = promptForEntity("Select a point",
typeof(CogoPoint));
CogoPoint cogoPoint = pointId.GetObject(OpenMode.ForRead)
 as CogoPoint;

if (pointGroup.ContainsPoint(cogoPoint.PointNumber){
    _editor.WriteMessage("Point {0} is not part of
PointGroup {1}",
        cogoPoint.PointName, pointGroup.Name);
}

// Set the elevation of all points in the PointGroup to
100
pointGroup.ElevationOverride.FixedElevation=100;
pointGroup.ElevationOverride.ActiveOverrideType =
PointGroupOverrideType.FixedValue;
pointGroup.IsElevationOverriden = true;
```

Point groups can also be used to define or modify a TIN surface. The `TinSurface.PointGroupsDefinition` property is a collection of point groups. When a point group is added to the collection, every point in the point group is added to the TIN surface.

Adding Points to Point Groups with Queries

Points can be selected and added to a PointGroup using either a standard (`StandardPointGroupQuery` object) or custom (`CustomPointGroupQuery` object) query, which both inherit from the `PointGroupQuery` base class.

`StandardPointGroupQuery` encapsulates the “basic method” of creating a query using the GUI, which is described in the “Creating a Point Group Using the Basic Method” section of the Civil 3D User Guide. This method uses the tabs on the **Create Point Group** dialog to match points by raw descriptions, include or exclude specific points or ranges of points, and to include points from other point groups. To accomplish the same effect using the API, you first create an instance of a `StandardPointGroupQuery` object, and then set the various `Include*` and `Exclude*` properties to create the query.

`StandardPointGroupQuery` has these properties for building a query:

`StandardPointGroupQuery` Properties

Properties	Possible Values
`IncludeElevations` `ExcludeElevations`	Any combination of, separated by commas: ■ A single number ■ An elevation range specified by a lower and upper bound separated by a hyphen (for example, 1-100) ■ A > or < followed by a number. This specifies all elevations greater-than or less-than the number. **Example**: "<-100,1-100,110.01,>200" – specifies all points whose elevations meet one of the following criteria: less than -100, equal to or between 1 and 100, equal to 110.01, or greater than 200.

`IncludeFullDescriptions`
`ExcludeFullDescriptions`
`IncludeRawDescriptions`
`ExcludeRawDescriptions`
	One or more descriptions, separated by commas. The * is a wildcard matching any string.
	Example: "IP*" matches all descriptions that start with IP

`IncludeNames` `ExcludeNames`	One or more point names separated by commas. The * is a wildcard matching any string.
`IncludeNumbers` `ExcludeNumbers`	Any combination of, separated by commas: ■ An individual point number ■ A Point number range specified by a lower and upper bound separated by a hyphen (for example, 100-105).

All of the `Include*` properties are ORed together, and all the `Exclude*` properties are ORed together to create the final query string. The example below illustrates how the query string is built from properties:

```
// _civildoc is the active CivilDocument instance.
ObjectId pointGroupId = _civildoc.PointGroups.Add("Example
 Point Group1");
```

```
PointGroup pointGroup =
pointGroupId.GetObject(OpenMode.ForWrite) as PointGroup;

StandardPointGroupQuery standardQuery = new
StandardPointGroupQuery();
standardQuery.IncludeElevations = "100-200";
standardQuery.IncludeFullDescriptions = "FLO*";
standardQuery.IncludeNumbers = ">2200";
standardQuery.ExcludeElevations = "150-155";
standardQuery.ExcludeNames = "BRKL";
standardQuery.UseCaseSensitiveMatch = true;

pointGroup.SetQuery(standardQuery);
pointGroup.Update();

_editor.WriteMessage("Number of points selected: {0}\n
Query string: {1}\n\n",
    pointGroup.PointsCount, standardQuery.QueryString);

// output:
// Query string: (FullDescription='FLO*' OR
PointElevation=100-200 OR
// PointNumber>2200) AND '' NOT (Name='BRKL' OR
PointElevation=150-155)
```

The `CustomPointGroupQuery` lets you specify a query string directly, and requires a more advanced knowledge of query operators. This method of query creation allows you to create nested queries that cannot be specified using the `StandardPointGroupQuery` object.

Custom queries are made up of expressions, and expressions have three parts:

1. A property
2. A comparison operator: > < >= <= =
3. A value

Multiple expressions are separated by logical set operators: AND, OR and NOT. Precedence of expressions and expression sets can be specified using parentheses.

The precedence of evaluation in queries is:

1. Expressions in parentheses, with the innermost expressions evaluated first.

2 NOT
3 Comparisons
4 AND
5 OR

The example below illustrates creating a point group and adding a custom query to it:

```
ObjectId pointGroupId2 = _civildoc.PointGroups.Add("Example
 Point Group2");
PointGroup pointGroup2 =
pointGroupId2.GetObject(OpenMode.ForWrite) as PointGroup;
CustomPointGroupQuery customQuery = new
CustomPointGroupQuery();
string queryString = "(RawDescription='GR*') AND
(PointElevation>=100 AND PointElevation<=300)";
customQuery.QueryString = queryString;
pointGroup2.SetQuery(customQuery);
pointGroup2.Update();
_editor.WriteMessage("PointGroup2: \n # points selected:
{0}\n",
    pointGroup2.PointsCount);
```

Pending changes for a `PointGroup` can be accessed from the `PointGroup.GetPendingChanges()` method, which returns a `PointGroupChangeInfo` object. The pending change information for a `PointGroup` is updated when:

- A new point is added to the drawing that matches the query for the `PointGroup`.
- An existing point in the `PointGroup` is removed.

Note that changes are not registered when the query itself is changed.

The example below adds a point that matches the custom query illustrated above, and removes a point from the `PointGroup`, registering both an added and removed point change in the `PointGroupChangeInfo` for the `PointGroup`.

```
Point3d point3d = new Point3d(100,200,225); // elevation
will be matched
string rawDesc = "GRND"; // raw description will be matched
_civildoc.CogoPoints.Add(point3d, rawDesc);

// Point # 779 is in the point group:
_civildoc.CogoPoints.Remove(779);
```

```
PointGroupChangeInfo pointGroupChangeInfo =
pointGroup2.GetPendingChanges();
_editor.WriteMessage("Point group {0} pending changes: \n
 add: {1} \n remove: {2}\n",
    pointGroup2.Name,
pointGroupChangeInfo.PointsToAdd.Length,
pointGroupChangeInfo.PointsToRemove.Length);

// changes are applied with update:
pointGroup2.Update();
```

For more information about point group queries, see Understanding Point Group Queries in the .

Point Style

The `PointStyle` class defines how a point is drawn. Labels attached to `CogoPoints` are styled with objects of type `LabelStyle`. This section covers creating and applying point and label styles to `CogoPoints`. It also covers using point description keys to match points by name and apply styles to them.

Creating Point Styles

A point style is a group of settings that define how a point is drawn. These settings include marker style, marker color and line type, and label color and line type. Point objects can use any of the point styles that are currently stored in the document. Styles are assigned to a point through the point's `CogotPoint.StyleId` property. Existing point styles are stored in the document's `CivilDocument.Styles.PointStyles` collection.

You can also create custom styles and add them to the document's collection of point styles. First, add a new style to the document's list of styles using the `CivilDocument.Styles.PointStyles.Add()` method. This method returns the `ObjectId` of a new style object that is set with all the properties of the default style. You can then make the changes to the style object you require.

The display settings for `PointStyle` objects are accessed with the style's `GetDisplay*(),GetLabelDisplay*()` and `GetMarkerDisplay*()` methods for the view mode (Model, Plan, Profile or Section) for the style.

The point marker type is set by the `PointStyle`'s `MarkerType` property, and can be a symbol, custom marker, or an AutoCAD point style. Custom markers are set using the `CustomMarkerStyle` and `CustomMarkersuperimposeStyle` properties. Symbol markers are set using the `MarkerSymbolName` property, which is a string that names an AutoCAD BLOCK symbol in the drawing.

Points assigned to `PointGroups` may have their styles overridden by the `PointGroup`. You can check the overridden style ID using the `LabelStyleIdOverride` and `StyleIdOverride` properties.

This sample creates a new points style, adjusts the style settings, and the assigns the style to point "cogoPoint1":

```
// Create a point style that uses a custom marker,
// a Plus sign inside a square.
ObjectId pointStyleId =
_civildoc.Styles.PointStyles.Add("Example Point Style");
PointStyle pointStyle =
pointStyleId.GetObject(OpenMode.ForWrite) as PointStyle;
pointStyle.MarkerType =
PointMarkerDisplayType.UseCustomMarker;
pointStyle.CustomMarkerStyle =
CustomMarkerType.CustomMarkerPlus;
pointStyle.CustomMarkerSuperimposeStyle =
CustomMarkerSuperimposeType.Square;

// Assign the style to a point object.
cogoPoint1.StyleId = pointStyleId;
```

Creating Point Label Styles

Any text labels or graphical markers displayed at the point location are set by assigning a label style `ObjectId` to the `CogoPoint.LabelStyleId` property. The collection of these label styles is accessed through the `CivilDocument.Styles.LabelStyles.PointLabelStyles` property.

Point label styles can use the following property fields in the contents of any text components:

Valid property fields for LabelStyle TextComponent Contents
<[Name(CP)]>

Valid property fields for LabelStyle TextComponent Contents
<[Point Number]>
<[Northing(Uft|P4|RN|AP|Sn|OF)]>
<[Easting(Uft|P4|RN|AP|Sn|OF)]>
<[Raw Description(CP)]>
<[Full Description(CP)]>
<[Point Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[Latitude(Udeg|FDMSdSp|P6|RN|DPSn|CU|AP|OF)]>
<[Longitude(Udeg|FDMSdSp|P6|RN|DPSn|CU|AP|OF)]>
<[Grid Northing(Uft|P4|RN|AP|Sn|OF)]>
<[Grid Easting(Uft|P4|RN|AP|Sn|OF)]>
<[Scale Factor(P3|RN|AP|OF)]>
<[Convergence(Udeg|FDMSdSp|P6|RN|AP|OF)]>
<[Survey Point]>

For more general information about creating and using label styles, see the Label Styles (page 27) section.

Using Point Description Keys

Point description keys are a method for attaching style, label style, and orientation to point locations in a drawing - possibly imported from a text file which lacks such information. Keys are objects of type `PointDescriptionKey`. The `PointDescriptionKey.Code` property is a pattern matching code. If any new points are created with a description that matches the code of an existing key, the point is assigned all the settings of that key.

The wildcards “?” and “*” are allowed in the code. Keys can contain either constant scale or rotation values for points or can assign orientation values depending on parameters passed through the description string. Point description keys are held in sets, objects of type `PointDescriptionKeySet`. The collection of all sets in a document are accessed with the `PointDescriptionKeySetCollection.GetPointDescriptionKeySets()` static method.

The collection of all key sets also exposes a `SearchOrder` property, which lets you specify the order that key sets are searched for matches when description keys are applied. Key sets at the beginning of the `SearchOrder` collection are searched first and therefore have higher priority. In the example below, this property is accessed, and the last item is moved to the first position.

```
// Create a Key Set in the collection of all Key Sets:
ObjectId pointDescriptionKeySetId =

PointDescriptionKeySetCollection.GetPointDescriptionKeySets(_acaddoc.Database).Add("Example
 Key Set");
PointDescriptionKeySet pointDescriptionKeySet =
  pointDescriptionKeySetId.GetObject(OpenMode.ForWrite) as
 PointDescriptionKeySet;

// Create a new key in the set we just made.  Match with
any description starting with "GRND".
ObjectId pointDescriptionKeyId =
pointDescriptionKeySet.Add("GRND*");
PointDescriptionKey pointDescriptionKey =
  pointDescriptionKeyId.GetObject(OpenMode.ForWrite) as
PointDescriptionKey;

// Assign chosen styles and label styles to the key
pointDescriptionKey.StyleId = pointStyleId;
pointDescriptionKey.ApplyStyleId = true;
pointDescriptionKey.LabelStyleId = labelStyleId;
pointDescriptionKey.ApplyLabelStyleId = true;

// Turn off the scale override, and instead scale
// to whatever is passed as the first parameter
pointDescriptionKey.ApplyDrawingScale = false;
pointDescriptionKey.ScaleParameter = 1;
pointDescriptionKey.ApplyScaleParameter = true;
pointDescriptionKey.ApplyScaleXY = true;

// Turn off the rotation override and rotate
// all points using the key 45 degrees clockwise
pointDescriptionKey.FixedMarkerRotation = 0.785398163; //
 radians
pointDescriptionKey.RotationDirection =
RotationDirType.Clockwise;
pointDescriptionKey.ApplyFixedMarkerRotation = true;

// Before applying the description keys, we can set the
order the key sets are searched
// when point description keys are applied.
// In this example, we take the last item and make it the
 first.
ObjectIdCollection pdKeySetIds =

PointDescriptionKeySetCollection.GetPointDescriptionKeySets(_acaddoc.Database).SearchOrder;
ObjectId keySetId = pdKeySetIds[pdKeySetIds.Count-1];
pdKeySetIds.Remove(keySetId);
pdKeySetIds.Insert(0, keySetId);
PointDescriptionKeySetCollection.GetPointDescriptionKeySets(_acaddoc.Database).SearchOrder
 =
  pdKeySetIds;

// Apply to point group
pointGroup.ApplyDescriptionKeys();
```

Creating Custom Subassemblies Using .NET

Overview

This chapter describes how to create a custom subassembly using Visual Basic .NET. This is the currently supported and preferred method of creating custom subassemblies.

This chapter covers design considerations, the structure of subassembly programs, and an example subassembly in VB that you can use as a template for your own custom subassemblies.

Subassembly Changes

The “target mapping’ feature in AutoCAD Civil 3D 2013 has changed, which affects how custom subassemblies are written. This feature now allows a subassembly to target object types in addition to alignments and profiles associated with the corridor that it requires to define its geometry. For more information about this feature, see Setting and Editing Targets in the .

There are four changes to the way you write a custom subassembly:

1. New parameter types in `ParamLogicalNameType`
2. New target collections in `corridorState`
3. Targets are now objects
4. The `CalcAlignmentOffsetToThisAlignment` utility method is changed

New parameter types in ParamLogicalNameType

A subassembly can now target an offset target and elevation target (instead of an alignment and a profile), which are represented by new parameter types in `ParamLogicalNameType`. If you want to support the new target types in your subassemblies, you need to replace:

- `ParamLogicalNameType.Alignment` with
  - `ParamLogicalNameType.OffsetTarget` — offset targets that are alignments, feature lines, survey figures and AutoCAD polylines OR
  - `ParamLogicalNameType.OffsetTargetPipe` — offset targets that are pipe networks

- `ParamLogicalNameType.Profile` with
  - `ParamLogicalNameType.ElevationTarget` — elevation targets that are profiles, feature lines, survey figures and AutoCAD 3D polylines OR
  - `ParamLogicalNameType.ElevationTargetPipe` — elevation targets that are pipe networks

New target collections in corridorState

To get the offset and elevation target collections from the `corridorState` object, use `ParamsOffsetTarget` and `ParamsElevationTarget` instead of `ParamsAlignment` and `ParamsProfile`. All the offset targets (including network pipe offset targets) are in `ParamsOffsetTarget`, and all elevation targets (including network pipe elevation targets) are in `ParamsElevationTarget`. Here's an example from the *BasicLaneTransition.vb* sample in the Sample VB.NET Subassembly (page 160) section below:

```
        Dim oParamsLong As ParamLongCollection
        oParamsLong = corridorState.ParamsLong

        Dim oParamsOffsetTarget As
ParamOffsetTargetCollection
        oParamsOffsetTarget =
corridorState.ParamsOffsetTarget
```

Targets are now objects

Targets are now objects, instead of alignment or profile IDs. Now `WidthOffsetTarget` is defined for offset targets, and `SlopeElevationTarget` is defined for elevation targets, so you can declare targets as objects instead of IDs. Here's an example from the *BasicLaneTransition.vb* sample in the Sample VB.NET Subassembly (page 160) section below:

```
        Dim offsetTarget As WidthOffsetTarget 'width or
offset target
        offsetTarget = Nothing
        Dim elevationTarget As SlopeElevationTarget 'slope
 or elevation target
        elevationTarget = Nothing
```

Changes to CalcAlignmentOffsetToThisAlignment()

The `CalcAlignmentOffsetToThisAlignment()` utility method now calculate the offset from this alignment to the offset target. This method no longer returns the station value; it now returns the XY coordinate of the offset target at the point perpendicular to the alignment's station.

You can also now use the `SlopeElevationTarget.GetElevation` method to get an elevation on an elevation target directly, instead of using `CalcAlignmentOffsetToThisAlignment()`. Here's an example from the *BasicLaneTransition.vb* sample in the Sample VB.NET Subassembly (page 160) section below:

```
                'get elevation on elevationTarget
                Try
```

```
                    dOffsetElev =
elevationTarget.GetElevation(oCurrentAlignmentId, _
                        corridorState.CurrentStation,
Utilities.GetSide(vSide))
                Catch
                    Utilities.RecordWarning(corridorState,
 _
                        CorridorError.LogicalNameNotFound,
 _
                        "TargetHA", "BasicLaneTransition")
                    dOffsetElev =
corridorState.CurrentElevation + vWidth * vSlope
                End Try
```

Designing Custom Subassemblies

This section describes the requirements for designing a custom subassembly.

Naming Custom Subassemblies

- Do not use spaces or other special characters.
- Use a combination of upper and lower case letters, with uppercase letters reserved for the first character of each word.
- Group subassemblies by making the type of component the first word. For example, in the AutoCAD Civil 3D Corridor Modeling catalogs all lane subassembly names begin with “Lane”, all shoulders with “Shoulder”, and so on.

Attachment and Insertion Methodology

Most subassembly components have a single point of attachment and extend in one direction or the other from that point. However, there are some exceptions to this general rule.

The list below describes the attachment and insertion methodology for three categories of subassemblies: medians, components joining two roadways, and rehabilitation and overlay.

- **Medians.** Medians tend to be inserted in both the left and right directions simultaneously about a centerline (which is not necessarily the corridor baseline alignment). Furthermore, the attachment point may not be a point on the median surface links. For example, the attachment point for a depressed median subassembly may be above the median ditch at the elevation of the inside edges-of-traveled-way.
- **Components Joining Two Roadways.** When modeling separated roadways in a single corridor model, it is often necessary to insert intersection fill slopes, or to connect from one edge-of-roadway to another. Typically, you assemble the components for as much of the first roadway as possible, switch baselines and assemble the components for the second roadway, then use special subassemblies to connect between the two roadways. In this case, two attachment points are needed. Do this by creating a subassembly with a normal attachment point on one side, and which attaches to a previously defined marked point on the other.
- **Rehabilitation and Overlay.** Typically, subassemblies that are used to strip pavement, level, and overlay existing roads are placed based on calculations involving the shape of the existing roadway section, rather than using a design centerline alignment and profile. For example, a pavement overlay subassembly may require a minimum vertical distance from the existing pavement for a given design slope. A lane-widening subassembly may need to attach to the existing edge-of-traveled-way and match the existing lane slope.

User-defined vs. Hard-coded Parameters

Determine which of the geometric dimensions, behavior, and methodology will be hard-coded into the subassembly, and which will be controlled by user-defined input parameters.

One approach is to specify that a majority of items can be controlled by user input. This can add time and complexity to using the subassembly. Another approach is to make it so that it cannot be adapted to different situations. Generally, widths, depths, and slopes should be variable, not fixed. A compromise is to include a larger number of inputs, but provide default values usable in most design situations.

A good example of where to use hard-coded dimensions is with structural components, such as barriers and curb-and-gutter shapes. If there are five commonly-used variations of the same basic shape with different dimensions, it may be better to provide five separate subassemblies with hard-coded dimensions rather than making the user define the dimensions on a single subassembly. For example, users may be comfortable selecting from separate subassemblies for curb types A-E, which have predefined, hard-coded dimensions. You can always provide a generic subassembly with variable dimensions for scenarios where the common ones do not apply.

Input Parameter Types

The table below describes the various types of input parameters:

Input Parameter	Description
Widths	The cross-sectional horizontal distance between two points on the roadway assembly. Widths are generally given as positive numeric values, and extend in the direction of insertion (left or right) of the subassembly. Many components that require a width are likely candidates for using an alignment as an optional target parameter. The width is then calculated at each station to tie to the alignment, if one is given.
Offsets	The cross-sectional horizontal distance from the corridor baseline to a point on the roadway assembly. The difference between an Offset and a Width is that Widths are measured from some point on the assembly, while Offsets are measured from the corridor baseline. Positive and negative values indicate positions right or left of the baseline. Components requiring an offset are also likely to use an alignment to allow a calculated offset.
% Slopes	Lanes, shoulders, and other components usually have a slope defined as ratio of rise-to-run. There are two common conventions for how these are expressed. They can either be a unitless ratio (-0.05), or a percent value (-5). Both of these examples define a 5% slope downwards. The same convention should be used for all subassemblies in a catalog. In some cases, you may want the component to have a variable slope, tying to a profile. The profile name can be given as a target parameter.

Input Parameter	Description
Ratio Slopes	Cut slopes, fill slopes, ditch side slopes, median slopes, and many other roadway components are commonly expressed as a run-to-rise ratio, such as 4 : 1. These may be signed or un-signed values, depending on circumstances. For example, a fill slope is always downward, so it may not be necessary to force the user to enter a value like "-4".
Point, Link, and Shape Codes	In most cases, point, link, and shape codes should be hard-coded to ensure that consistent codes are assigned across the entire assembly. The primary exception is with generic link subassemblies that allow users to add links to the assembly as needed. These might be used for paved or unpaved finish grade, structural components, pavement subsurfaces, and many other unanticipated components. In these scenarios, the end-user assigns point, link, and shape codes that coordinate with the overall assembly.

Superelevation Behavior and Subassemblies

Make sure you consider differences in component behavior, such as when the roadway is in a normal crown condition or is in superelevation. Gutter and median subassemblies may also be designed to exhibit different behaviors in normal and superelevated sections.

The superelevation properties of the corridor alignment define the lane and shoulder slopes at all stations on the roadway. However, the way these slopes are applied depends on a combination of how the subassemblies are manipulated in layout mode and the internal logic of the subassemblies. Different agencies have varying methodologies. The code behind the subassemblies makes it possible to adapt to just about any situation.

Most importantly you need to determine where the superelevation pivot point is located, and how that point relates to the design profile grade line (PGL). Pivot point/PGL combinations that are commonly encountered include:

- Pivot point and PGL are both at the crown of road.
- Pivot point and PGL are at the inside edge-of-traveled-way on a divided road.
- Pivot point and PGL are at one edge-of-traveled-way on an undivided road.

- Pivot point is at the edge-of-traveled-way on the inside of the curve, while the PGL is at the centerline of the road.
- On a divided road with crowned roadways, the PGL is at the crown points, while the pivot point is at the inside edge-of-traveled-ways.
- On a divided road with uncrowned roadways, the PGL and pivot point is above the median at the centerline.

Whatever the situation, the subassemblies must be designed so that they can be placed with the correct behavior.

Sometimes the roadway components have special behavior in superelevated sections. Some examples of special superelevation behavior:

- **Broken Back Subbase.** Some agencies put a break point in the subbase layer on the high side of superelevation. The subbase parallels the finish grade up to a certain point, then extends at a downward slope until it intersects the shoulder or clear zone link.
- **Shoulder Breakover.** Usually a maximum slope difference, or breakover, must be maintained between the travel lane and the shoulder, or between the paved and unpaved shoulder links.
- **Curbs-and-Gutters.** Some agencies require that gutters on the high side of a superelevated road tip downward toward the low side, while others leave the gutters at their normal slope.

NOTE

When writing your custom subassembly, avoid writing code that makes AutoCAD calls and interrupts AutoCAD Civil 3D subassembly operations during runtime. For example, avoid building selection sets.

Axis of Rotation Pivot Point Calculation Notes

This section explains the methods that must be called when creating subassemblies that use superelevation and support axis of rotation calculations.

- If one subassembly in an assembly is specified as a potential pivot, all subassemblies between it and the baseline that use superelevation must report their superelevation cross slope data to the corridor using the Autodesk.CIvil.Runtime.CorridorState method:

  ```
  Public Sub SetAxisOfRotationInformation (ByVal
  isPotentialPivot As Boolean, ByVal superElevationSlope
  As Double, ByVal superElevationSlopeType As
  ```

```
Autodesk.Civil.Land.SuperelevationCrossSegmentType, ByVal
 isReversedSlope As Boolean)
```

superElevationSlope is the cross slope used in the subassembly. For axis of rotation to work correctly, it must be the value obtained from lane type *superElevationSlopeType* in the superelevation table. If the value obtained from the table is adjusted in any way, axis of rotation will not work properly.

isReversedSlope set to True indicates that the cross slope applied is the negated value obtained from the superelevation table.

- The LaneSuperelevationAOR subassembly, which is provided in the AutoCAD Civil 3D content, supports axis of rotation pivot points. If LaneSuperelevationAOR (or a custom subassembly that supports pivot points) is not contained in the assembly, the pivot type specified in the Calculate Superelevation wizard is ignored, and the Baseline Pivot Type is applied. A custom subassembly can support axis of rotation pivot points by passing in True for the *isPotentialPivot* argument when calling SetAxisOfRotationInformation().
- To determine the Centers pivots in a divided crowned roadway, the crown points specified in the subassemblies supporting pivots are used. To specify a crown point to be used in axis of rotation calculations when building the corridor, use the Autodesk.Civil.Runtime.CorridorState method:

  ```
  Public Sub SetAxisOfRotationCrownPoint (ByVal
  nCrownPointIndex As UInteger)
  ```

  To display the crown point in assembly layout mode, the subassembly property "SE AOR Crown Point For Layout" must be set with the index of the crown point. To set this property, call the following method in Utilities class:

  ```
  Public Shared Sub SetSEAORCrownPointForLayout (ByVal
  corridorState As CorridorState, ByVal nCrownPoint As
  Integer)
  ```

- By default, the axis of rotation calculation assumes that the recorded cross slope is applied to the full width of the subassembly in its calculation. The starting offset will be the smallest offset in a subassembly drawn to the right, and the largest offset for a subassembly drawn to the left. If the subassembly is applying the cross slope to just a portion of the subassembly, record the starting and ending offsets of this range with the corridor using the Autodesk.Civil.Runtime.CorridorState method:

```
Public Sub SetAxisOfRotationSERange (ByVal
dApplySE_StartOffset As Double, ByVal dApplySE_EndOffset
 As Double)
```

- Only one set of axis of rotation information can be recorded per assembly. This means that the calculated delta Y value is the same no matter which point in the subassembly is used as the attachment point. Therefore, if a subassembly applies multiple cross slopes, or a single cross slope affects only certain point in the subassembly, only the pivot location will be correctly calculated and recorded to the corridor.
- If a subassembly uses superelevation, but does not record the superelevation cross slope data with the corridor, it should notify the corridor so that the corridor can issue a warning that axis of rotation results may be unexpected. To notify the corridor, call the following method in Utilities class:

  ```
  Public Shared Sub SetSEAORUnsupportedTag (ByVal
  corridorState As CorridorState)
  ```

  If a subassembly conditionally supports axis of rotation, it may need to clear the unsupported flag. To clear the flag, call the following method in Utilities class:

  ```
  Public Shared Sub ClearSEAORUnsupportedTag (ByVal
  corridorState As CorridorState)
  ```

Creating Subassembly Help Files

Each subassembly included in the AutoCAD Civil 3D Corridor Modeling catalog has a Help file that provides detailed construction and behavior information. You can display the Help file for the AutoCAD Civil 3D Corridor Modeling subassemblies using any of the following methods:

- **From a Tool Palette.** Right-click a subassembly in a tool palette, then click Help.
- **From a Tool Catalog.** Right-click a subassembly in a tool catalog, then click Help.
- **From the Subassembly Properties dialog box Parameters tab.** Right-click a subassembly in the Prospector tree, then click Properties ➤ Parameters tab ➤ Subassembly Help.

When creating custom subassemblies, you should also create a custom Help file to accompany the subassembly. You can use a Microsoft Compiled HTML

Help file (*.chm*) to create the subassembly Help file. The Help file content and style should be similar to that in the AutoCAD Civil 3D Subassembly Reference Help. The table below describes the sections that should be included, as a minimum, in subassembly Help files. This information is required so that users understand the subassembly behavior and intended use.

Section	Description
Title	The name of the selected subassembly should appear prominently as the top heading of the subassembly Help file.
Descriptions	A brief description of the subassembly that includes the type of component the subassembly creates (for example, a lane, median, or shoulder), special features, and situations it is designed to cover.
Subassembly Diagram	The subassembly diagram should be a schematic showing the geometry of the component that is created by the subassembly. Diagrams should label as many of the input parameters as feasible, especially those pertaining to dimensions and slopes. You may need to include multiple subassembly diagrams for different behavior and/or conditions in order to include all of the possible input items. The subassembly diagram should also show the subassembly reference point, which is the point on the subassembly where it is attached when building an assembly layout. It is useful to adopt diagramming conventions that help the user understand the operations. For example, the subassemblies included in the AutoCAD Civil 3D Corridor Modeling catalog use bold blue lines to represent links that are added to the assembly by the subassembly. This helps to show adjacent roadway components that the subassembly might attach to in a lighter line with a background color. Ideally, dimension lines and labels should also be a different color.
Attachment	Describes where the attachment point is located relative to the subassembly links.
Input Parameters	Describes each of the user-definable input parameters that can be specified when using the subassembly. These should precisely match the parameter names and order seen by the user when using the assembly layout tool, and should describe the effect of each parameter. These are best presented in a table that includes a description of each parameter, the type of input expected, and default values for metric or imperial unit projects. For input parameters for slope values, note that there are two common ways of specifying slopes: as a percent value like -2%, or as a

Section	Description
	run-to-rise ratio like 4 : 1. Any slope parameter should clearly specify which type is expected. In the subassemblies included in the AutoCAD Civil 3D Corridor Modeling catalog, the convention is to precede the word “Slope” with the “%” character in the parameter name if a percent slope is expected. Otherwise a ratio value is required. Note the practice of using positive numeric values for both cut and fill slopes. If a slope parameter is known to be used only in a fill condition, it should not be necessary for the user to have to specify a negative slope value. However, in a more generic situation, for example with the LinkWidthAndSlope subassembly, a signed value may be necessary.
Target Parameters	Describes the parameters in the subassembly that can be mapped to one or more target objects. Input parameters are defined when building an assembly in layout mode. Target parameters are substitutions that can be made for input parameters when applying the assembly to a corridor model. Typically, subassembly parameters that can use a target object to define a width or an offset can use the following types of objects to define that width or offset: alignments, AutoCAD polylines, feature lines, or survey figures. Similarly, subassembly parameters that can use a target object to define an elevation can use the following types of objects to define that elevation: profiles, AutoCAD 3D polylines, feature lines, or survey figures. Subassemblies that can use a target object to define a surface can only use a surface object to define that surface. A few subassemblies allow you to use pipe network objects as targets, such as the TrenchPipe subassemblies. A typical scenario is a travel lane where the width is a numeric input parameter, which can use an alignment as a target parameter to replace the numeric width. The given numeric width is used when displaying the lane in layout mode. If an alignment is given, the width is calculated at each station during corridor modeling to tie to the offset of the alignment. For more information, see Setting and Editing Targets in the .
Behavior	Describes the behavior of the subassembly in detail. If necessary, this section should include diagrams showing different behaviors in different conditions. This section should provide both the subassembly programmer and the end user with all of the information needed to understand exactly what the subassembly does in all circumstances. Subheadings are recommended if the Behavior section covers several different topics.

Section	Description
Layout Mode Operation	During the process of creating an assembly from subassemblies, also known as the assembly layout mode, specific information such as alignment offsets, superelevation slopes, profile elevations, and surface data, are not known. The Layout Mode Operation section of the subassembly Help file describes how the subassembly is displayed in the assembly layout mode. Layout mode refers to an assembly that has not yet been applied to a corridor. Some subassemblies behave differently in different situations. For example, a Daylight type of subassembly may create different geometric shapes depending on whether it is in a cut or fill situation. Shoulders may behave differently for normal crown and superelevated roadways. In layout mode, the subassembly designer must specify some arbitrary choices as to how the subassembly is displayed. It should appear as much like the final result in the corridor model as possible. Lanes and shoulders, for example, should be shown at typical normal crown slopes. Where there is alternate geometry, such as for the cut and fill daylight cases, both cases should be shown. Also, links that extend to a surface should be shown with arrowheads indicating the direction of extension.
Layout Mode Diagram	A diagram illustrating layout mode behavior and visual representation is useful if layout mode behavior and/or visual representation of the subassembly differs significantly between layout mode when the assembly and its associated subassemblies are applied to a corridor.
Point, Link, and Shape Codes	Describes the items that are hard-coded into the subassembly, including dimensions, point codes, link codes, and shape codes. Common practice is to reference the point, link, and shape codes to labels on the coding diagram.
Coding Diagram	The coding diagram has a twofold purpose. First, it labels the point, link, and shape numbers referred to in the previous section. Secondly, it provides the subassembly programmer with a numbering scheme for points, links, and shapes. These should correspond to the array indices used in the script for points, links, and shapes. This is to make it easier to later modify or add to the subassembly.

After creating the custom Help files for custom subassemblies, you must reference the Help files in the tool catalog *.atc* file associated with the subassemblies. For more information, see Sample Tool Catalog ATC File (page 175).

Structure of Subassembly Programs

The Subassembly Template (SATemplate.vb)

All custom subassemblies are defined as classes that inherit from the `SATemplate` class. `SATemplate` provides four methods that you can override in your own class to provide functionality of your subassembly. They are described in the following table:

Overridable Method	Purpose for Overriding
GetLogicalNamesImplement (input: CorridorState)	Define the list of target parameters that appear in the "Set All Logical Names" dialog box used when creating a corridor model.
GetInputParametersImplement (input: CorridorState)	Define the list of input parameters, including their names, types, and default values.
GetOutputParametersImplement (input: CorridorState)	Define the list of output parameters, including their names, types, and default values. If a parameter is to be used for both input and output, that property is specified in this method.
DrawImplement (input: CorridorState)	**Must be overridden.** Contains the code for accessing parameter values, adjusting the shape of the subassembly, and then adding the points, links, and shapes that make up your subassembly to an existing assembly.

SATemplate.vb is located in the *<AutoCAD Civil 3D Install Directory>\Sample\Civil 3D API\C3DStockSubAssemblies* directory.

The Corridor State Object

A reference to an object of type `CorridorState` is passed to each of the `SATemplate` methods you override. The `CorridorState` object is the primary interface between the custom subassembly and the collection of points, links, and shapes of the current assembly which the subassembly is to connect to. It provides references to the current alignment, profile, station, offset, elevation, and style, which may affect the appearance of the subassembly. It also includes several parameter buckets used for collecting parameters of types boolean, long, double, string, alignment, profile, surface, and point. Each parameter is defined by a literal name and a value.

The `CorridorState` methods provide useful calculation functions for corridor design. These include the following:

IntersectAlignment	Finds the intersection of a cross-sectional line with an offset alignment.
IntersectLink	Finds the intersection of a cross-sectional line with a link on the assembly.
IntersectSurface	Finds the intersection of a cross-sectional line with a surface.
IsAboveSurface	Determines if a subassembly point is above or below a surface.
SampleSection	Constructs a set of cross section links from a surface.
SoeToXyz XyzToSoe	Converts between station, offset, elevation coordinates and X,Y,Z coordinates.

Support Files (CodesSpecific.vb, Utilities.vb)

You can also use the two support files *CodesSpecific.vb* and *Utilities.vb* in your subassembly.

CodesSpecific.vb provides the `CodeType` and `AllCodes` structures and the global variable `Code` – an instance of an `AllCodes` structure with all code information filled.

Utilities.vb provides a series of shared helper functions for error handling, computing subassembly geometry, attaching code strings, and other tasks. For example, to report a “parameter not found” error to the AutoCAD Civil 3D event viewer, use the following line:

```
Utilities.RecordError( _
   corridorState, _
   CorridorError.ParameterNotFound, _
   "Edge Offset", _
   "BasicLaneTransition")
```

The following table lists all the functions within the `Utility` class:

Utility function	Description
RecordError	Sends an error message to the Event Viewer.
RecordWarning	Sends a warning to the Event Viewer.
AdjustOffset	For a given offset from an offset baseline, this function computes the actual offset from the base baseline.
GetVersion	Returns the version number of the subassembly as specified in the ***.atc*** file.
GetAlignmentAndOrigin	Returns the assembly and the origin point for the subassembly.
CalcElevationOnSurface	Given a surface Id, alignment Id, a station, and an offset from the station, this returns the elevation of the surface at that location.
GetRoundingCurve	Returns an array of points along a curve where it will be tessellated, given a series of parameters describing the curve and how it is to be tessellated.
CalcAlignmentOffsetToThisAlignment	Calculates the offset from this alignment to an offset target, and returns the XY coordinate of

Utility function	Description
	the offset target at the point perpendicular to this alignment's station
AddCodeToLink	Adds a series of code strings to a particular link. The parameter **i** identifies which set of code strings to use. The parameter **iLinks** contains the collection of all links in an applied subassembly. The parameter **linkIndex** identifies which link in the link collection is given the code strings. The **strArrCode** parameter is a two-dimensional array of code strings. The first dimension identifies which set of codes to use, and the second dimension contains a variable-length array of code strings.
AddCodeToPoint	Adds a series of code strings to a particular point. The parameter **i** identifies which set of code strings to use. The parameter **iPoints** contains the collection of all points in an applied subassembly. The parameter **pointIndex** identifies which point in the point collection is given the code strings. The **strArrCode** parameter is a two-dimensional array of code strings. The first dimension identifies which set of codes to use, and the second dimension contains a variable-length array of code strings.
IsProjectUnitsFeet	Returns `True` if the corridor is modeled in Imperial units, `False` if modeled in metric.
GetProjectUnitsDivisor	Returns the value to go from the general units of measurement to smaller units of measurement. If the corridor is modeled in feet, this will return 12 to allow you to compute the number of inches. If the corridor is modeled in meters, this will return 1000 to allow you to compute the number of millimeters.

Utility function	Description
GetSlope	Returns the percent slope of the alignment superelevation at the specified assembly section. The first parameter is a two-letter string indicates the part of the roadway to use. The first letter can be "S" for shoulder or "L" for lane. The second letter can be "I" for inside or "O" for outside. To determine the slope of the left side of the road, set the fourth parameter to `True`. To determine the slope of the right side, set the fourth parameter to `False`.
AddPoints	Given arrays of X and Y locations and code strings, this creates an array of `Point` objects (`AeccRoadwayPoint` objects) and a `PointsCollection` object (`AeccRoadwayPoints` object).
GetMarkedPoint	Given the string name of a point, returns the point object.

The *CodesSpecific.vb* and *Utilities.vb* files are located in the *<AutoCAD Civil 3D Install Directory>\Sample\Civil 3D API\C3DStockSubAssemblies* directory.

Sample VB.NET Subassembly

The following class module defines the BasicLaneTransition subassembly provided in the Stock Subassemblies catalog. The original source code for this and all other subassemblies that come with AutoCAD Civil 3D can be found in the *<AutoCAD Civil 3D Install Directory>\Sample\Civil 3D API\C3DStockSubAssemblies\Subassemblies* directory.

Before reviewing the code you should familiarize yourself with the subassembly, how it behaves in the cut and fill conditions, the point and link codes to be assigned, and the point and link numbers specified in the subassembly coding diagram. Refer to the BasicLaneTransition subassembly Help for this information.

```
Option Explicit On
Option Strict Off
Imports System.Math
Imports DBTransactionManager =
Autodesk.AutoCAD.DatabaseServices.TransactionManager
'
**********************************************************************
'
**********************************************************************
'
**********************************************************************
'          Name: BasicLaneTransition
'
'   Description: Creates a simple cross-sectional
representation of a corridor
'                lane composed of a single closed shape.
 Attachment origin
'                is at top, most inside portion of lane.
 The lane can
'               transition its width and cross-slope based
 on the position
'                supplied by an optional horizontal and
vertical alignment.
'
' Logical Names: Name        Type     Optional   Default
value    Description
'
--------------------------------------------------------------
'                TargetHA    Alg      yes        none
         Horizontal alignment to transition to
'                TargetVA    Profile  yes        none
         Vertical alignment to transition to
'
'    Parameters: Name            Type     Optional
Default Value    Description
'
--------------------------------------------------------------
'                Side            long     yes        Right
           specifies side to place SA on
'                Width           double   yes        12.0
             width of lane
'               Depth           double   yes        0.667
           depth of coarse
'                Slope           double   yes        -0.02
```

```
              cross-slope of lane
'               TransitionType  long     yes        2
            hold grade, move to offset HA
'              InsertionPoint  long     yes       kCrown
          Specifies insertion point of the lane either at
 (a) Crown or (b) Edge of Travel Way
'              CrownPtOnInside Long     no        g_iTrue
         Specifies that inside edge of travelway to be
coded as Crown
'
**********************************************************************
Public Class BasicLaneTransition
    Inherits SATemplate
    Private Enum InsertionPoint
        kCrown = 0
        kEdgeOfTravelWay = 1
    End Enum
    Private Enum TransitionTypes ' Transition types
supported
        kHoldOffsetAndElevation = 0
        kHoldElevationChangeOffset = 1
        kHoldGradeChangeOffset = 2
        kHoldOffsetChangeElevation = 3
        kChangeOffsetAndElevation = 4
    End Enum
    '
----------------------------------------------------------------------
    ' Default values for input parameters
    Private Const SideDefault = Utilities.Right
    Private Const InsertionPointDefault =
InsertionPoint.kCrown
    Private Const CrownPtOnInsideDefault = Utilities.IFalse
    Private Const LaneWidthDefault = 12.0#
    Private Const LaneDepthDefault = 0.667
    Private Const LaneSlopeDefault = -0.02    '0.25 inch
per foot
    Private Const HoldOriginalPositionDefault =
TransitionTypes.kHoldOffsetAndElevation
    Protected Overrides Sub GetLogicalNamesImplement(ByVal
 corridorState As CorridorState)
        MyBase.GetLogicalNamesImplement(corridorState)
        ' Retrieve parameter buckets from the corridor
state
```

```
        Dim oParamsLong As ParamLongCollection
        oParamsLong = corridorState.ParamsLong
        ' Add the logical names we use in this script
        Dim oParamLong As ParamLong
        oParamLong = oParamsLong.Add("TargetHA",
ParamLogicalNameType.OffsetTarget)
        oParamLong.DisplayName = "690"
        oParamLong = oParamsLong.Add("TargetVA",
ParamLogicalNameType.ElevationTarget)
        oParamLong.DisplayName = "691"
    End Sub
    Protected Overrides Sub
GetInputParametersImplement(ByVal corridorState As
CorridorState)
        MyBase.GetInputParametersImplement(corridorState)
        ' Retrieve parameter buckets from the corridor
state
        Dim oParamsLong As ParamLongCollection
        oParamsLong = corridorState.ParamsLong
        Dim oParamsDouble As ParamDoubleCollection
        oParamsDouble = corridorState.ParamsDouble
        ' Add the input parameters we use in this script
        oParamsLong.Add(Utilities.Side, SideDefault)
        oParamsLong.Add("InsertionPoint",
InsertionPointDefault)
        oParamsLong.Add("CrownPtOnInside",
CrownPtOnInsideDefault)
        oParamsDouble.Add("Width", LaneWidthDefault)
        oParamsDouble.Add("Depth", LaneDepthDefault)
        oParamsDouble.Add("Slope", LaneSlopeDefault)
        oParamsLong.Add("TransitionType",
HoldOriginalPositionDefault)
    End Sub
    Protected Overrides Sub DrawImplement(ByVal
corridorState As CorridorState)
        ' Retrieve parameter buckets from the corridor
state
        Dim oParamsDouble As ParamDoubleCollection
        oParamsDouble = corridorState.ParamsDouble
        Dim oParamsLong As ParamLongCollection
        oParamsLong = corridorState.ParamsLong
        Dim oParamsOffsetTarget As
ParamOffsetTargetCollection
```

```
        oParamsOffsetTarget =
corridorState.ParamsOffsetTarget
        Dim oParamsElevationTarget As
ParamElevationTargetCollection
        oParamsElevationTarget =
corridorState.ParamsElevationTarget
      '---------------------------------------------------
        ' flip about Y-axis
        Dim vSide As Long
        Try
            vSide = oParamsLong.Value(Utilities.Side)
        Catch
            vSide = SideDefault
        End Try
        Dim dFlip As Double
        dFlip = 1.0#
        If vSide = Utilities.Left Then
            dFlip = -1.0#
        End If
      '---------------------------------------------------
        ' Transition type
        Dim vTransitionType As Long
        Try
            vTransitionType =
oParamsLong.Value("TransitionType")
        Catch
            vTransitionType = HoldOriginalPositionDefault
        End Try
      '---------------------------------------------------
        ' Insertion Ponit
        Dim vInsertionPoint As Long
        Try
            vInsertionPoint =
oParamsLong.Value("InsertionPoint")
        Catch
            vInsertionPoint = InsertionPointDefault
        End Try
        Dim vCrownPtOnInside As Long
        Try
            vCrownPtOnInside =
oParamsLong.Value("CrownPtOnInside")
        Catch
            vCrownPtOnInside = CrownPtOnInsideDefault
```

```
          End Try
        '--------------------------------------------------------
          ' BasicLaneTransition dimensions
          Dim vWidth As Double
          Try
              vWidth = oParamsDouble.Value("Width")
          Catch
              vWidth = LaneWidthDefault
          End Try
          Dim vDepth As Double
          Try
              vDepth = oParamsDouble.Value("Depth")
          Catch
              vDepth = LaneDepthDefault
          End Try
          Dim vSlope As Double
          Try
              vSlope = oParamsDouble.Value("Slope")
          Catch
              vSlope = LaneSlopeDefault
          End Try
        '--------------------------------------------------------
          ' Get version, and convert values if necessary
          Dim sVersion As String
          sVersion = Utilities.GetVersion(corridorState)
          If sVersion <> Utilities.R2005 Then
              'need not change
          Else
              'R2005
              'convert %slope to tangent value
              vSlope = vSlope / 100
          End If
          Dim nVersion As Integer
          nVersion = Utilities.GetVersionInt(corridorState)
          If nVersion < 2010 Then
              vCrownPtOnInside = Utilities.ITrue
          End If
        '--------------------------------------------------------
          ' Check user input
          If vWidth <= 0 Then
              Utilities.RecordError(corridorState,
CorridorError.ValueShouldNotBeLessThanOrEqualToZero,
"Width", "BasicLaneTransition")
```

```
            vWidth = LaneWidthDefault
        End If
        If vDepth <= 0 Then
            Utilities.RecordError(corridorState,
CorridorError.ValueShouldNotBeLessThanOrEqualToZero,
"Depth", "BasicLaneTransition")
            vDepth = LaneDepthDefault
        End If
        ' Calculate the current alignment and origin
according to the assembly offset
        Dim oCurrentAlignmentId As ObjectId
        Dim oOrigin As New PointInMem
        Utilities.GetAlignmentAndOrigin(corridorState,
oCurrentAlignmentId, oOrigin)
      '-------------------------------------------------------
        ' Define codes for points, links and shapes
        Dim sPointCodeArray(0 To 4, 0) As String
        Dim sLinkCodeArray(0 To 2, 0 To 1) As String
        Dim sShapeCodeArray(0 To 1) As String
       FillCodesFromTable(sPointCodeArray, sLinkCodeArray,
 sShapeCodeArray, vCrownPtOnInside)
      '-------------------------------------------------------
       ' Get alignment and profile we're currently working
 from
        Dim offsetTarget As WidthOffsetTarget 'width or
offset target
        offsetTarget = Nothing
        Dim elevationTarget As SlopeElevationTarget 'slope
 or elvation target
        elevationTarget = Nothing
        Dim dOffsetToTargetHA As Double
        Dim dOffsetElev As Double
        If corridorState.Mode = CorridorMode.Layout Then
            vTransitionType =
TransitionTypes.kHoldOffsetAndElevation
        End If
        Dim dXOnTarget As Double
        Dim dYOnTarget As Double
        Select Case vTransitionType
            Case TransitionTypes.kHoldOffsetAndElevation
           Case TransitionTypes.kHoldElevationChangeOffset
                'oHA must exist
                Try
```

```
                    offsetTarget =
oParamsOffsetTarget.Value("TargetHA")
                Catch
                    'Utilities.RecordError(corridorState,
 CorridorError.ParameterNotFound, "Edge Offset",
"BasicLaneTransition")
                    'Exit Sub
                End Try
                'get offset to targetHA
                If False =
Utilities.CalcAlignmentOffsetToThisAlignment(oCurrentAlignmentId,
 corridorState.CurrentStation, offsetTarget,
Utilities.GetSide(vSide), dOffsetToTargetHA, dXOnTarget,
dYOnTarget) Then
                   Utilities.RecordWarning(corridorState,
 CorridorError.LogicalNameNotFound, "TargetHA",
"BasicLaneTransition")
                    dOffsetToTargetHA = vWidth +
oOrigin.Offset
                Else
                   If (dOffsetToTargetHA = oOrigin.Offset)
 Or ((dOffsetToTargetHA > oOrigin.Offset) And (vSide =
Utilities.Left)) Or _
                        ((dOffsetToTargetHA <
oOrigin.Offset) And (vSide = Utilities.Right)) Then
                     Utilities.RecordWarning(corridorState,
 CorridorError.ValueInABadPosition, "TargetHA",
"BasicLaneTransition")
                        dOffsetToTargetHA = vWidth +
oOrigin.Offset
                    End If
                End If
            Case TransitionTypes.kHoldGradeChangeOffset
                'oHA must exist
                Try
                    offsetTarget =
oParamsOffsetTarget.Value("TargetHA")
                Catch
                    'Utilities.RecordError(corridorState,
 CorridorError.ParameterNotFound, "Edge Offset",
"BasicLaneTransition")
                    'Exit Sub
                End Try
                'get offset to targetHA
                If False =
Utilities.CalcAlignmentOffsetToThisAlignment(oCurrentAlignmentId,
 corridorState.CurrentStation, offsetTarget,
Utilities.GetSide(vSide), dOffsetToTargetHA, dXOnTarget,
dYOnTarget) Then
                    Utilities.RecordWarning(corridorState,
 CorridorError.LogicalNameNotFound, "TargetHA",
"BasicLaneTransition")
                    dOffsetToTargetHA = vWidth +
oOrigin.Offset
                Else
                    If (((dOffsetToTargetHA >
oOrigin.Offset) And (vSide = Utilities.Left)) Or _
                        ((dOffsetToTargetHA <
oOrigin.Offset) And (vSide = Utilities.Right))) Then
                     Utilities.RecordWarning(corridorState,
 CorridorError.ValueInABadPosition, "TargetHA",
"BasicLaneTransition")
                        dOffsetToTargetHA = vWidth +
oOrigin.Offset
                    End If
                End If
           Case TransitionTypes.kHoldOffsetChangeElevation
                'oVA must exist
                Try
                    elevationTarget =
oParamsElevationTarget.Value("TargetVA")
                Catch
                    'Utilities.RecordError(corridorState,
 CorridorError.ParameterNotFound, "Edge Elevation",
"BasicLaneTransition")
                    'Exit Sub
                End Try
                Dim db As Database =
HostApplicationServices.WorkingDatabase
                Dim tm As DBTransactionManager =
db.TransactionManager
                Dim oProfile As Profile = Nothing
                'get elevation on elevationTarget
                Try
                    dOffsetElev =
elevationTarget.GetElevation(oCurrentAlignmentId,
corridorState.CurrentStation, Utilities.GetSide(vSide))
                Catch
                   Utilities.RecordWarning(corridorState,
 CorridorError.LogicalNameNotFound, "TargetHA",
"BasicLaneTransition")
                    dOffsetElev =
corridorState.CurrentElevation + vWidth * vSlope
                End Try
            Case TransitionTypes.kChangeOffsetAndElevation
                'both oHA and oVA must exist
                Try
                    offsetTarget =
oParamsOffsetTarget.Value("TargetHA")
                Catch
                    'Utilities.RecordError(corridorState,
 CorridorError.ParameterNotFound, "Edge Offset",
"BasicLaneTransition")
                    'Exit Sub
                End Try
                Try
                    elevationTarget =
oParamsElevationTarget.Value("TargetVA")
                Catch
                    'Utilities.RecordError(corridorState,
 CorridorError.ParameterNotFound, "Edge Elevation",
"BasicLaneTransition")
                    'Exit Sub
                End Try
                'get elevation on elevationTarget
                Try
                    dOffsetElev =
elevationTarget.GetElevation(oCurrentAlignmentId,
corridorState.CurrentStation, Utilities.GetSide(vSide))
                Catch
                   Utilities.RecordWarning(corridorState,
 CorridorError.LogicalNameNotFound, "TargetHA",
"BasicLaneTransition")
                    dOffsetElev =
corridorState.CurrentElevation + vWidth * vSlope
                End Try
                'get offset to targetHA
                If False =
Utilities.CalcAlignmentOffsetToThisAlignment(oCurrentAlignmentId,
 corridorState.CurrentStation, offsetTarget,
Utilities.GetSide(vSide), dOffsetToTargetHA, dXOnTarget,
dYOnTarget) Then
                 Utilities.RecordWarning(corridorState,
 CorridorError.LogicalNameNotFound, "TargetHA",
"BasicLaneTransition")
                 dOffsetToTargetHA = vWidth +
oOrigin.Offset
                Else
                   If (dOffsetToTargetHA = oOrigin.Offset)
 Or ((dOffsetToTargetHA > oOrigin.Offset) And (vSide =
Utilities.Left)) Or _
                       ((dOffsetToTargetHA <
oOrigin.Offset) And (vSide = Utilities.Right)) Then
                     Utilities.RecordWarning(corridorState,
 CorridorError.ValueInABadPosition, "TargetHA",
"BasicLaneTransition")
                       dOffsetToTargetHA = vWidth +
oOrigin.Offset
                    End If
                End If
        End Select
       '---------------------------------------------------
        ' Create the subassembly points
        Dim corridorPoints As PointCollection
        corridorPoints = corridorState.Points
        Dim dX As Double
        Dim dy As Double
        dX = 0.0#
        dy = 0.0#
        Dim oPoint1 As Point
        oPoint1 = corridorPoints.Add(dX, dy, "")
        ' compute outside position of lane
        Select Case vTransitionType
            Case TransitionTypes.kHoldOffsetAndElevation
                ' hold original position (always used in
layout mode)
                dX = vWidth
                dy = Abs(vWidth) * vSlope
           Case TransitionTypes.kHoldElevationChangeOffset
                ' hold original elevation, move offset to
 that of TargetHA
                'dX = Abs(dOffsetToTargetHA -
corridorState.CurrentSubassemblyOffset)
              dX = Abs(dOffsetToTargetHA - oOrigin.Offset)
                dy = Abs(vWidth) * vSlope
            Case TransitionTypes.kHoldGradeChangeOffset
                ' hold original grade, move offset to that
 of TargetHA
                ' (also used if TargetVA is not defined)
                'dX = Abs(dOffsetToTargetHA -
corridorState.CurrentSubassemblyOffset)
              dX = Abs(dOffsetToTargetHA - oOrigin.Offset)
                dy = Abs(dX) * vSlope
           Case TransitionTypes.kHoldOffsetChangeElevation
              ' hold original offset, but change elevation
 to that of TargetVA
                dX = vWidth
                'dY = dOffsetElev -
corridorState.CurrentSubassemblyElevation
                dy = dOffsetElev - oOrigin.Elevation
            Case TransitionTypes.kChangeOffsetAndElevation
                ' move position to that of TargetHA, and
elevation to that of TargetVA
              dX = Abs(dOffsetToTargetHA - oOrigin.Offset)
                dy = dOffsetElev - oOrigin.Elevation
        End Select
      '----------------------------------------------------
        Dim dActualWidth As Double
        dActualWidth = dX
        Dim dActualSlope As Double
        If 0 = dActualWidth Then
            dActualSlope = 0.0#
        Else
            dActualSlope = dy / Abs(dActualWidth)
        End If
      '----------------------------------------------------
        Dim oPoint2 As Point
        oPoint2 = corridorPoints.Add(dX * dFlip, dy, "")
        dX = dX - 0.001
        dy = dy - vDepth
        Dim oPoint3 As Point
        oPoint3 = corridorPoints.Add(dX * dFlip, dy, "")
        dX = 0.0#
        dy = -vDepth
        Dim oPoint4 As Point
        oPoint4 = corridorPoints.Add(dX, dy, "")
        If vInsertionPoint = InsertionPoint.kCrown Then
            Utilities.AddCodeToPoint(1, corridorPoints,
oPoint1.Index, sPointCodeArray)
            Utilities.AddCodeToPoint(2, corridorPoints,
oPoint2.Index, sPointCodeArray)
            Utilities.AddCodeToPoint(3, corridorPoints,
oPoint3.Index, sPointCodeArray)
            Utilities.AddCodeToPoint(4, corridorPoints,
oPoint4.Index, sPointCodeArray)
        Else
            Utilities.AddCodeToPoint(2, corridorPoints,
oPoint1.Index, sPointCodeArray)
            Utilities.AddCodeToPoint(1, corridorPoints,
oPoint2.Index, sPointCodeArray)
            Utilities.AddCodeToPoint(4, corridorPoints,
oPoint3.Index, sPointCodeArray)
            Utilities.AddCodeToPoint(3, corridorPoints,
oPoint4.Index, sPointCodeArray)
        End If
       '----------------------------------------------------
        ' Create the subassembly links
        Dim oCorridorLinks As LinkCollection
        oCorridorLinks = corridorState.Links
        Dim oPoint(1) As Point
        Dim oLink(3) As Link
        oPoint(0) = oPoint1
        oPoint(1) = oPoint2
        oLink(0) = oCorridorLinks.Add(oPoint, "") 'L1
        oPoint(0) = oPoint2
        oPoint(1) = oPoint3
        oLink(1) = oCorridorLinks.Add(oPoint, "") 'L2
        oPoint(0) = oPoint3
        oPoint(1) = oPoint4
        oLink(2) = oCorridorLinks.Add(oPoint, "") 'L3
        oPoint(0) = oPoint4
        oPoint(1) = oPoint1
        oLink(3) = oCorridorLinks.Add(oPoint, "") 'L4
        Utilities.AddCodeToLink(1, oCorridorLinks,
oLink(0).Index, sLinkCodeArray)
        Utilities.AddCodeToLink(2, oCorridorLinks,
oLink(2).Index, sLinkCodeArray)
       '----------------------------------------------------
```

```
        ' Create the subassembly shapes
        Dim corridorShapes As ShapeCollection
        corridorShapes = corridorState.Shapes
        corridorShapes.Add(oLink, sShapeCodeArray(1))
      '-------------------------------------------------------
      '-------------------------------------------------------
       ' Write back all the Calculated values of the input
 parameters into the RoadwayState object.
        ' Because they may be different from the default
design values,
        ' we should write them back to make sure that the
 RoadwayState object
       ' contains the Actual information of the parameters.
        oParamsLong.Add(Utilities.Side, vSide)
       oParamsLong.Add("InsertionPoint", vInsertionPoint)
      oParamsLong.Add("CrownPtOnInside", vCrownPtOnInside)
       oParamsDouble.Add("Width", Math.Abs(dActualWidth))
        oParamsDouble.Add("Depth", vDepth)
        oParamsDouble.Add("Slope", dActualSlope)
       oParamsLong.Add("TransitionType", vTransitionType)
    End Sub
    Protected Sub FillCodesFromTable(ByVal
sPointCodeArray(,) As String, ByVal sLinkCodeArray(,) As
String, ByVal sShapeCodeArray() As String, ByVal
CrownPtOnInside As Long)
        If CrownPtOnInside = Utilities.ITrue Then
            sPointCodeArray(1, 0) = Codes.Crown.Code
        Else
            sPointCodeArray(1, 0) = ""
        End If
        sPointCodeArray(2, 0) = Codes.ETW.Code
        sPointCodeArray(3, 0) = Codes.ETWSubBase.Code 'P4
        If CrownPtOnInside = Utilities.ITrue Then
           sPointCodeArray(4, 0) = Codes.CrownSubBase.Code
 'P3
        Else
            sPointCodeArray(4, 0) = "" 'P3
        End If
        sLinkCodeArray(1, 0) = Codes.Top.Code
        sLinkCodeArray(1, 1) = Codes.Pave.Code
        sLinkCodeArray(2, 0) = Codes.Datum.Code
        sLinkCodeArray(2, 1) = Codes.SubBase.Code
        sShapeCodeArray(1) = Codes.Pave1.Code
    End Sub
End Class
```

The Subassembly Tool Catalog

Overview

An Autodesk tool catalog organizes groups of customized subassemblies and makes them available to AutoCAD Civil 3D users. Autodesk tool catalogs are defined using xml-formatted files with an *.atc* (Autodesk Tool Catalog) extension. You also need to create a catalog registry file as catalogs must be registered in the Windows registry. Some items within the *.atc* and registry files must contain unique identifiers known as GUIDs (Global Unique Identifiers). New GUIDs can be created using the *GuidGen.exe* utility that is included with many Microsoft development products and is freely available for download from the Microsoft web site.

To create a tool catalog for a subassembly

1. Using Notepad or any other appropriate editor, create a plain ASCII text file named *<Name>Tools Catalog.atc,* where *<Name>* is the name of this new tool catalog. For information about the contents of the file, see Creating a Tool Catalog ATC File (page 175).

   NOTE

   XML tags in ATC files are case-sensitive. Be sure that your tags match the case of the tags described in this chapter.

2. Save the *.atc* file to the location where your tool catalogs are stored. The default location is .
3. Create any optional files, such as images for icons displayed in the catalog and help files for subassemblies, and place these files in appropriate locations for reference.
4. Using Notepad or any other text editor, create a registry file for the tool catalog with the extension *.reg*. For more information, see Creating a Tool Catalog Registry File (page 186).
5. Register the tool catalog by double-clicking on the *.reg* file. Once registered, the subassembly tool catalog will be displayed in the AutoCAD Civil 3D Content Browser.

Creating a Tool Catalog ATC File

Sample Tool Catalog ATC Files

The sample tool catalog *.atc* files define the contents and organization of an Autodesk subassembly tool catalog. These are generally split into separate files to keep files manageable: one listing all categories of tools and others listing tools within a single category.

NOTE

XML tags in ATC files are case-sensitive. Make sure the tags in your files match the case of the tags defined in this chapter.

Main Catalog File Example

The following is a portion of the file *"Autodesk Civil 3D Metric Corridor Catalog.atc."* It contains a list of tool categories. See the table following the sample for descriptions of the contents. This excerpt only contains the "Getting Started" category of tools.

```
1)  <Catalog option="0">
2)     <ItemID
idValue="{0D75EF58-D86B-44DF-B39E-CE39E96077EC}"/>
3)     <Properties>
4)        <ItemName resource="9250"
src="AeccStockSubassemblyScriptsRC.dll"/>
5)        <Images option="0">
6)           <Image cx="93" cy="123"
src=".\Images\AeccCorridorModel.png"/>
7)        </Images>
8)        <Description resource="9201"
src="AeccStockSubassemblyScriptsRC.dll"/>
9)        <AccessRight>1</AccessRight>
10)       <Time
createdUniversalDateTime="2003-01-22T00:31:56"
modifiedUniversalDateTime="2006-09-04T13:28:12"/>
11)    </Properties>
12)    <Source>
13)       <Publisher>
14)          <PublisherName>Autodesk</PublisherName>
```

```
15)       </Publisher>
16)    </Source>
17)    <Tools/>
18)    <Palettes/>
19)    <Packages/>
20)    <Categories>
21)       <Category option="0">
22)          <ItemID
idValue="{4F5BFBF8-11E8-4479-99E0-4AA69B1DC292}"/>
23)          <Url href=".\\C3D Metric Getting Started
Subassembly Catalog.atc"/>
24)          <Properties>
25)             <ItemName resource="9212"
src="AeccStockSubassemblyScriptsRC.dll"/>
26)             <Images option="0">
27)                <Image cx="93" cy="123"
src=".\Images\AeccGenericSubassemblies.png"/>
28)             </Images>
29)             <Description resource="9213"
src="AeccStockSubassemblyScriptsRC.dll"/>
30)             <AccessRight>1</AccessRight>
31)          </Properties>
32)          <Source/>
33)       </Category>
34)
35) <!-- Other category items omitted -->
36)
37)    </Categories>
38)    <StockTools/>
39)    <Catalogs/>
40) </Catalog>
```

Line Number	Description
1-40	The <Catalog> section contains the entire contents of the catalog file.
2	This <ItemID> defines the GUID for the catalog. The same GUID must be used in the registry file to identify this catalog.
3-11	This section defines general properties of the catalog.

Line Number	Description
4	<ItemName> defines the name that appears beneath the catalog icon in the catalog browser. In this case, we use a string resource to support localization. You can also specify a constant string by using the line "<ItemName>*Name*</ItemName>".
5-7	<Images> defines the image file for the icon that appears for the catalog in the catalog browser. Images used for catalogs and sub-assemblies should be a 64-by-64 pixel image. Valid image file types include *.bmp*, *.gif*, *.jpg*, and *.png*.
8	<Description> contains the string description for the catalog. In this case, we use a string resource to support localization. You can also specify a constant string by using the line "<Description>*String*</Description>".
10	<Time> defines the time and date the catalog was created in the universal date/time format. This information is required, but not used. Any date or time may be given.
12-16	<Source> defines the source or creator of the catalog.
17-19	Empty definitions for Tools, Palettes, and Packages.
20-36	The <Categories> group defines a list of categories, each of which may contain sub-categories or subassembly tools.
21-33	Definition of the "Getting Started" category. This block has been designed to load all category information from a separate file. Category information can also be placed within this file if you only want one *.atc* file by using a <Category> section as described in the Tool File Example (page 178)
22	<ItemID> defines the unique GUID for this category. It must be the same GUID as in the separate category *.atc* file.
23	<Url> specifies the location of the category *.atc* file.

Line Number	Description
24-31	The properties of a category.
25	<ItemName> defines the name of this category of tools. In this case, we use a string resource to support localization. You can also specify a constant string by using the line "<ItemName>*Name*</Item-Name>".
26-28	Sets the image used to identify the category to users.
29	<Description> contains the string description for the category. In this case, we use a string resource to support localization. You can also specify a constant string by using the line "<Descrip-tion>*String*</Description>".
32	Empty definition for Source.
39	Empty definition for StockTools

Tool File Example

The following is a portion of the "*Autodesk Metric Getting Started Subassembly Catalog.atc*" file. It contains all the tools in the "Getting Started" catalog. This excerpt only contains the "BasicBarrier" tool.

```
1)  <Category>
2)     <ItemID
idValue="{4F5BFBF8-11E8-4479-99E0-4AA69B1DC292}"/>
3)     <Properties>
4)       <ItemName src="AeccStockSubassemblyScriptsRC.dll"
 resource="9212"/>
5)        <Images>
6)           <Image cx="93" cy="123"
src=".\Images\AeccGenericSubassemblies.png"/>
7)        </Images>
8)             <Description
src="AeccStockSubassemblyScriptsRC.dll" resource="9213"/>
9)        <AccessRight>1</AccessRight>
10)       <Time
```

```
createdUniversalDateTime="2002-09-16T14:23:31"
modifiedUniversalDateTime="2004-06-17T07:08:09"/>
11)    </Properties>
12)    <CustomData/>
13)    <Source/>
14)    <Palettes/>
15)    <Packages/>
16) <Tools>
17) <Tool>
18)    <ItemID
idValue="{F6F066F4-ABF2-4838-B007-17DFDDE2C869}"/>
19)    <Properties>
20)            <ItemName resource="101"
src="AeccStockSubassemblyScriptsRC.dll"/>
21)       <Images>
22)          <Image cx="64" cy="64"
src=".\Images\AeccBasicBarrier.png"/>
23)       </Images>
24)       <Description resource="102"
src="AeccStockSubassemblyScriptsRC.dll"/>
25)       <Keywords>_barrier subassembly</Keywords>
26)       <Help>
27)       <HelpFile>.\Help\C3DStockSubassemblyHelp.chm</HelpFile>
28)          <HelpCommand>HELP_HHWND_TOPIC</HelpCommand>
29)          <HelpData>SA_BasicBarrier.htm</HelpData>
30)       </Help>
31)       <Time
createdUniversalDateTime="2002-04-05T21:58:00"
modifiedUniversalDateTime="2002-04-05T21:58:00"/>
32)    </Properties>
33)    <Source/>
34)    <StockToolRef
idValue="{7F55AAC0-0256-48D7-BFA5-914702663FDE}"/>
35)    <Data>
36)       <AeccDbSubassembly>
37)       <GeometryGenerateMode>UseDotNet</GeometryGenerateMode>
38)          <DotNetClass
Assembly=".\C3DStockSubassemblies.dll">Subassembly.BasicBarrier</DotNetClass>
39)          <Resource
Module="AeccStockSubassemblyScriptsRC.dll"/>
40)          <Content
DownloadLocation="http://www.autodesk.com/subscriptionlogin"/>
41)          <Params>
```

```
42)             <Version DataType="String"
DisplayName="Version" Description="Version">R2007</Version>
43)             <TopWidth     DataType="Double"
TypeInfo="16" DisplayName="105"
Description="106">0.15</TopWidth>
44)             <MiddleWidth  DataType="Double"
TypeInfo="16" DisplayName="107"
Description="108">0.225</MiddleWidth>
45)             <CurbWidth    DataType="Double"
TypeInfo="16" DisplayName="109"
Description="110">0.57</CurbWidth>
46)             <BottomWidth  DataType="Double"
TypeInfo="16" DisplayName="111"
Description="112">0.6</BottomWidth>
47)             <TopHeight    DataType="Double"
TypeInfo="16" DisplayName="113"
Description="114">0.9</TopHeight>
48)             <MiddleHeight DataType="Double"
TypeInfo="16" DisplayName="115"
Description="116">0.45</MiddleHeight>
49)             <CurbHeight   DataType="Double"
TypeInfo="16" DisplayName="117"
Description="118">0.075</CurbHeight>
50)          </Params>
51)       </AeccDbSubassembly>
52)       <Units>m</Units>
53)    </Data>
54) </Tool>
55)
56) <!-- Other tool items omitted -->
57)
58) </Tools>
59) <StockTools/>
60) </Category>
```

Line Number	Description
1-59	A <Category> is a list of subassemblies.
2	<ItemID> defines the unique GUID for this category. It must be the same GUID as in the parent catalog *.atc* file.

Line Number	Description
3-11	The properties of a category.
4	<ItemName> defines the name of this category of tools. In this case, we use a string resource to support localization. You can also specify a constant string by using the line "<ItemName>*Name*</ItemName>".
5-7	Sets the image used to identify the category to users.
8	<Description> contains the string description for the category. In this case, we use a string resource to support localization. You can also specify a constant string by using the line "<Description>*String*</Description>".
12-15	Empty definitions for Custom Data, Source, Palettes, Packages.
16-57	<Tools> contains all the separate subassemblies.
17-53	<Tool> represents a single subassembly.
18	<ItemID> defines the unique GUID for this subassembly.
19-32	The properties of the subassembly.
20	<ItemName> defines the name of this subassembly. In this case, we use a string resource to support localization. You can also specify a constant string by using the line "<ItemName>*Name*</ItemName>".
21-23	Sets the image used to identify the subassembly to users.
24	<Description> defines the name of this subassembly. In this case, we use a string resource to support localization. You can also specify a constant string by using the line "<Description>*String*</Description>".
25	Keywords describing the subassembly.

Line Number	Description
27	<HelpFile> defines the filename and path of the help file.
28	<HelpCommand> defines the command used to display the help file.
29	<HelpData> is the particular topic in the help file to display.
31	<Time> defines the time and date the catalog was created in the universal date/time format. This information is required, but not used. Any date or time may be given.
33	Empty Source tag.
34	<StockToolRef> defines a GUID specifically for catalog tools. This must use the idVlaue of {7F55AAC0-0256-48D7-BFA5-914702663FDE}
35-52	Describes the nature of the subassembly.
36-50	Identifies the tool as a subassembly.
37	<GeometryGenerateMode> is a new tag that describes the source code of the subassembly. It can either have a value of “UseDotNet” or “UseVBA”. If this tag is not used, “UseVBA” is assumed.
38	<DotNetClass> is a new tag that lists the .NET assembly and class which contains the subassembly. The VBA equivalent is the <Macro> tag. Note that all paths specified in the ATC file must be relative paths to the ATC file itself.
39	The *.dll* containing the resource strings and images used by the subassembly tool.
40	<Content> is a new tag that specifies a location where you can download the subassembly if it isn’t located on the local machine. If the subassembly hasn’t been downloaded, the location contained

Line Number	Description
	in the `DownloadLocation` attribute is displayed in the event viewer.
41-50	<Params> defines the names of the input parameters associated with the subassembly tool. This list appears in the Properties page of the subassembly, in the order they appear in the *.atc* file. Each parameter is defined on a single line.
42-49	Each parameter is described with the following: Parameter name - The internal name of the parameter (e.g., "CrownHeight"). This is the name that must be used when saving or retrieving parameters to the parameter buckets. DataType – Defines the type of variable used to store the parameter value, such as Long, Double, or String. For more information, see Tool Catalog Data Type Information (page 183). DisplayName – Defines the name that is displayed for the parameter in the subassembly Properties page. This is what the user sees to identify each parameter. Description – Provides a description of the input parameter. When a parameter name is highlighted in the subassembly Properties page, the description appears at the bottom of the page. value – The default value for the parameter. This is the value that appears for the parameter in the subassembly Properties page.
52	<Units> describes the type of units the subassembly expects. Valid values are "m" for meters of 'foot" for feet.

Tool Catalog Data Type Information

The following tables describe the data types that can be used to define the variable that stores parameter values in the corridor modeling tool catalog *.atc* file.

Boolean Data Types

Data Type	Type String	Description
Bool	0	0 = True; 1 = False.

Data Type	Type String	Description
BoolNoYes	1	0 = Yes; 1 = No.
BoolDisabledEnabled	5	0 = Enabled; 1 = Disabled.
BoolOffOn	6	0 = On; 1 = Off.
BoolRightLeft	7	0 = Right; 1 = Left.

Long Data Types

Data Type	Type String	Description
Long	0	Any integer
NonZeroLong	1	Any non-zero integer
NonNegativeLong	2	Zero or any positive integer
NonNegativeNonZeroLong	3	Any non-zero positive integer
NonPositiveLong	4	Zero or any negative integer
NonPositiveNonZeroLong	5	Any non-zero negative integer

Double Data Types

Data Types	Type String	Description
Double	0	Any double value
NonNegativeDouble	1	Zero or any positive double value
NonNegativeNonZeroDouble	2	Any non-zero positive double value
NonPositiveDouble	3	Zero or any negative double value

Data Types	Type String	Description
NonPositiveNonZeroDouble	4	Any non-zero negative double value
NonZeroDouble	5	Any non-zero double value
Grade	8	Slope or grade input values
TransparentCmdGrade	9	Grade input values
TransparentCmdSlope	10	Slope input values
Angle	14	Angular values
ConvergenceAngle	15	Convergence angular value
Distance	16	Distance values in feet or meters
Dimension	17	Dimension values in inches or millimeters
Elevation	21	Elevation values
Percent	25	Percent values

Creating a Tool Catalog Cover Page

Use an *.html* file to create introductory content that will display when a user clicks on this new catalog in the AutoCAD Civil 3D Content Browser. You can use any html editor, and the cover page can be as simple or as comprehensive as you like. Typically, the cover page gives an overview of the tools supplied in the catalog, and a brief description of how they may be used.

By convention, cover pages are named *<Name> - ToolCatalogCoverPage.html* where "<Name>" is the name of the new tool catalog. The location of the

.html file is specified within the *.atc* file. This file is usually placed in the same directory as the *.atc* files themselves.

Creating a Tool Catalog Registry File

Each subassembly tool catalog needs to be registered in the Windows registry before it can be used by AutoCAD Civil 3D. One way to do this is to create a *.reg* file, which is a text file containing the new keys, value names, and values to be added to the registry. Double-clicking on the *.reg* file will modify the registry. After this process, the *.reg* file is no longer needed.

The following is the contents of a *.reg* file that registers the *Autodesk Civil 3D Imperial Corridor Catalog.atc* catalog file. See the table following the sample for descriptions of the contents.

```
1)  REGEDIT4
2)
3)  [HKEY_CURRENT_USER\Software\Autodesk\Autodesk Content
 Browser\60]
4)
5)  [HKEY_CURRENT_USER\Software\Autodesk\Autodesk Content
 Browser\60\RegisteredGroups]
6)
7)  [HKEY_CURRENT_USER\Software\Autodesk\Autodesk Content
 Browser\60\RegisteredGroups\Roads Group]
8)  "ItemID"="{5BD79109-BC69-41eb-9AC8-7E9CD469C8D3}"
9)  "ItemName"="Roads Group"
10)
11)
12)  [HKEY_CURRENT_USER\Software\Autodesk\Autodesk Content
 Browser\60\RegisteredCatalogs]
13)
14)  [HKEY_CURRENT_USER\Software\Autodesk\Autodesk Content
 Browser\60\RegisteredCatalogs\Autodesk Civil 3D Imperial
 Corridor Catalog]
15)  "ItemID"="{410D0B43-19B3-402f-AB41-05A6E174AA3F}"
16)  "Image"=".\\Images\\AeccRoadway.png"
17)  "Url"="C:\\Documents and Settings\\All
Users\\Application Data\\Autodesk\\C3D2010\\enu\\Tool
Catalogs\\Road Catalog\\Autodesk Civil 3D Imperial Corridor
 Catalog.atc"
18)  "DisplayName"="Civil 3D Subassemblies (Imperial Units)"
```

```
19)  "Description"="Imperial Units Subassemblies"
20)  "Publisher"="Autodesk"
21) "ToolTip"="Autodesk Civil 3D Imperial Corridor Catalog"
22)  "GroupType"="{5BD79109-BC69-41eb-9AC8-7E9CD469C8D3}"
23)
24)
25) [HKEY_CURRENT_USER\Software\Autodesk\AutoCAD\R18\ACAD-8000:409\AEC\60\General\Tools]
26)  "ToolContentRoot"="C:\\Documents and Settings\\All
Users\\Application Data\\Autodesk\\C3D2010\\enu\\Tool
Catalogs\\Road Catalog"
```

Line Number	Description
1	Identifies the file as a registry edit file.
3-9	These statements create a Group for the Autodesk Content Browser. The group id name is “Roads Group”. Each group must have a unique GUID for the “ItemID”. The Roads Group is already registered by the AutoCAD Civil 3D installation. If you are adding a catalog to this group, you should use the GUID shown in the example.
12	Identifies the item being registered as an Autodesk catalog for the Autodesk Content Browser.
14-22	These statements define the Catalog Entry.
15	“ItemId” must be a unique GUID for this catalog. This must match the GUID for the Catalog ItemID value in the catalog *.atc* file.
16	“Image” must be a unique GUID for this catalog. This must match the GUID for the Catalog ItemID value in the catalog *.atc* file.
17	“URL” is a pointer to the catalog *.atc* file that is being registered.
18	“DisplayName” is the text that displays beneath the catalog icon in the Autodesk Content Browser.
19	“Description” – description of the tool catalog.
20	“Publisher” – name of the creator / publisher of the tool catalog.

Line Number	Description
21	"ToolTip" – the text that displays for the tooltip when the cursor is hovered over the tool catalog in the Catalog Browser.
22	"GroupType" – the GUID that defines which the tool catalog belongs to in the Catalog Browser. This GUID must match the one used for the "ItemID" in the group definition.
26	The directory where the .atc catalog files are located.

Installing Custom Subassemblies

Once you've created a custom subassembly, you can install it on other AutoCAD Civil 3D users' machines.

NOTE

It's simpler to create a subassembly package file to distribute to users than to install custom subassemblies manually. See Exporting Subassemblies Using a Package File (page 189).

To install a custom subassembly:

1. Copy the compiled AutoCAD Civil 3D subassembly *.dll* library to its destination directory. By default, libraries are located in *<AutoCAD Civil 3D Install Directory>\Sample\Civil 3D API\C3DstockSubAssemblies*.
2. Copy the tool catalog *.atc* files to its destination directory. The tool catalog files are normally located in the directory. For information about creating these, see Creating a Tool Catalog ATC File (page 175).
3. Copy optional files such as the image file representing the subassemblies or the help file to their destination directory. Images are normally located in , and help files are normally located in , although these can be any directory as long as the *.atc* file has the correct relative path information. For information about creating help files, see Creating Subassembly Help Files (page 152)
4. Copy the catalog cover page *.html* file to its destination. Usually this is the same location as the *.atc* file, although it can be any directory as long as the *.atc* file has the correct relative path information. For information

about creating cover pages, see Creating a Tool Catalog Cover Page (page 185).

5 Register the tool catalog using a registry (*.reg*) file. This *.reg* file must have the correct paths to the *.atc* file and the catalog image file from steps 2) and 3). For information about creating registry files, see Creating a Tool Catalog Registry File (page 186)

Exporting Custom Subassemblies Using a Package File

You can share custom subassemblies with others by copying the required files to a package file.

Custom subassemblies that have been created using .NET or VBA can be exported and imported using a package file. A package file contains all the files necessary for the custom subassemblies to work. Once the package file is created, users can import the package file into AutoCAD Civil 3D, and then copy the custom subassemblies directly into a tool palette or catalog. For more information about importing subassembly package files, see Sharing Subassemblies in the .

Like other types of AutoCAD Civil 3D content, subassemblies can also be shared with others through the Autodesk Civil Engineering Community site. Go to *http://civilcommunity.autodesk.com/* to access the Autodesk Civil Engineering web site, then click on Content Sharing.

To create a package file, you must copy all of the files that make up the custom subassembly or subassemblies into a folder. Create a *.zip* file of that folder, and then change the file extension from *.zip* to *.pkt*.

NOTE

Subassemblies created from polylines cannot be included in a package file. Package files are intended for sharing custom subassemblies that have been created using .NET or VBA.

Naming the Package File

If you are sharing a single subassembly, it is recommended that you name the package file the same name as the subassembly. For example, if you plan to export a subassembly named OpenChannel, name the package file *OpenChannel.pkt*. If you plan to export multiple subassemblies in a single package file, give the folder a name that easily identifies the types of subassemblies that are contained in it. For example, *DitchSubassemblies.pkt*.

Required Subassembly Files

The following table describes the files that must be included in a package file to successfully export and import one or more subassemblies.

File	Description
.atc file(s)	A valid *.atc* file that defines the shape and behavior of the subassembly or subassemblies is required. You can have one or more *.atc* files included in the package file. For example, you can have one *.atc* file that defines one or more subassemblies, or you can have multiple *.atc* files, each of which defines one or more subassemblies. If the package file contains multiple *.atc* files, each *.atc* file must have a unique name. All the paths referenced in .atc file must be relative paths, if they point to files in the same .pkt file.
.dll or *.dvb* file(s)	A *.dll* file is required for subassemblies that are defined using .NET. A *.dvb* file is required for subassemblies that are defined using VBA. A package file can contain both *.dll* and *.dvb* files.
Help file(s)	A Help file is not required in order for a subassembly to function properly. However, the Help file is needed for others to understand how to use the subassembly. Therefore, it is recommended that you always include a Help file with each subassembly. The Help file for each subassembly is specified in the *.atc* file, and can be any of the following formats: *.dwf*, *.doc*, *.pdf*, *.txt*, *.chm*, *.hlp*. For more information, see Creating Subassembly Help Files (page 152).
Image file(s)	An image file for each is displayed on the tool palette and is used to provide a conceptual graphical representation of the subassembly shape.

Exporting Custom Subassemblies Using a Package File

To create a package file

1 Copy all of the required subassembly files into a folder. Make sure that folder contains only the files required for the subassembly or subassemblies you plan to export.

2 Create a *.zip* file of the contents of that folder.

3 Change the file extension from *.zip* to *.pkt*.

The package (*.pkt*) file is created and can be shared with other users.

Converting VBA Subassemblies to .NET

This chapter describes a process to convert existing custom subassemblies written in COM/VBA to .NET.

Although VBA custom subassemblies are still supported in AutoCAD Civil 3D 2013, VBA support is deprecated and will be discontinued in a future release. In addition, .NET subassemblies offer several advantages: they are easier to write and maintain, and most importantly, they offer significant performance improvements. On average, they perform about 50% faster than the equivalent code in VBA, and the performance increase can be even higher in complex drawings.

Once you have converted your custom subassemblies to .NET and imported them into the catalog file, you will need to update the dependent assemblies in your drawings, and re-build the corridors. See the last section in this Appendix for some sample code that replaces old VBA subassemblies with new .NET subassemblies.

Procedure

This section describes the steps required to convert a VBA subassembly to VB.NET. Although this procedure only describes converting the subassembly to VB.NET, you can use a similar approach to convert the subassembly to another .NET language, such as C#.

Where applicable, a regular expression is suggested to automate some of the changes required. In most cases, using regular expressions can reduce the time required to port a custom subassembly. See the MSDN help topic on *Regular Expressions* and *Replace in Files* for more information.

Create the Visual Basic.NET Subassembly Module

Start by creating the module for your subassembly. You can add a new module to the `C3DStockSubassemblies` project, or you can create your own .NET subassembly project and add the new class module there. New projects should use the Visual Studio "Class Library" template, and should reference the following files:

- `acdbmgd.dll`
- `acmgd.dll`
- `AecBaseMgd.dll`
- `AeccDbMgd.dll`

Be sure to include *CodeSpecific.vb*, *SATemplate.vb*, and *Utilties.vb* in your project as well.

Add the following framework to the Visual Basic subassembly class module:

```
Public Class UserDefinedSA
Inherits SATemplate
    ' Member Variables.
    Protected Overrides Sub GetLogicalNamesImplement(ByVal
 corridorState As Autodesk.Civil.Runtime.CorridorState)
        ' Todo
    End Sub
    Protected Overrides Sub
GetInputParametersImplement(ByVal corridorState As
CorridorState)
        ' Todo
    End Sub
    Protected Overrides Sub
GetOutputParametersImplement(ByVal corridorState As
CorridorState)
        ' Todo
    End Sub
    Protected Overrides Sub DrawImplement(ByVal
corridorState As CorridorState)
         ' Todo
    End Sub
End Class
```

Note that:

- The class must inherit from `SATemplate`.
- You must override the `DrawImplement()` method, otherwise the subassembly will do nothing.
- The other overridden methods may be removed if they are not used.
- You can add your own functions and subroutines within the class's scope.

Copy Subassembly Code

Copy the original code from the VBA module (*.bas) to the corresponding place in the new class:

From	To
`UserdefinedSA_GetLogicalNames()`	`GetLogicalnamesImplement()`
`UserdefinedSA _GetInputParamet-ers()`	`GetInputParametersImplement()`
`UserdefinedSA _GetOutputParamet-ers()`	`GetOutputParametersImplement()`
`UserdefinedSA()`	`DrawImplement()`
Const variables	Member variables section

Port the VBA Code to Visual Basic .NET Code

Now begin the porting work. The following sections outline the main steps in porting. Code from the stock subassembly *DaylightBench.vb* is used as an illustration.

Step 1: Import the necessary namespaces

Adding the namespaces `Autodesk.Civil.Roadway` and `Autodesk.Civil` is recommended because members in these two namespaces are frequently used in subassemblies. Your subassembly may require additional namespaces. These are the namespaces imported in the stock Civil 3D subassemblies (from *C3DStockSubassemblies.vbproj*):

- Autodesk.Civil
- Autodesk.Civil.ApplicationServices
- Autodesk.Civil.DatabaseServices
- Autodesk.Civil.Land
- Autodesk.Civil.Land.DatabaseServices

- Autodesk.Civil.Land.Settings
- Autodesk.Civil.Roadway
- Autodesk.Civil.Roadway.DatabaseServices
- Autodesk.Civil.Roadway.Settings
- Autodesk.Civil.Runtime
- Autodesk.Civil.Settings

Step 2: Remove all `On Error`… statements

Remove the `On Error` statements in the `GetLogicalNamesImplement()`, `GetInputParametersImplement()`, and `GetOutputParametersImplement()` methods. Comment these statements out in `DrawImplement ()`, because you will re-use the code in Step 14.

Regular expression:

- Find: `On Error{.*}`
- Leave the Replace field blank if you want to delete these statements. Use `' On Error\1` if you only want to comment them out.

Step 3: Remove `Exit Sub` and `Error Handler`

Remove all the `Exit Sub` and `Error Handler:` statements at the end of each subroutine. In VBA, you may see following code at the end of each subroutine - remove it, or move it into the appropriate `Catch` statement when converting error handling logic in step 14.

```
Exit Sub
ErrorHandler:
RecordError Err.Number, Err.Description, Err.Source
```

Regular expression:

- Find: `Exit Sub{ *}\n{ *}\n{ *}ErrorHandler\:{ *}\n{ *}RecordError.+{ *}\n{ *}{End Sub}`
- Replace: `\8`

Step 4: Remove `oRwyState` definition.

Remove the following code at the beginning of each subroutine:

```
' Get the roadway state object
Dim oRwyState As AeccRoadwayState
Set oRwyState = GetRoadwayState()
```

The `corridorState` object is already passed in by the argument of the subroutine, so it is not necessary to get it yourself.

Regular expression:

- Find: `{'.*}{ *}\n{ +}Dim.+As AeccRoadwayState{ *}\n{ *}.+= GetRoadwayState\(\)`
- Replace: leave blank to delete these statements

Step 5: Replace `oRwyState` with `corridorState`

All instances of the variable `oRwyState` used in VBA should be renamed `corridorState` in .NET. The `corridorState` variable is passed in by argument.

Regular expression:

- Find: `oRwyState`
- Replace: `corridorState`

Step 6: Check for errors when accessing parameter values

In VBA, the parameter `Value()` method returns null if the key does not exist. In the .NET API, the same code will throw an exception. Where you access a parameter, you need to use a `Try / Catch` block to catch this case:

```
' VBA:
vCutSlope = oParamsDouble.Value("CutSlope")
If IsEmpty(vCutSlope) Then vCutSlope = c_dCutSlopeDefault
```

This code can be changed to:

```
' .NET:
Try
    vCutSlope = oParamsDouble.Value("CutSlope ")
Catch
    vCutSlope = c_ dCutSlopeDefault
End Try
```

Regular expression:

- Find: `^{.+ =.+\.Value\(.+\)}\n{ +}If IsEmpty.+Then{.+}`
- Replace: `Try\n\1\nCatch\n\3\nEnd Try\n`

Step 7: Update `RecordError()`

The global function `RecordError()` is replaced by `Utilities.RecordError()`.

```
' VBA:
RecordError(aeccRoadwayErrorValueTooLarge,
"RoundingTesselation", "DaylightBench")
```

Change to:

```
'.NET:
Utilities.RecordError(corridorState,
CorridorError.ValueTooLarge, "RoundingTesselation",
"DaylightBench")
```

Regular expression:

- Find: `{Record}{Warning|Error}{\(}{aecc}{RoadwayError}`
- Replace: `Utilities.Record\2\3roadwayState,RoadwayError.`

Step 8: Replace global variables with `Utilities` variables.

In VBA subassembly code, you will see global variables such as `g_iRight`, `g_sSide`, which have a "g_" prefix. Most of these global variables have been moved to the `Utilities` class and renamed.

The following table lists some commonly used global variables and their corresponding ones in the `Utilities` class. Refer to the definition of the `Utilities` class for more information.

From	To
`g_sSide`	`Utilities.Side`
`g_iLeft`	`Utilities.Left`

From	To
g_iRight	Utilities.Right
g_iTrue	Utilities.ITrue
Rounding_Option.NoneType	Utilities.RoundingOption.None-Type
CutSituation	Utilities.FillOrCut.FillSitu-ation
SubbaseType	Utilities.ShoulderSubbase-Type.Subbase

Step 9: Rename enumerations

In VBA subassembly code, nearly all the COM enumerations have an "aecc" prefix, such as `aeccParamLogicalNameTypeAlignment`. Replace them with corresponding .NET enumerations.

As a rule, the corresponding .NET enumerations are named by removing the "aecc" prefix and making the detail category a child member. For example, `aeccParamLogicalNameTypeAlignment` becomes `ParamLogicalNameType.Alignment`.

The following table lists some commonly used COM enumerations and their corresponding .NET enumerations.

From	To
aeccParamLogicalNameTypeAlign-ment	ParamLogicalNameType.Alignment
aeccRoadwayModeLayout	CorridorMode.Layout
aeccParamAccessOutput	ParamAccessType.Output

Step 10: Rename types

Replace COM types with their corresponding .NET types.

The following table lists some commonly used COM types and their corresponding .NET types.

From	To
`IAeccParamsDouble`	`ParamDoubleCollection`
`IAeccRoadwayLinks`	`LinkCollection`
`IAeccParam`	`Param`
`AeccRoadwayLink`	`Link`
`AeccParamLong`	`ParamLong`

Regular expressions:

1 **Convert `IAeccParamsAlignment` and `AeccParamsAlignment` to `ParamAlignmentCollection`**

- Find: `{ }I*AeccParams{.+}{ |\n}`
- Replace: `\1Param\2Collection\3`

2 **Convert: `IAeccRoadwayLinks` to `LinkCollection`**

- Find: `{ }I*AeccRoadway{.*}s{ |\n}`
- Replace: `\1\2Collection\3`

3 **Convert `AeccRoadwayLink` to `Link`**

- Find: `{ }(I|())AeccRoadway{Link|Shape}{ |\n}`
- Replace: `\1\2\3`

4 **Convert `AeccParamLong` to `ParamLong`**

- Find: `{ }I*AeccParam{Long|Bool|Double|Point|String|()}{ |\n}`
- Replace: `\1Param\2\3`

These four regular expressions do not cover all the required changes, so you will have to make additional changes manually.

Step 11: Change `Object` to build-in types

Where the subassembly uses `Object`, it should be changed to a built-in type, as `Object` is not type-safe.

```
' .NET Before:
Dim vSide As Object
Try
      vSide = oParamsLong.Value(Utilities.Side)
Catch
      vSide = SideDefault
End Try
```

```
' .NET After:
Dim vSide As Long
Try
      vSide = oParamsLong.Value(Utilities.Side)
Catch
      vSide = SideDefault
End Try
```

Step 12: Update code variables to use .NET naming

Remove the `g_All`, `g_s` and `s` prefixes in each variable name to match the new convention for code names.

From	To
`g_AllCodes.g_sHinge_Cut.sCode`	`Codes.HingeCut.Code`
`g_AllCodes.g_sDaylight.sCode`	`Codes.Daylight.Code`
`g_AllCodes.g_sDaylight_Cut.sCode`	`Codes.DaylightCut.Code`

Regular expression:

- Find: `g_All{Codes\.}g_s{.+\.}s{.+}`
- Replace: `\1\2\3`

Step 13: Update array definitions

In VBA, many arrays are defined with a lower bound at index “1”. This is not allowed in .NET, so you should modify their definitions.

In most situations, you can just modify the array definition do not need to make any other changes. The array element 0 is defined but left unused.

```
' .NET Before:
Dim sPointCodeArray(1 To 9, 0 To 1) As String
```

```
' .NET After:
Dim sPointCodeArray(0 To 9, 0 To 1) As String
```

If this array is passed as an argument to a method, you will need to do more work. In this case, the array is most likely used from index “0”. You need to modify all the code that uses this array to take into account the change in index numbering.

Step 14: Modify error handling

In VBA, `On Error Resume Next` and `On Error GoTo ErrorHandler` statements are used to handle errors. But in .NET, exceptions are used instead. Modify all cases where errors are detected to use `Try…Catch` blocks.

```
' VBA:
On Error Resume Next
Dim oTargetDTM As IAeccSurface
Set oTargetDTM = oParamsSurface.Value("TargetDTM")
If oTargetDTM Is Nothing Then
    ' Error handling code goes here
    Exit Sub
End If
On Error GoTo ErrorHandler
```

Change error handling to:

```
' .NET:
Dim oTargetDTMId As ObjectId
Try
    oTargetDTMId = oParamsSurface.Value("TargetDTM")
Catch
      ' Error handling code goes here
    Exit Sub
End Try
```

Step 15: New rules for database objects

In the COM API, the arguments and return values all refer to a database object instance. However, in the .NET API, arguments and return values are the `ObjectId` of the database object, not the object itself.

```
' VBA:
Dim oTargetDTM As IAeccSurface
Set oTargetDTM = oParamsSurface.Value("TargetDTM")
```

In .NET this would be:

```
' .NET:
Dim oTargetDTMId As ObjectId
Try
   oTargetDTMId = oParamsSurface.Value("TargetDTM")
Catch
End Try
```

Then, to use the database object's instance, you have to use `TransactionManager.GetObject()` to open the database object.

```
' VBA:
Dim oCurrentAlignment As AeccAlignment
GetAlignmentAndOrigin(oRwyState, oCurrentAlignment, oOrigin)
```

In .NET:

```
' .NET:
Dim currentAlignmentId As ObjectId
Dim currentAlignment As Alignment
Utilities.GetAlignmentAndOrigin(corridorState,
currentAlignmentId, origin)
' ...
Dim db As Database =
Autodesk.AutoCAD.DatabaseServices.HostApplicationServices.WorkingDatabase
Dim tm As
Autodesk.AutoCAD.DatabaseServices.TransactionManager =
db.TransactionManager
```

```
currentAlignment = tm.GetObject(currentAlignmentId,
OpenMode.ForRead, false, false)
```

Final Adjustments

The above rules cover the majority of changes needed to convert VBA code to working Visual Basic .NET code. Depending on the nature of your subassembly, there may still be some syntax errors.

Installing the New Subassembly

The Autodesk Tool Catalog files (*.atc* files, found in *[DataLocation]\Autodesk\C3D 2011\enu\Tool Catalogs\Road Catalog*) need to be modified in order to list your new subassembly in the Civil 3D Subassembly Catalog.

NOTE

DataLocation in the path above depends on the operating system:

- On Windows XP it is *C:\Documents and Settings\All Users\Application Data*
- On Windows Vista and Windows 7, it is *C:\ProgramData\Autodesk\*

The *.atc* file format for .NET subassemblies is largely the same as for VBA subassemblies except that two new tags are used:

`<GeometryGenerateMode>UseDotNet</GeometryGenerateMode>`
`<GeometryGenerateMode>` tells Civil 3D that you are using a .NET subassembly. It is placed within the `<Tool>` tag.`<DotNetClass Assembly="%AECCCONTENT_DIR%\C3DStockSubassemblies.dll">Subassembly.NewCurb</DotNetClass>`
`<DotNetClass>` is used instead of `<Macro>` tag when using .NET subassemblies

Replacing the VBA Subassembly

Once the VBA custom subassembly has been ported to .NET and installed in a catalog file, drawings need to be updated to use the new code. Below is an example .NET macro that performs this task. This macro gets a subassembly by name (“VBASubassembly”). It then creates a new `SubassemblyGenerator` object, passing in the mode (“UseDotNet”), the dll in which the new subassembly is located, the name of the .NET subassembly. Finally it sets the subassembly `GeometryGenerator` parameter to the new `SubassemblyGenerator`. When the transaction is committed, the subassembly is replaced.

```
[CommandMethod("ConvertVbaSA")]
public void ConvertVbaSA()
{
    using(Transaction trans =
m_transactionManger.StartTransaction())
    {
    ObjectId saId =
Autodesk.Civil.ApplicationServices.CivilApplication.ActiveDocument.SubassemblyCollection["VBASubassembly"];
    Subassembly sa = trans.GetObject(saId,
OpenMode.ForWrite) as Subassembly;
    SubassemblyGenerator genData = new
SubassemblyGenerator(SubassemblyGeometryGenerateMode.UseDotNet,
 "C3DStockSubassemblies.dll",
"Subassembly.DotNetSubassembly");
    sa.GeometryGenerator = genData;
    trans.Commit();
    }
}
```

Legacy COM API

This section of the Developer's Guide contains chapters covering the legacy COM API. Where possible, the newer .NET API should be used for performance reasons. However, the .NET API does not cover all Civil 3D functionality, and you may want to use features that are only available in COM. You can access the COM API using interop - see Limitations and Using Interop (page 19).

Using VBA in AutoCAD Civil 3D

The following AutoCAD Civil 3D command-line instructions enable you to interact with the Visual Basic programming environment and loaded VBA macros.

NOTE

VBA support is not included with AutoCAD Civil 3D by default, and must be obtained as a separate download.

- **VBAMAN** - Displays a dialog box allowing you to load, unload, edit, embed, extract, edit, and save copies of VBA applications.
- **VBAIDE** - Displays the Visual Basic programming environment.
- **VBARUN** - Displays a dialog box where you can choose a macro to run from all the available subroutines.
- **-VBARUN**<*module.macro*> - Runs a particular macro from the command line.
- **VBALOAD** - Loads a global VBA project into the current work session. After entering this instruction, you are presented with a file selection dialog box.
- **-VBALOAD**<*full path and filename*> - Loads a particular project from the command line. If the path contains spaces, you need to type quotes around the path and filename string.
- **VBAUNLOAD** - Unloads an existing VBA project. After entering this instruction, you are prompted to type in the full path and filename of the project. If the path contains spaces, you need to type quotes around the path and filename string.
- **VBASTMT** - Executes a VBA statement on the command line. A statement generally occupies a single line, although you can use a colon (:) to include more than one statement on a line. VBA statements are executed in the context of the current drawing.

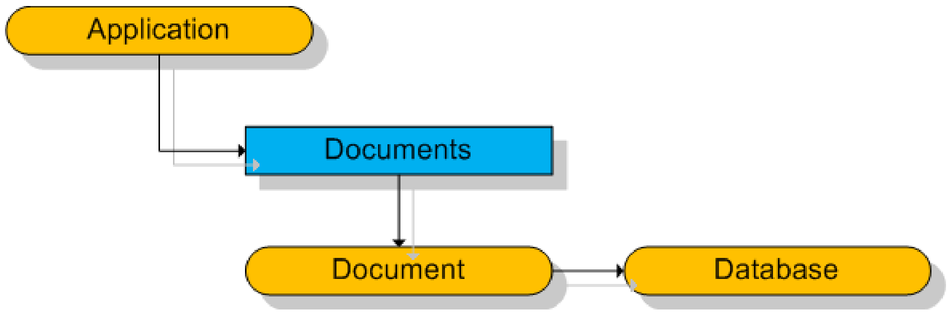
VBA projects are usually stored in separate *.dvb* files, which allow macros to interact with many separate AutoCAD Civil 3D drawings.

Root Objects and Common Concepts in COM

Root Objects

This section explains how to acquire references to the base objects which are required for all applications using the COM API. It also explains the uses of the application, document, and database objects and how to use collections, which are commonly used throughout the COM API.

Object Hierarchy



Root Object Model

Accessing Application and Document Objects

The root object in the AutoCAD Civil 3D COM hierarchy is the `AeccApplication` object. It contains information about the main application window, base AutoCAD objects, and a collection of all open documents. `AeccApplication` is inherited from the AutoCAD object `AcadApplication`. See the AutoCAD ObjectARX documentation for information about all inherited methods and properties.

The `AeccApplication` object is accessed by first using the exposed `ThisDrawing` object, an AutoCAD object of type `AcadDocument`. Its `AcadDocument.Application` property returns the AutoCAD `AcadApplication` object. From this point, use COM Automation to get the desired AutoCAD Civil 3D`AeccApplication` object.

This example demonstrates the process of accessing the `AeccApplication` and `AeccDocument` objects:

```
Dim oAcadApp As AcadApplication
Set oAcadApp = ThisDrawing.Application
' Specify the COM name of the object we want to access.
' Note that this always accesses the most recent version
' of AutoCAD Civil 3D installed.
Const sCivilAppName = "AeccXUiLand.AeccApplication.6.0"
Dim oCivilApp As AeccApplication
Set oCivilApp = oAcadApp.GetInterfaceObject(sCivilAppName)
```

```
' Now we can use the AeccApplication object.
' Get the AeccDocument representing the currently
' active drawing.
Dim oDocument As AeccDocument
Set oDocument = oCivilApp.ActiveDocument
' Set the viewport of the current drawing so that all
' drawing elements are visible.
oCivilApp.ZoomExtents
```

This sample gets the objects from AutoCAD Civil 3D 2009. To gain access to the libraries of an older version, use the version number of the desired libraries to the COM object name. For example, to make a program that works with AutoCAD Civil 3D 2007, replace:

```
Const sCivilAppName = "AeccXUiLand.AeccApplication.4.0"
```

with the following line of code:

```
Const sCivilAppName = "AeccXUiLand.AeccApplication.6.0"
```

The application object contains references to all open documents in the `AeccApplication.Documents` collection and the `AeccApplication.ActiveDocument` property. `AeccDocument` is inherited from the AutoCAD object `AcadDocument`. See the AutoCAD ObjectARX documentation for information on all inherited methods and properties.

Using Collections Within the Document Object

The document object not only contains collections of all AutoCAD Civil 3D drawing elements (such as surfaces and alignments) but all objects that modify those elements (such as styles and label styles). These collections have the same core set of methods: `Count`, `Item`, `Remove`, and `Add`. `Count` represents the number of items in the collection. `Item` returns an object from the collection, usually specified by the identification number or the name for the object. `Remove` deletes a specified item from the collection. `Add` creates a new item, adds it to the collection, and returns a reference to the newly created item. This item is already set with default values as defined by the document ambient settings.

This example creates a new point style:

```
Dim oPointStyle As AeccPointStyle
```

```
Set oPointStyle = oAeccDocument.PointStyles.Add("Name")
' Now a new point style is added to the collection of
styles,
' and we can modify it by setting the properties
' of the oPointStyle object.
oPointStyle.Elevation = 114.6
```

If you attempt to add a new element with properties that match an already existing element, try to access an item that does not exist, or remove an item that does not exist or is in use, an error will result. You should trap the error and respond accordingly.

The following sample demonstrates one method of dealing with such errors:

```
' Try to access the style named "Name".
Dim oPointStyle As AeccPointStyle
On Error Resume Next
Set oPointStyle = oAeccDocument.PointStyles.Item("Name")
' Turn off error handling as soon as we no longer need it.
On Error Goto 0

' If there was a problem retrieving the item, then
' the oPointStyle object will remain empty. Check if that
' is the case. If so, it is most likely because the item
' does not exist. Try making a new one.
If (oPointStyle Is Nothing) Then
    Set oPointStyle = oAeccDocument.PointStyles.Add("Name")
End If
```

Accessing and Using the Database Object

Each document has an associated database object of type `AeccDatabase` that is accessed through the `AeccDocument.Database` property. The database object is inherited from the AutoCAD ObjectARX object `AcadDatabase`. The `AeccDatabase` object contains references to all the same AutoCAD Civil 3D entities as the `AeccDocument` object. However, it is the only object that contains references to base AutoCAD entities. See the AutoCAD ObjectARX documentation for more information.

```
' Add a polyline to the drawing's collection of objects.
Dim oPolyline As AcadPolyline
Dim dPoints(0 To 8) As Double
```

```
dPoints(0) = 1000: dPoints(1) = 1000: dPoints(2) = 0
dPoints(3) = 1000: dPoints(4) = 4000: dPoints(5) = 0
dPoints(6) = 4000: dPoints(7) = 4000: dPoints(8) = 0
' The database is where all such objects are stored.
Set oPolyline =
oDocument.Database.ModelSpace.AddPolyline(dPoints)
oPolyline.Closed = True
```

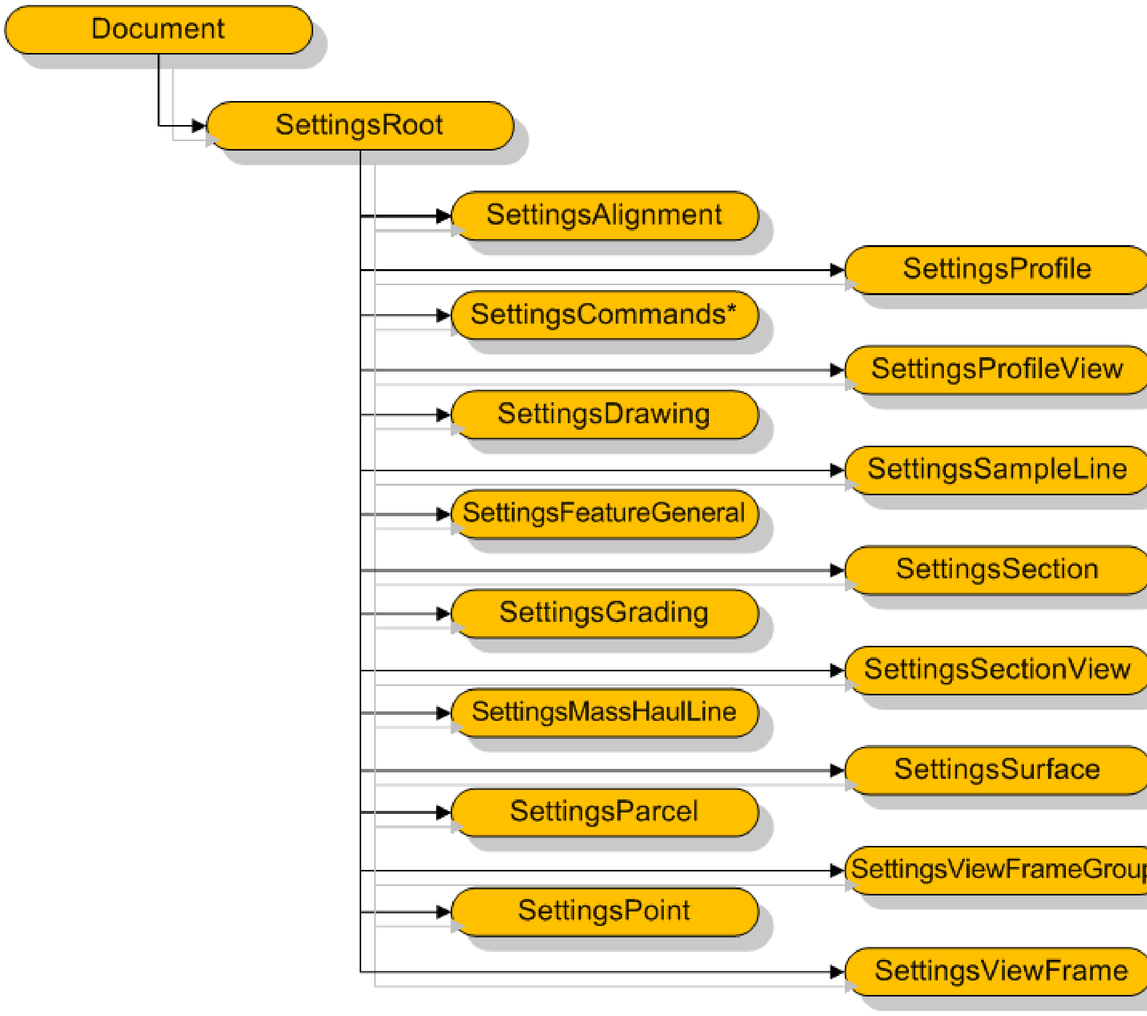
The `AeccDatabase` object also has an `AddEvent()` method, that lets you send an event to the Event Viewer in the AutoCAD Civil 3D user interface:

```
Dim oAeccApp As AeccApplication
Set oAeccApp =
ThisDrawing.Application.GetInterfaceObject("AeccXUiLand.AeccApplication.6.0")
oAeccApp.Init ThisDrawing.Application
Dim oDatabase As AeccDatabase
Set oDatabase = oAeccApp.ActiveDocument.Database
oDatabase.AddEvent aeccEventMessageError, "PipeLengthRule",
 "Parameter Bad"
```

Ambient Settings

This section explains the purpose and use of the document settings objects, and covers changing general and specific settings.

Object Hierarchy



Ambient Settings Object Model

Changing General and Specific Settings

Ambient settings are default properties and styles that apply to the drawing as a whole or to objects when they are first created. The document's settings are accessed through the properties of the `AeccSettingsRoot` object, which is obtained from the `AeccDocument.Settings` property. There are settings for objects and commands. The object properties define the settings for items in general, or of particular classes of items, such as alignments, gradings, parcels, points, profiles, profile views, sample lines, sections, section views, and surfaces. Although each of these objects are unique, they all share some common features:

Property Name	Description
`StyleSettings`	Specifies the default styles and label styles.
`LabelStyleDefaults`	Specifies the common attributes for labels.
`NameTemplate`	Specifies the standard pattern of names.
`AmbientSettings`	Specifies which units of measurement are used and displayed.
`CreationSettings`	Not on all objects. Optional. Specifies item-specific default values.

The command settings apply to commands, and correspond to the settings in the **Commands** folder for each item in the AutoCAD Civil 3D**Toolspace Settings Tab**.

The following sample determines what angle units are used for all displays related to points:

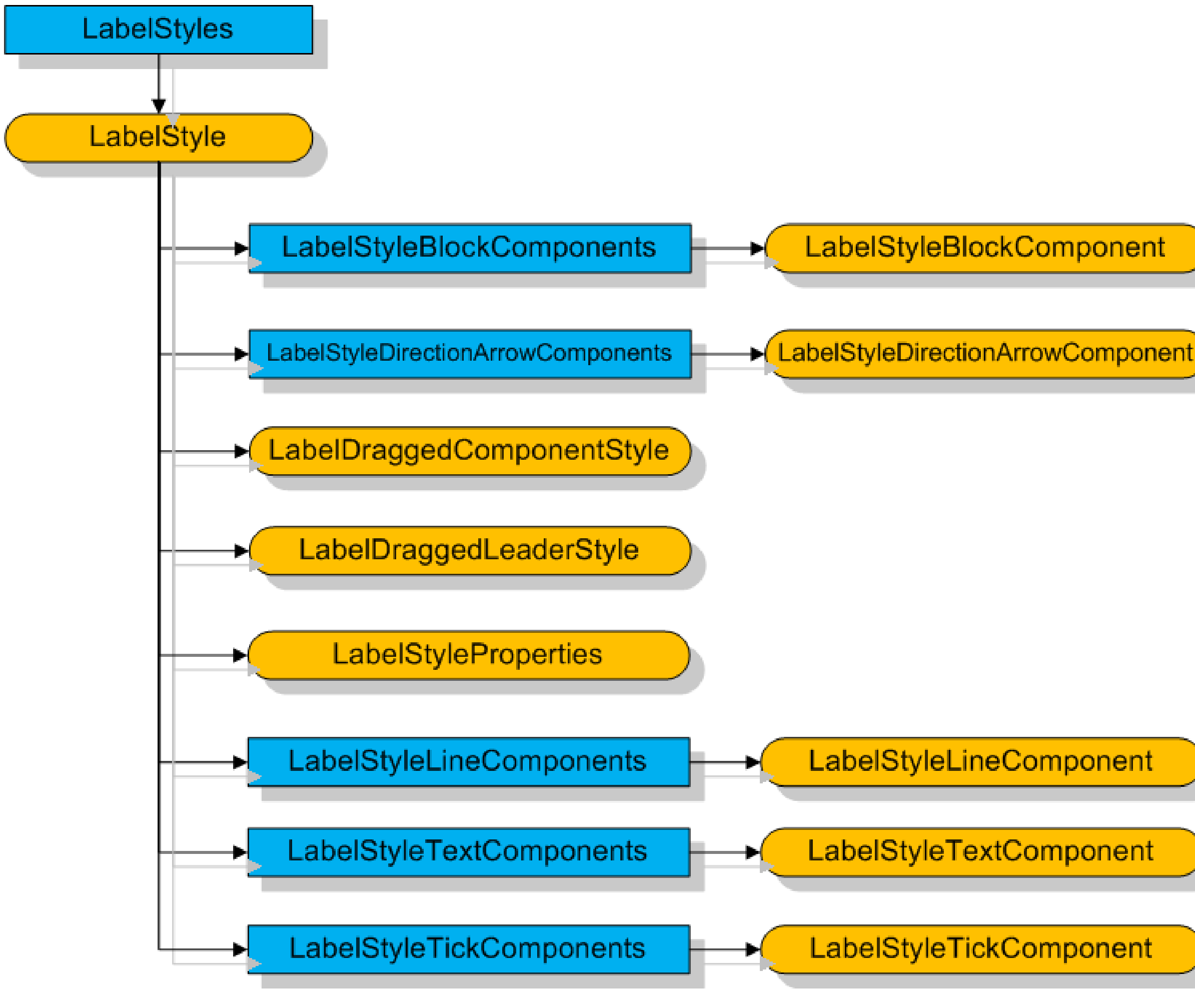
```
Dim oPointSettings as AeccSettingsPoint
Set oPointSettings = oDocument.Settings.PointSettings
Dim oAmbientSettings as AeccSettingsAmbient
Set oAmbientSettings = oPointSettings.AmbientSettings
Dim oAngleUnit as AeccAngleUnitType
Set oAngleUnit = oAmbientSettings.AngleSettings.Unit.Value

If (oAngleUnit = aeccAngleUnitDegree) Then
    ' Units are displayed in degrees
ElseIf (oAngleUnit = aeccAngleUnitRadian) Then
    ' Units are displayed in radians
Else
```

```
    ' Units are displayed in gradians
End If
```

Label Styles

This section explains common features of label styles. It covers creating a new label style object, defining a label style, and using property fields in label style text strings. Details specific to each construct are covered in the appropriate chapters.

Object Hierarchy



Label Style Object Model

Creating a Label Style Object

All types of annotation for AutoCAD Civil 3D elements are governed by label styles, which are objects of type `AeccLabelStyle`. A label style can include any number of text labels, tick marks, lines, markers, and direction arrows.

The following example creates a new label style object that can be used with points:

```
Dim oLabelStyle As AeccLabelStyle
Set oLabelStyle = oDocument.PointLabelStyles.Add _
  ("New Label Style for Points")
```

Defining a Label Style

A label style consists of collections of different features of a label. The properties containing these collections are: `AeccLabelStyle.BlockComponents` for symbols, `AeccLabelStyle.DirectionArrowComponents` for direction arrows, `AeccLabelStyle.LineComponents` for lines, `AeccLabelStyle.TextComponents` for text, and `AeccLabelStyle.TickComponents` for both major and minor tick marks. Not all of these may have meaning depending on the label style type. For example, adding a tick mark component to a label style meant for a point has no visible effect. Label styles also depend on graphical objects that may or may not be part of the current document. For example, if the style references a block that is not part of the current document, then the specified block or tick components are not shown.

To add a feature to a label style, add a new component to the corresponding collection. Then set the properties of that component to the appropriate values. Always make sure to set the `Visible` property to `True`.

```
' Add a line to the collection of lines in our label style.
Dim oLabelStyleLineComponent As AeccLabelStyleLineComponent
Set oLabelStyleLineComponent = oLabelStyle.LineComponents
 _
  .Add("New Line")

' Now the line can be changed to suit our purpose.
oLabelStyleLineComponent.Visibility = True
oLabelStyleLineComponent.color = 40 ' orange-yellow
oLabelStyleLineComponent.Angle = 2.094 ' radians, = 120
deg
' Negative lengths are allowed. They mean the line
' is drawn in the opposite direction of the angle
' specified.
oLabelStyleLineComponent.Length = -0.015
oLabelStyleLineComponent.StartPointXOffset = 0.005
oLabelStyleLineComponent.StartPointYOffset = -0.005
```

When first created, the label style object is set according to the ambient settings. Because of this, a new label style object may already contain features. If you are creating a new label style object, be sure to check for such existing features or your style might contain unintended elements.

```
' Check to see if text components already exist in the
' collection. If any do, just modify the first text
' feature instead of making a new one.
Dim oLabelStyleTextComponent As AeccLabelStyleTextComponent
If (oLabelStyle.TextComponents.Count > 0) Then
    Set oLabelStyleTextComponent = oLabelStyle _
      .TextComponents.Item(0)
Else
    Set oLabelStyleTextComponent = oLabelStyle _
      .TextComponents.Add("New Text")
End If
```

The ambient settings also define which units are used. If you are creating an application designed to work with different drawings, you should take ambient settings into account or labels may demonstrate unexpected behavior in each document.

Using Property Fields in Label Style Text

Text within a label is designated by the `AeccLabelStyleTextComponent.Contents` property, a string value. Of course, text labels are most useful if they can provide some sort of information that is unique to each particular item being labeled. This is accomplished by specifying property fields within the string. These property fields are of the form "<[*Property name*(*modifier 1*|[..] *modifier n*)]>". Modifier values are optional and can be in any order. Any number of property fields can be combined with normal text in the Contents property.

In this example, a string component of a label is modified to show design speeds and station values for a point along an alignment:

```
Dim oTextComponent As AeccLabelStyleTextComponent
Set oTextComponent = oLabelStyle.TextComponents.Item(0)
```

```
oTextComponent.Contents = "SPD=<[Design
Speed(P0|RN|AP|Sn)]>"
oTextComponent.Contents = oTextComponent.Contents & _
  "STA=<[Station Value(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>"
```

Valid property fields for each element are listed in the appropriate chapter.

Sharing Syles Between Drawings

Label styles, like all style objects, can be shared between drawings. To do this, call the style's `exportTo` method, targeting the drawing you want to add the style to. In this example, the first style in the `MajorStationLabelStyles` collection is exported from the active drawing to another open drawing named *Drawing1.dwg*:

```
Dim oDocument As AeccDocument
Set oDocument = oCivilApp.ActiveDocument
Dim dbDestination As AeccDatabase
Set dbDestination =
oCivilApp.Documents("Drawing1.dwg").Database
Call
oDocument.AlignmentLabelStyles.MajorStationLabelStyles(0).ExportTo(dbDestination)
```

Survey in COM

Object Hierarchy

Root Objects

This section describes how to gain access to the survey-specific versions of the root document and database objects. It covers the use of ambient settings, user settings, and the equipment database. It also describes the creation of survey projects, which contain networks and figures, and provide access to survey points.

Obtaining Survey-Specific Root Objects

Applications that perform survey operations require special versions of the base objects representing the application and document. The `AeccSurveyApplication` object is identical to the `AeccApplication` it is inherited from except that its `AeccSurveyApplication.ActiveDocument` property returns an object of type `AeccSurveyDocument` instead of `AeccDocument`. The `AeccSurveyDocument` object contains collections of survey-related items, such as projects and equipment databases in addition to all of the methods and properties of `AeccDocument`.

When using survey root objects, be sure to reference the “Autodesk Civil Engineering Survey 6.0 Object Library” (AeccXSurvey.tlb) and “Autodesk Civil Engineering UI Survey 6.0 Object Library” (AeccXUISurvey.tlb) libraries.

This sample demonstrates how to retrieve the survey root objects:

```
Dim oApp As AcadApplication
Set oApp = ThisDrawing.Application
Dim sAppName As String
sAppName = "AeccXUiSurvey.AeccSurveyApplication"
Dim oSurveyApplication As AeccSurveyApplication
Set oSurveyApplication = oApp.GetInterfaceObject(sAppName)

' Get a reference to the currently active document.
Dim oSurveyDocument As AeccSurveyDocument
Set oSurveyDocument = oSurveyApplication.ActiveDocument
```

Changing Survey-Specific Ambient Settings

Ambient settings for a survey document are held in the `AeccSurveyDocument.Settings` property, an object of type `AeccSurveySettingsRoot`. `AeccSurveySettingsRoot` inherits all the properties of the `AeccSettingsRoot` class from which it is derived. Ambient settings allow you to get and set the units and default property settings of survey objects. This is done through the `AeccSurveySettingsRoot.SurveySettings`, which contains a standard `AeccSettingsAmbient` object.

Changing Survey User Settings

The survey document also provides access to the survey user settings object. User settings are not specific to a particular document but are tied to a particular user, and all documents a user creates or loads will use the same settings. Survey user settings manage the visibility and appearance of prism sites, backsight lines, foresight lines, and baselines. Survey user settings also control how network and figure previews are shown, and under what conditions points, figures, and observations are automatically erased or exported. These settings also allow you to determine or set which figure prefix database, equipment database, or item of equipment is currently in use. The survey user settings object is retrieved through the `AeccSurveyDocument.GetUserSettings` method, which returns an `AeccSurveyUserSettings` object. To apply any changes made to the user settings, pass the modified `AeccSurveyUserSettings` object to the `AeccSurveyDocument.UpdateUserSettings` method.

```
Dim oUserSettings  As AeccSurveyUserSettings
Set oUserSettings = oSurveyDocument.GetUserSettings

' Modify and examine the current settings.
With oUserSettings
    .ShowBaseline = True
    Dim oColor As New AcadAcCmColor
    oColor.SetRGB 255, 165, 0 ' bright orange
    Set .BaselineColor = oColor
    .EraseAllFigures = False
    Debug.Print "Default layer:"; .DefaultFigureLayer
End with
```

```
' Save the changes to the user settings object.
oSurveyDocument.UpdateUserSettings oUserSettings
```

Using the Equipment Database

The equipment database contains a list of equipment used to gather surveying data. The information about each item of equipment is used in least squares and other computations. A collection of all equipment lists is contained in the `AeccSurveyDocument.EquipmentDatabases` property. Each equipment database is a collection of individual items of equipment. An equipment database is an object of type `AeccSurveyEquipmentDatabase`, and contains methods for searching the list of equipment and for copying lists from other databases.

Each item of equipment is represented by an `AeccSurveyEquipment` object, which contains properties describing aspects of the equipment, including the name and description, how the instrument measures angles, the unit types for angle and distance, Electronic Distance Meter settings, prism accuracy and offset, and the accuracy of the instrument.

This sample program displays selected information about each equipment item in the document's database:

```
Dim oEquipDatabases As AeccSurveyEquipmentDatabases
Dim oEquipDatabase As AeccSurveyEquipmentDatabase
Dim oEquipment As AeccSurveyEquipment

Set oEquipDatabases = oSurveyDocument.EquipmentDatabases
For Each oEquipDatabase In oEquipDatabases
   Debug.Print "Database: "; oEquipDatabase.Name
   Debug.Print

   For Each oEquipment In oEquipDatabase
   With oEquipment
      Debug.Print "----"
      Debug.Print "Item: "; .Name; "   Id: "; .Id
      Debug.Print " Description: "; .Description
      Debug.Print " Angle Type: "; .AngleType
      Debug.Print " Angle Unit: "; .AngleUnit
      Debug.Print " Azimuth Std: "; .AzimuthStandard
      Debug.Print " Wave Constant: "; .CarrierWaveConstant
      Debug.Print " Center Standard: "; .CenterStandard
      Debug.Print " Circle Standard: "; .CircleStandard
      Debug.Print " Coordinate Std: "; .CoordinateStandard
      Debug.Print " Distance Unit: "; .DistanceUnit
      Debug.Print " Edm Error: "; .EdmMmError; "mm"
      Debug.Print " Edm Error: "; .EdmPpmError; "ppm"
      Debug.Print " Edm Offset: "; .EdmOffset
      Debug.Print " Elevation Std: "; .ElevationStandard
      Debug.Print " H Collimation: "; .HorizontalCollimation
      Debug.Print " Is Prism Tilted: "; CStr(.IsTiltedPrism)
      Debug.Print " Measuring Device: "; .MeasuringDevice
      Debug.Print " Pointing Std: "; .PointingStandard
      Debug.Print " Prism Constant: "; .PrismConstant
      Debug.Print " Prism Offset: "; .PrismOffset
      Debug.Print " Prism Std: "; .PrismStandard
      Debug.Print " Revision: "; .Revision
      Debug.Print " Target Std: "; .TargetStandard
      Debug.Print " Theodolite Std: "; .TheodoliteStandard
      Debug.Print " Vertical Angle Type: ";
.VerticalAngleType
      Debug.Print " V Collimation: "; .VerticalCollimation
      Debug.Print
   End With
   Next
Next
```

Creating a Survey Project

The survey database is the high-level construct that contains collections of networks and figures, and provides access to survey points. Throughout the API, survey databases are called “projects.” The collection of all projects in a document are held in the `AeccSurveyDocument.Projects` property. Once created, a project cannot be removed from this collection using API methods. The only way to remove a survey project is to delete the Project folder and refresh the collection. When a new project is created, a unique GUID identifying the project is generated.

This sample creates a new survey project with the name “Proj01”:

```
Dim oSurveyProject As AeccSurveyProject
Set oSurveyProject =
oSurveyDocument.Projects.Create("Proj01")
```

```
' Print the next available survey point Id available.
Debug.Print "Next available Id:"; _
  oSurveyProject.GetNextWritablePointNumber()
```

Adjusting Survey Project Settings

A survey project has a group of properties accessed through the survey project settings object. Survey project settings define what measurement units are used, what angle types are used, and the precision of each measurement. It records what kinds of adjustments are made to observations, and the types and accuracy of traverse analyses and angle balancing. It also has methods for converting between metric units and the units of the ambient settings, and between easting and northing and longitude and latitude. The project settings object is retrieved through the `AeccSurveyProject.GetProjectSettings` method, which returns an `AeccSurveyProjectSettings` object. To apply any changes made to the project settings, pass the modified `AeccSurveyProjectSettings` object to the `AeccSurveyProject.UpdateProjectSettings` method.

```
Dim oProjectSettings As AeccSurveyProjectSettings
Set oProjectSettings = oSurveyProject.GetProjectSettings()

' Modify and examine the current settings.
With oProjectSettings
   .AngleType = aeccSurveyAngleTypeAzimuth
   Debug.Print "Sea level correction? :";
.SeaLevelCorrection
End With

' Save the changes to the project settings object.
oSurveyProject.UpdateProjectSettings oProjectSettings
```

Accessing Extended Properties

Survey root objects can have extended properties associated with them. LandXML extended properties are used to import and export Survey LandXML attributes or collections of elements. For example, in the LandXML schema, a CgPoint element (a SurveyPoint) within a CgPoints collection (a SurveyPointGroup), may contain a DTMAttribute where its enumeration value can be used to aid the user in determining whether the point should be included in a surface definition. User-defined extended properties can be used to define additional attributes beyond those defined by the LandXML schema. For an example of standard LandXML extended properties, see the *C:\Documents and Settings\All Users\Application Data\Autodesk\C3D2012\enu\Survey\LandXML - Standard.sdx_def*file.

The `AeccSurveyFigure`, `AeccSurveyNetwork`, `AeccSurveyPoint`, and `AeccSurveyProject` objects each have a `LandXMLXPropertiesRoot` property, which gives you access to the `AeccSurveyLandXMLXPropertiesRoot` for that object. Those objects also each have a UserDefinedXProperties object, which gives you access to the `AeccSurveyUserDefinedXProperties` for that object.

Survey Network

A network represents the spatial framework of a survey. Each network contains a collection of control points, directions, setups, and non-control points. It also contains information on how these objects are related, such as how the control points and setups are connected relative to one another.

Creating a Survey Network

Survey networks are created through the `Create` method of the `AeccSurveyProject.Networks` collection.

```
Dim oSurveyNetwork As AeccSurveyNetwork
Set oSurveyNetwork =
oSurveyProject.Networks.Create("Net_01")
```

Adding Control Points to a Network

Control points are the “known” or “fixed” locations that other survey points are based from. The collection of control points in a network is held in the `AeccSurveyNetwork.ControlPoints` property. New control points can be added to the network through the collection’s `Create` method. `Create` adds a new point with the specified features to the `AeccSurveyControlPoints` collection and returns a reference to the newly created `AeccSurveyControlPoint` object.

```
' Get collection of all control points.
```

```
Dim oControlPoints As AeccSurveyControlPoints
Set oControlPoints = oSurveyNetwork.ControlPoints

' Create a control point with an id number of 3001 at
' the location 5000.0m, 5000.0m, elevation 100.0m.
Dim oControlPoint As AeccSurveyControlPoint
Set oControlPoint = oControlPoints.Create( _
  3001, _
  "ControlPoint_01", _
  "Description of control point", _
  5000#, _
  5000#, _
  100#)
```

Adding Directions to a Network

A direction is a “known” or “fixed” direction (either azimuth or bearing) from a control point to another point reference. The point reference may be observed at a later time in the survey, or it may never actually be occupied by a setup. This can happen if the reference point is a mountain top or tower or some other location where survey equipment cannot physically be placed but the direction from the control point to that location is known. An entire survey network can be defined from a single control point and a single direction.

The collection of directions in a network is held in the `AeccNetwork.Directions` property. New directions can be added to the network through the collection’s `Create` method. `Create` adds a new direction between two points at the specified angle to the `AeccSurveyDirections` collection and returns a reference to the newly created `AeccSurveyDirection` object.

```
' Get collection of all directions.
Dim oDirections As AeccSurveyDirections
Set oDirections = oSurveyNetwork.Directions

' Create a direction from point 3001 to the (not yet
' existing) point 3004 at an angle of 45.0 degrees azimuth.
Dim oDirection As AeccSurveyDirection
' 0.785398163 = 40 degrees in radians
Set oDirection = oDirections.Create( _
  3001, _
  3004, _
```

```
    0.785398163, _
    aeccSurveyDirectionTypeAzimuth)
```

Adding Setups to a Network

A setup represents a survey instrument session made in the field. The instrument is set up on a control point or a previously observed point, and a reference direction (backsight) is made to another point. Locations on the Earth's surface are then determined with surveying methods relative to the setup and the direction of the backsight. These locations are called "observations", and are represented by an `AeccSurveyObservation` object. Each setup object contains a collection of observation objects. An observation object is composed of any or all of the following measurements: angle, distance, vertical, latitude, longitude, northing, easting, or elevation. The observation object also has a series of properties describing the nature of the observation equipment.

The collection of setups in a network is held in the `AeccNetwork.Setups` property. New setups can be added to the network through the collection's `Create` method. `Create` adds a new setup at the specified location to the `AeccSurveySetups` collection and returns a reference to the newly created `AeccSurveySetup` object.

This sample creates a setup at the location of point 3001 using the location of point 3004 as the backsight. It then creates an observation to another location, which is given the identification "3002".

```
Dim oSetups As AeccSurveySetups
Dim oSetup1 As AeccSurveySetup

Set oSetups = oSurveyNetwork.Setups
' Create a setup at 3001 with a backsight at 3004 (the
' backsight direction should be calculated automatically).
Set oSetup1 = oSetups.Create(3001, 3004)
Debug.Print "Direction:"; oSetup1.BacksightDirection
Debug.Print "Orientation:"; oSetup1.BacksightOrientation
' Now any observation angle is based on the line
' from 3001 to 3004.

Dim oObservations As AeccSurveyObservations
Set oObservations = oSetup1.Observations
Dim oObservation1 As AeccSurveyObservation
```

```
' On setup "Station:3001, Backsight:3004" create an
observation
' of point 3002.
'    Angle = 90 degrees (1.57079633 radians)
'    Angle Type = Angle
'    Distance = 100#
'    Distance Type = Slope
'    Vertical = 90 degrees (1.57079633 radians)
'    Vertical Type = Vertical Angle
'    Target Height = 0#
'    Target Type = Prism
Set oObservation1 = oObservations.Create( _
  3002, _
  aeccSurveyAngleTypeAngle, _
  1.57079633, _
  aeccSurveyDistanceTypeSlope, _
  100#, _
  aeccSurveyVerticalTypeVerticalAngle, _
  1.57079633, _
  aeccSurveyTargetTypePrism, _
  0#, _
  "From setup <Station:3001, Backsight:3004> to Point 3002")

' From this survey equipment data, the location of point
' 3002 can be determined:
Debug.Print "Point 3002 at:"; oObservation1.Easting;
Debug.Print ", "; oObservation1.Northing
```

Adding Non-control Points to a Network

A survey network also contains a collection of non-control points, which represent points within a survey network whose location is determined by observation, but are not connected to other survey observations and remain unaffected by a network analysis. These non-control points are represented by objects of type `AeccNonControlPoint`, and are defined by an easting, northing, and an optional elevation. The collection of all non-control points in a survey network are held in the `AeccSurveyNetwork.NonControlPoints` property.

```
' Create a non-control point with an id number of 3006 at
' the location 4950.0, 5000.0, elevation 100.0.
Dim oNonControlPt As AeccSurveyNonControlPoint
Set oNonControlPt = oSurveyNetwork.NonControlPoints.Create(
 _
  3006,
  "NonControlPoint_01",
  "Description of non-control point",
  4950#,
  5000#,
  100#)
```

A non-control point may be promoted to a control point if you reference the point as a control point when creating a traverse, or reference the point as a setup to make observations to other points that may affect locations during an analysis.

Creating Paths for Traverse Analysis

A traverse is a path through survey points. A traverse starts at a control point, continues through other survey points, and either returns to the original control point (a “closed” loop) or ends at another control point (an “open” loop). These paths are used for traverse analysis, which are methods for determining the amount of error in the locations and directions of survey points. The API provides methods for creating and examining traverse paths, but it does not provide methods for performing a traverse analysis.

The collection of all traverses in a network are stored in the `AeccSurveyNetwork.Traverses` property. A single traverse is represented by an object of type `AeccSurveyTraverse`. New traverses are created using the collection’s `Create` method. This method requires the identification number of the starting control point, a backsight for that point, the final control point reached, and an array of the identification numbers of all intermediate points.

This sample creates a traverse starting at control point 3001, continuing though points 3002, 3003, and 3004, and finishing back at the starting control point 3001. The user can then perform a traverse analysis based on this closed loop.

```
Dim oTraverse As AeccSurveyTraverse
Dim lStations(0 To 2) As Long
lStations(0) = 3002: lStations(1) = 3003; lStations(2) =
3004
Set oTraverse = oSurveyNetwork.Traverses.Create( _
  "Traverse_01", _
```

```
    3001, _
    3004, _
    3001, _
    lStations)
```

Adding Survey Data to the Drawing

Adding elements or modifying information in a survey network object changes the survey database but does not automatically change the drawing. After you are done adding points, directions, and other elements that have a graphical element, call the `AeccSurveyNetwork.AddToDrawing` method to create AutoCAD elements that correspond to the survey network data. This is equivalent to the AutoCAD Civil 3D command “Insert Into Drawing”.

```
oSurveyNetwork.AddToDrawing
```

Getting Survey Network Drawing Objects

You can get and manipulate the AutoCAD drawing objects that make up a survey network from the AutoCAD Civil 3D user interface. The drawing objects are represented by the `AeccSurveyNetworkEntity` object. For example, this code prompts the user to select the survey network, tests whether it is the survey network object, and then prints some information about it:

```
Dim objPart As AeccSurveyNetworkEntity
Dim objEnt As AcadObject
Dim objAcadEnt As AcadEntity
Dim varPick As Variant
ThisDrawing.Utility.GetEntity objEnt, varPick, "Select the
 Survey Network"
If TypeOf objEnt Is AeccSurveyNetworkEntity Then
   Set objPart = objEnt
   Debug.Print objPart.Name, TypeName(objPart)
ElseIf TypeOf objEnt Is AcadEntity Then
   Set objAcadEnt = objEnt
   Debug.Print objAcadEnt.Name, TypeName(objAcadEnt)
End If
```

Figures

Figures are a group of connected lines and arcs that have meaning within a survey. A figure can represent a fence, building, road, parcel, or similar object. Unlike a normal polyline, a vertex in a figure can reference a survey point. If a referenced survey point is moved, vertices in the figure are moved as well. The first line or arc added to the figure sets the location all other lines and arcs will be drawn from. Each line and arc added to a figure is in turn based on the endpoint of the previous element. The position of the last endpoint can be determined from the read-only properties `AeccSurveyFigure.LastPointX` and `AeccSurveyFigure.LastPointY`.

Creating a Figure Object

A collection of all figures in the survey database are stored in the `AeccSurveyProject.Figures` property. New figures are made using the collection's `Create` method.

```
Dim oFigure As AeccSurveyFigure
Set oFigure = oSurveyProject.Figures.Create("Figure_01")
```

Adding Lines to a Figure

Each line added to a figure is drawn from the endpoint of the previous line or arc. The new line can be drawn to a particular point location, for a distance along an absolute azimuth, or for a distance along an azimuth relative to the direction of the previous line. If the figure has no lines or arcs, then the first line added will only set the point that the next line or arc is drawn from.

AddLineByPoint

`AddLineByPoint` adds a line to the figure to the specified point location. An optional parameter can specify a survey point to reference so that whenever it changes the figure vertex will change as well.

```
' Draw a line to the location of survey point 3001.
Dim oPoint1 As AeccSurveyPoint
Set oPoint1 = oSurveyProject.GetPointByNumber(3001)
```

```
oFigure.AddLineByPoint oPoint1.Easting, oPoint1.Northing,
 3001
```

AddLineByAzimuthDistance

`AddLineByAzimuthDistance` adds a line to the figure of the specified length and of the specified angle from the major axis of the survey project.

```
' Draw a line 30 meters long at 10 degrees from the major
 axis.
oFigure.AddLineByAzimuthDistance 0.17453, 30
```

AddLineByDeltaAzimuthDistance

`AddLineByDeltaAzimuthDistance` adds a line to the figure of the specified length and of the specified angle from the endpoint of the previous line.

```
' Draw a line 20 meters long at 270 degrees from the
' previous line.
oFigure.AddLineByDeltaAzimuthDistance 4.7124, 20
```

Adding Arcs to a Figure

Each arc added to a figure is drawn from the endpoint of the previous line or arc. `AddArc` adds a segment of a circle to the figure to the specified point location. The shape of the arc is defined by the `Radius`, `Bulge`, and `CenterX` and `CenterY` parameters. The `Bulge` parameter defines what fraction of a circle the arc comprises. If the `Radius` parameter is zero or negative the arc is curved in the clockwise direction, otherwise the arc is curved in the counter-clockwise direction. An optional parameter can specify a survey point to reference so that whenever it changes the arc endpoint will change as well.

```
' Create a clockwise arc comprising of roughly half a circle

' that ends at survey point 3001.
oFigure.AddArc 10, 1.7, 0, 0, oPoint1.Easting, _
  oPoint1.Northing, , 3001
```

Adding Figures to the Drawing

Adding lines and arcs to a figure changes the survey database but does not automatically change the drawing. After you are done adding elements to the figure, call the `AeccSurveyFigure.AddToDrawing` method to create AutoCAD elements that correspond to the figure. This is equivalent to the AutoCAD Civil 3D command “Insert Into Drawing”.

Figures and AutoCAD Civil 3D

A figure can influence AutoCAD Civil 3D objects such as parcels and TIN surfaces. If the `AeccSurveyFigure.IsLotLine` property is set to `True`, then parcel segments are created for each figure line and arc when the figure is added to the drawing. If any set of parcel segments creates closed figures, then a parcel is formed. You can assign the parcel segments to a particular site by setting the `AeccSurveyFigure.Site` property - otherwise, a new site named “Survey Site” is created automatically. The figure still remains in the survey database, and deleting the figure removes all associated parcel segments from the drawing.

A figure can also form a breakline that defines how surfaces are triangulated. This is accomplished by setting the `AeccSurveyFigure.IsBreakline` property to `True`

Creating a Figure Style

A figure style controls the visual appearance of figures. The `AeccSurveyFigureStyle` object has `AeccDisplayStyle` properties for controlling the color, line type, and visibility of lines and of markers at the figure endpoints, the figure midpoint, at line vertices. The types of markers used are controlled by separate `AeccMarkerStyle` properties for the start, mid, and end points of the figure, and for all vertexes. Markers can also be drawn at a normal alignment to the orientation of the figure by setting the `IsAlignAdditionalMarkersWithFigure`, `IsAlignMidPointMarkersWithFigure`, `IsAlignStartAndEndPointMarkersWithFigure`, and `IsAlignVertexMarkersWithFigure` properties to `True` for the appropriate type of marker.

Additional markers can also be placed along the figure. The nature of these additional markers is set by the

AeccSurveyFigureStyle.AdditionalMarkersPlacementMethod property. If the placement method is set for interval placement, then a new marker is placed every *n* units apart where *n* is the value of the AeccSurveyFigureStyle.AdditionalMarkersInterval property. If the placement method is set for divided placement, then the figure is divided into *n* parts of equal length where *n* is the value of the AeccSurveyFigureStyle.AdditionalMarkersDivideFigureBy property. A marker is placed at each part, including the figure start and end points. If the placement method is set for continuous, then the markers are placed exactly one marker's width apart along the length of the figure.

You can determine the style of figure drawing by examining the FigureDisplayMode property. There are three ways a figure can be visualized: using figure elevations, flattening the figure to a single elevation, or exaggerating figure elevations. If the figure is flattened to a single elevation, the elevation can be read from the FlattenFigureElevation property. If the figure elevations are exaggerated when displayed, the amount of exaggerations is held in the read-only FigureElevationScaleFactor property.

All figure styles are stored in the AeccSurveyDocument.FigureStyle collection. The figure object's AeccSurveyFigure.Style property takes the string name of the style to use.

This sample creates a new figure style object and adjusts some of the style settings:

```
Dim oFigureStyles As AeccSurveyFigureStyles
Dim oFigureStyle As AeccSurveyFigureStyle
Set oFigureStyles = oSurveyDocument.FigureStyles
Set oFigureStyle = oFigureStyles.Add(sStylename)

' Set the style so that additional markers are visible,
' blue, and drawn every 20 units along the figure.
With oFigureStyle
    .AdditionalMarkersDisplayStylePlan.Visible = True
    .AdditionalMarkersDisplayStylePlan.Color = 150 ' blue
    .AdditionalMarkersPlacementMethod = _
      aeccSurveyAdditionalMarkerPlacementMethodAtInterval
    .AdditionalMarkersInterval = 20
End With

' Assign the style to a figure.
oFigure.Style = oFigureStyle.Name
```

Using the Figure Prefix Database

A figure read from a fieldbook file can have a letter prefix signifying what object or concept the figure represents. This figure prefix describes the style and property settings to use with the figure. A list of figure prefixes is stored in `AeccSurveyFigurePrefixDatabase` objects. The collection of all databases in the document are stored in the `AeccSurveyDocument.FigurePrefixDatabases` property. New figure prefixes and databases are added through the parent collection’s `Create` method. Once a prefix or database is created, it becomes a permanent part of the figure prefix database and it is not lost upon loading a new document or running a new instance of AutoCAD Civil 3D. Because of this, it is important to check for existing prefix databases and prefixes by name before trying to create new ones. The `AeccSurveyFigurePrefixDatabases.FindItem` method can be used to search for an existing database id or name.

NOTE

The similarly named `AeccSurveyFigurePrefixDatabase.FindItem` can only be used to search for the identification numbers of prefixes within a database - to find a prefix by name, use the `AeccSurveyFigurePrefixDatabase.GetMatchedFigurePrefix` method.

This sample creates a new figure prefix database and a new figure prefix. It switches the current database to the newly created one, and sets an existing figure to use the new prefix’s style.

```
' Get a reference to all the prefix databases.
Dim oPrefixDatabases As AeccSurveyFigurePrefixDatabases
Set oPrefixDatabases = oSurveyDocument.FigurePrefixDatabases

' See if our database already exists. If it does not,
' create a new one.
Dim oPrefixDatabase As AeccSurveyFigurePrefixDatabase
Set oPrefixDatabase = oPrefixDatabases.FindItem("NewDB")
If (oPrefixDatabase Is Nothing) Then
    Set oPrefixDatabase = oPrefixDatabases.Create("NewDB")
End If

' See if our figure prefix already exists. If it does not,
' create a new figure prefix.
Dim oSurveyFigurePrefix As AeccSurveyFigurePrefix
```

```
On Error Resume Next
Set oSurveyFigurePrefix = _
  oPrefixDatabase.GetMatchedFigurePrefix("BV")
On Error GoTo 0
If (oSurveyFigurePrefix Is Nothing) Then
    Set oSurveyFigurePrefix = oPrefixDatabase.Create("BV")
End If

' Set the properties of the prefix.
oSurveyFigurePrefix.Style = _
  oSurveyDocument.FigureStyles(0).Name
oSurveyFigurePrefix.IsLotLine = True
oSurveyFigurePrefix.IsBreakline = True
oSurveyFigurePrefix.Layer = "0"
oSurveyFigurePrefix.Save
```

You can set a figure to use a prefix style manually by using the `AeccSurveyFigure.InitializeFromFigurePrefix` method. It searches through the current prefix database for a prefix name that matches the first part of the name of the figure. For example, a figure with the name "BV 01" matches a prefix with the name "BV". The current prefix database can be determined through the `CurrentFigurePrefixDatabase` property of the document's survey user settings. The following code shows the correct method for doing this:

```
Dim oUserSettings  As AeccSurveyUserSettings
Dim CurrentDatabase As String
Set oUserSettings = oSurveyDocument.GetUserSettings
CurrentDatabase = oUserSettings.CurrentFigurePrefixDatabase
```

You can change the current database by setting the `CurrentFigurePrefixDatabase` property to the new name and then updating the survey user settings:

```
oUserSettings.CurrentFigurePrefixDatabase = "NewDB"
oSurveyDocument.UpdateUserSettings oUserSettings
```

Sample Program

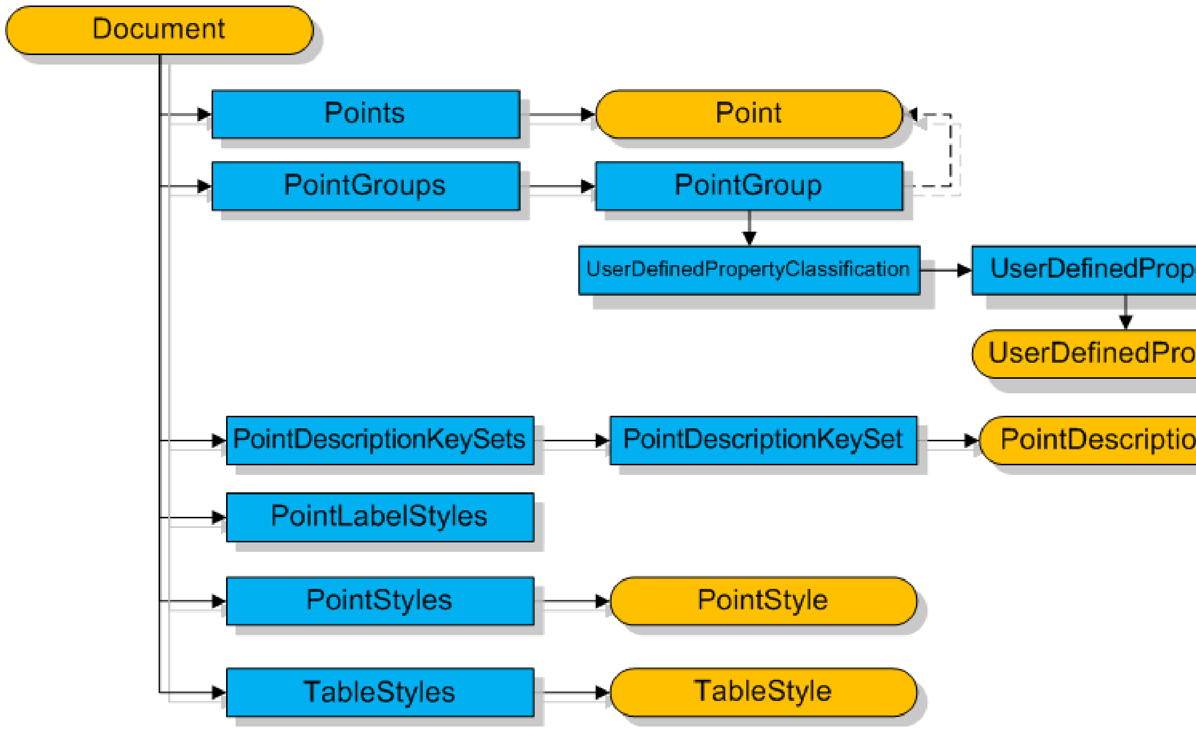
SurveySample.dvb

<installation-directory>\Sample\Civil 3D API\Vba\Survey\SurveySample.dvb

This sample creates a simple survey from hard-coded data using control points, setups, directions, and figures. A survey style is created and applied. The ambient settings and other global settings are demonstrated.

Points in COM

Object Hierarchy



Points Object Model

Points

This section covers the collection of all points in a document, accessing points stored in a file, and the use of the point object.

Using the Points Collection

All points in a document are held in the `AeccDocument.Points` property, an object of type `AeccPoints`. Besides the usual collection properties and methods, the Points object also has methods for dealing with large numbers of points at once. An array of positions can be added using the `AeccPoints.AddMultiple` method.

The following example adds a series of points to the `AeccDocument.Points` collection using `AddMultiple` and then accesses points in the collection directly:

```
' This adds an array of point locations to the document's
' points collection.
Dim lNumAdded As Long
Const NUM_LOCATIONS As Long = 3
' One dimensional array, 3 for each point location.
Dim dLocations(0 To (3 * NUM_LOCATIONS) - 1) As Double
' One point per line
dLocations(0) = 4927: dLocations(1) = 3887: dLocations(2)
 = 150
dLocations(3) = 5101: dLocations(4) = 3660: dLocations(5)
 = 250
dLocations(6) = 5144: dLocations(7) = 3743: dLocations(8)
 = 350
lNumAdded = oPoints.AddMultiple(NUM_LOCATIONS, dLocations,
 0)

' This computes the average elevation of all points in a
document.
Dim oPoints As AeccPoints
Dim i As Long
Dim avgElevation As Double
Set oPoints = g_oAeccDocument.Points
For i = 0 To oPoints.Count - 1
    avgElevation = avgElevation + oPoints.Item(i).Elevation
Next i
avgElevation = avgElevation / oPoints.Count
MsgBox "Average elevation: "& avgElevation & _
  vbNewLine & "Number of points: " & oPoints.Count
```

Accessing Points in a File

The `AeccPoints` object also has methods for reading and writing point locations in a file. The `AeccPoints.ImportPoints` method creates points from locations stored in a text file. The `AeccPoints.ExportPoints` method writes point locations to a text file.

The second parameter of the `ImportPoints` and `ExportPoints` methods is a string that describes how the point values are stored in the file. The following table lists some commonly available file formats. You can create other formats by using the Point File Format dialog box.

Point File Formats

String Literal	Format of values in the file
ENZ (comma delimited)	Easting, Northing, Elevation
NEZ (space delimited)	Northing Easting Elevation
NEZ (comma delimited)	Northing, Easting, Elevation
PENZ (space delimited)	Point# Easting Northing Elevation
PENZ (comma delimited)	Point#, Easting, Northing, Elevation
PENZD (space delimited)	Point# Easting Northing Elevation Description
PENZD (comma delimited)	Point#, Easting, Northing, Elevation, Description
PNE (space delimited)	Point# Northing Easting
PNE (comma delimited)	Point#, Northing, Easting
PNEZ (space delimited)	Point# Northing Easting Elevation
PNEZ (comma delimited)	Point#, Northing, Easting, Elevation
PNEZD (space delimited)	Point# Northing Easting Elevation Description

String Literal	Format of values in the file
PNEZD (comma delimited)	Point#, Northing, Easting, Elevation, Description
ENZ (space delimited)	Easting Northing Elevation
Autodesk Uploadable File	Point# Northing Easting Elevation Description

The third parameter of the `ImportPoints` method is an object of type `AeccPointImportOptions`, which can be set to perform actions as the data is being loaded. For example, you can add offsets to the point positions or elevations, determine which points to read from the file, or specify the point group where the points are placed. The third parameter of the `ExportPoints` method is of the similar `AeccPointExportOptions` type.

This example demonstrates the `ImportPoints` and `ExportPoints` methods:

```
Dim oPoints As AeccPoints
Dim oImportOptions As New AeccPointImportOptions
Dim sFilename As String
Dim sFileFormat As String
Dim iCount As Integer

Set oPoints = oDocument.Points
sFilename = "C:\My Documents\SamplePointFile.txt"
sFileFormat = "PENZ (space delimited)"
oImportOptions.PointDuplicateResolution =
aeccPointDuplicateOverwrite
iCount = oPoints.ImportPoints(sFilename, sFileFormat,
oImportOptions)

' Export the files to a separate file.
Dim oExportOptions As New AeccPointExportOptions
sFilename = "C:\My Documents\SamplePointFile2.txt"
oExportOptions.ExpandCoordinateData = True
oPoints.ExportPoints sFilename, sFileFormat, oExportOptions
```

NOTE

When you add points using the `ImportPoints` method, it is possible that the point numbers will conflict with those that already exist in the drawing. In such cases, the user is given an option to renumber the point numbers from the file, or to cancel the operation which will result with a Visual Basic error. An application that uses `ImportPoints` should take this into account.

Using Points

Each individual point is represented by an object of type `AeccPoint`. The point object contains the identification number, description, and location for the point. The identification number, held in the `Point.Number` property, is unique and is automatically assigned when the point is first created. It cannot be changed. The read-only `Point.FullDescription` property is only meaningful when the point is read from a file that contains description information.

You can access the local position through either the `AeccPoint.Easting` and `AeccPoint.Northing` properties or by using the `AeccPoint.Location` property, a three-element array containing the easting, northing, and elevation. The point's location can also be specified by using the `AeccPoint.Grideasting` and `AeccPoint.GridNorthing` properties or the `AeccPoint.Latitude` and `AeccPoint.Longitude` properties, depending on the coordinate settings of the drawing.

This sample adds a new point to the document's collection of points and sets some of its properties.

```
Dim oPoints As AeccPoints
Set oPoints = g_oAeccDocument.Points
Dim oPoint1 As AeccPoint
Dim dLocation(0 To 2) As Double
dLocation(0) = 4958
dLocation(1) = 4078
Set oPoint1 = oPoints.Add(dLocation)
oPoint1.Name = "point1"
oPoint1.RawDescription = "Point Description"
```

Point User-Defined Properties

Point objects can have user-defined properties associated with them, and the properties can be organized into user-defined classifications, or are put into an “Unclassified” classification. You can create new classifications and user-defined properties via the API, though you can’t access the values of existing user-defined properties attached to points. For more information about user-defined properties and classifications, see User-Defined Property Classifications in the .

This sample creates a new user-defined property classification for points called “Example”, and then adds a new user-defined property with upper and lower bounds and a default value:

```
Dim oApp As AcadApplication
Set oApp = ThisDrawing.Application
' NOTE - Always specify the version number.
Const sAppName = "AeccXUiLand.AeccApplication.6.0"
Set g_vCivilApp = oApp.GetInterfaceObject(sAppName)
Set g_oDocument = g_vCivilApp.ActiveDocument
Set g_oAeccDb = g_oDocument.Database
Dim oUDPClass As AeccUserDefinedPropertyClassification
Dim oUDPProp As AeccUserDefinedProperty
'Create a user-defined parcel property classification
Set oUDPClass =
g_oAeccDb.PointUserDefinedPropertyClassifications.Add("Example")
' Add a Property to our new classification An integer using
 upper
' and lower bound limits of 10 and 20 with a default value
 of 15
Set oUDPProp = oUDPClass.UserDefinedProperties.Add("Extra
 Data", _
   "Some Extra Data", aeccUDPPropertyFieldTypeInteger,
True, False, 10, True, _
   False, 20, True, 15, Null)
```

Style

This section covers the creation of point styles and point-specific features of the point label style object. It also explains point description keys, which are used to assign styles to points read from a text file.

Creating Point Styles

A point style is a group of settings that define how a point is drawn. These settings include marker style, marker color and line type, and label color and line type. Point objects can use any of the point styles that are currently stored in the document. Styles are assigned to a point through the point's `AeccPoint.Style` property. Existing point styles are stored in the document's `AeccDocument.PointStyles` property.

You can also create custom styles and add them to the document's collection of point styles. First, add a new style to the document's list of styles using the `AeccDocument.PointStyles.Add` method. This method returns a new style object that is set with all the properties of the default style. You can then make the changes to the style object you require.

This sample creates a new points style, adjusts the style settings, and the assigns the style to point "Point1":

```
' Create the style objects to use.
Dim oPointStyles As AeccPointStyles
Dim oPointStyle As AeccPointStyle

Set oPointStyles = oDocument.PointStyles

' Add the style to the document's collection of point
styles.
Set oPointStyle = oPointStyles.Add("Sample Point Style")

' This style substitutes the normal point marker
' with a dot with a circle around it.
oPointStyle.MarkerType = aeccUseCustomMarker
oPointStyle.CustomMarkerStyle = aeccCustomMarkerDot
oPointStyle.CustomMarkerSuperimposeStyle = _
  aeccCustomMarkerSuperimposeCircle

' Now set the point to use this style.
oPoint1.Style = oPointStyle
```

Creating Point Label Styles

Any text labels or graphical markers displayed at the point location are set by assigning a label style object to the `AeccPoint.LabelStyle` property. The collection of these label styles is accessed through the `AeccDocument.PointLabelStyle` property.

Point label styles can use the following property fields in the contents of any text components:

Valid property fields for AeccLabelStyleTextComponent.Contents
<[Name(CP)]>
<[Point Number]>
<[Northing(Uft|P4|RN|AP|Sn|OF)]>
<[Easting(Uft|P4|RN|AP|Sn|OF)]>
<[Raw Description(CP)]>
<[Full Description(CP)]>
<[Point Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[Latitude(Udeg|FDMSdSp|P6|RN|DPSn|CU|AP|OF)]>
<[Longitude(Udeg|FDMSdSp|P6|RN|DPSn|CU|AP|OF)]>
<[Grid Northing(Uft|P4|RN|AP|Sn|OF)]>
<[Grid Easting(Uft|P4|RN|AP|Sn|OF)]>
<[Scale Factor(P3|RN|AP|OF)]>
<[Convergence(Udeg|FDMSdSp|P6|RN|AP|OF)]>

Valid property fields for AeccLabelStyleTextComponent.Contents
<[Survey Point]>

Label styles are described in detail in the chapter 1 section Label Styles (page 211).

Using Point Description Keys

Point description keys are a method for attaching style, label style, and orientation to point locations in a drawing - possibly imported from a text file which lacks such information. Keys are objects of type `AeccPointDescriptionKey`. The `AeccPointDescriptionKey.Name` property is a pattern matching code. If any new points are created with a description that matches the name of an existing key, the point is assigned all the settings of that key.

The wildcards “?” and “*” are allowed in the name. Keys can contain either constant scale or rotation values for points or can assign orientation values depending on parameters passed through the description string. Point description keys are held in sets, objects of type `AeccPointDescriptionKeySet`. The collection of all sets in a document are held in the document’s `AeccDocument.PointDescriptionKeySets` property.

```
' Create a key set in the document's collection
' of sets.
Dim oPointDescriptionKeySet As AeccPointDescriptionKeySet
Set oPointDescriptionKeySet = _
  oDocument.PointDescriptionKeySets.Add("Sample Key Set")

' Create a new key in the set we just made. Match with
' any description beginning with "SMP".
Dim oPointDescriptionKey As AeccPointDescriptionKey
Set oPointDescriptionKey =
oPointDescriptionKeySet.Add("SAMP*")

' Assign chosen styles and label styles to the key.
oPointDescriptionKey.PointStyle = oPointStyle
oPointDescriptionKey.OverridePointStyle = True
oPointDescriptionKey.PointLabelStyle = oLabelStyle
oPointDescriptionKey.OverridePointLabelStyle = True
```

```
' Turn off the scale override, and instead scale
' to whatever is passed as the first parameter.
oPointDescriptionKey.OverrideScaleParameter = False
oPointDescriptionKey.UseDrawingScale = False
oPointDescriptionKey.ScaleParameter = 1
oPointDescriptionKey.ScaleXY = True

' And turn on the rotation override, and rotate
' all points using this key 45 degrees clockwise.
oPointDescriptionKey.OverrideFixedRotation = True
oPointDescriptionKey.FixedRotation = 0.785398163 ' radians
oPointDescriptionKey.ClockwiseRotation = True
```

The following is the contents of a text file in “PENZD (comma delimited)” format with point information, a description, and parameter. This creates two points using the previously defined SAMP* description key, resulting in point markers four times normal size.

```
2000,3700.0,4900.0,150.0,SAMPLE 4
2001,3750.0,4950.0,150.0,SAMPLE 4
```

When a text file is loaded using `Points.ImportPoints`, the first alphanumeric element in a point’s description is compared to the names of all point description keys. If a match is found, the point’s settings are adjusted to match the description key. Any parameters to pass to the key are added after the description, separated by spaces. If using parameters, use a comma delimited file format or else any parameters will be ignored. This process only takes place when points are read from a file - after a point is created, setting the `AeccPoint.RawDescription` property does nothing to change the point’s style.

Point Groups

This section explains the creation and use of point groups, which is a named subset of the points in a document.

Creating Point Groups

A point group is a collection that contains a subset of the points in a document. A collection of all point groups is held in a document‘s `AeccDocument.PointGroups` property. Add a new point group by using the

`AeccDocument.PointGroups.Add` method and specifying a unique identifying string name. A new empty point group is returned.

```
' Get the collection of all point groups from the document.
Dim oPtGroups As AeccPointGroups
Dim oPtGroup As AeccPointGroup
Set oPtGroups = oAeccDocument.PointGroups

' Add our group to the collection of groups.
Set oPtGroup = oPtGroups.Add("Sample point group")
```

Adding Points to a Point Group Using QueryBuilder

Points can be placed into a point group by using the QueryBuilder, which is a mechanism for selecting from among all the points in the document. The `AeccPointGroup.QueryBuilder` property is an object of type `AeccPointGroupQueryBuilder`, and contains many different properties that allow including or excluding points based on a specific criteria. These properties are:

`InlcudeElevations`	`ExcludeElevations`
`IncludeFullDescriptions`	`ExcludeFullDescriptions`
`IncludeNames`	`ExcludeNames`
`IncludeNumbers`	`ExcludeNumbers`
`IncludeRawDescriptions`	`ExcludeRawDescriptions`
`IncludePointGroups`	

Each of these properties is a string that describes the selection criteria used. As many properties may be used as needed to make your selection. Any property left blank has no affect on the selection.

The properties that query string properties of points (`FullDescription`, `Name`, `RawDescription`) consist of a comma-separated list of possible strings to match against. Each element in that list may contain the wildcards “?” and “*”. The properties that deal with number properties of points consist of a comma delineated list of specific numbers or ranges of numbers (two values separated by a hyphen, such as “100-200”). The properties that deal with elevation consist of a comma delineated list of specific elevations, ranges of elevation, upper limits (a less-than symbol followed by a value, such as “<500”), or lower limits (a greater-than symbol followed by a value, such as “>100”).

```
' Add points to the point group using the QueryBuilder.

' Add point 1 and point 2 to the point group.
oPtGroup.QueryBuilder.IncludeNames = "po?nt1,point2"
' Add point 3 to the point group by using its number.
oPtGroup.QueryBuilder.IncludeNumbers = "3"
' You can also use wildcards. The following would
' select all points with a name beginning with "poi".
oPtGroup.QueryBuilder.IncludeNames = "poi*"
' Exclude those points with an elevation above 300.
oPtGroup.QueryBuilder.ExcludeElevations = ">300"
' And include those points with identification numbers
' 100 through 200 inclusive, and point number 206.
oPtGroup.QueryBuilder.IncludeNumbers = "100-200,206"
```

To create a group that contains every point in the document, set the boolean `AeccPointGroup.InlcudeAllPoints` property to `True`.

Using Point Groups

Once a point group has been created, you can perform actions upon all the points in that group in a single operation. You can override point elevations, descriptions, styles, and label styles.

```
' Check to see if a particular point exists in the group.
If (oPtGroup.ContainsPoint(oPoint1.Number) = False) Then
    Debug.Print oPoint1.Name & " is not in the point group."
End If

' Set the elevation of all the points in the group to 100.
oPtGroup.Elevation = 100
oPtGroup.OverrideElevation = True
```

Point groups can also be used to define or modify a TIN surface. The `AeccTinSurface.PointGroups` property is a collection of point groups. When a point group is added to the collection, every point in the point group is added to the TIN surface.

```
' oTinSurf is a valid object of type AeccTinSurface.
' oPointGroup is a valid object of type AeccPointGroup.
oTinSurf.PointGroups.Add oPointGroup
```

Sample Program

SurfacePointsSample.dvb

<installation-directory>\Sample\Civil 3D API\Vba\SurfacePoints\SurfacePointsSample.dvb

The sample code from this section can be found in context in the SurfacePointsSample.dvb program. The Points module contains functions that load points from a text file, create points manually, use description keys, deal with point groups, and use points to modify a surface. The PointStyles module demonstrates the creation of a point style and a point label style.

Surfaces in COM

Object Hierarchy

Using the Surfaces Collection

All surfaces in a drawing are located in the `AeccDocument.Surfaces` collection. Each surface in the collection can be accessed through the `AeccSurfaces.Item` method, which takes either an integer index or the string name of the surface. The `AeccSurfaces.Item` method returns a generic reference of type `AeccSurface`, so you need to check the `AeccSurface.Type` property to actually determine what kind of surface it is.

This sample examines each surface in the drawing and reports what kind of surface it is:

```
Dim oSurface As AeccSurface
Dim i As Integer

For i = 0 To oAeccDocument.Surfaces.Count - 1
    Set oSurface = oAeccDocument.Surfaces.Item(i)
    Select Case (oSurface.Type)
        Case aecckGridSurface:
            Dim oGridSurface As AeccGridSurface
            Set oGridSurface = oSurface
            Debug.Print oGridSurface.Name & ": Grid"
        Case aecckTinSurface:
            Dim oTinSurface As AeccTinSurface
            Set oTinSurface = oSurface
            Debug.Print oTinSurface.Name & ": TIN"
        Case aecckGridVolumeSurface:
            Dim oGridVolume As AeccGridVolumeSurface
            Set oGridVolume = oSurface
            Debug.Print oGridVolume.Name & ": Grid Volume"
        Case aecckTinVolumeSurface:
            Dim oTinVolume As AeccTinVolumeSurface
            Set oTinVolume = oSurface
            Debug.Print oTinVolume.Name & ": TIN Valume"
    End Select
Next i
```

Creating a Surface

This section covers the various methods for creating surfaces. Loading surfaces from LandXML files, .TIN files, or DEM files is explained. It also demonstrates creating new TIN, grid, and volume surfaces.

Creating a Surface From a LandXML File

A surface saved as a LandXML file can be loaded using the `AeccSurfaces.ImportXML` method. The file also describes the kind of surface to be created, so you do not need to know beforehand. After the surface has been loaded, you can examine the `AeccSurface.Type` property and assign the generic surface reference to a more specific type.

```
Dim oSurface As AeccSurface
Dim sFileName As String
sFileName = "C:\My Documents\EG.xml"
Set oSurface = oAeccDocument.Surfaces.ImportXML(sFileName)

Dim oTinSurface as AeccTinSurface
If (oSurface.Type = aecckTinSurface) Then
    Set oTinSurface = oSurface
End If
```

Creating a TIN Surface

Creating a Surface From a .tin File

The `AeccSurfaces.ImportTIN` method can create a new TIN surface from a binary *.tin* file.

```
Dim oTinSurface As AeccTinSurface
Dim sFileName As String
sFileName = "C:\My Documents\EG.tin"
oTinSurface = oAeccDocument.Surfaces.ImportTIN(sFileName)
```

Creating a Surface Using AddTinSurface

You can also create empty TIN surfaces by adding to the document's collection of surfaces through the `AeccSurfaces.AddTinSurface` method. This method requires preparing an object of type `AeccTinCreationData`. It is important to specify every property of the `AeccTinCreationData` object to avoid errors.

```
' Create a blank surface using the first layer in the
' document's collection of layers and the first
' surface style in the document's collection of
' surface styles.
Dim oTinSurface As AeccTinSurface
Dim oTinData As New AeccTinCreationData

oTinData.Name = "EG"
oTinData.Description = "Sample TIN Surface"
oTinData.Layer = oAeccDocument.Layers.Item(0).Name
oTinData.BaseLayer = oAeccDocument.Layers.Item(0).Name
oTinData.Style = oAeccDocument.SurfaceStyles.Item(0).Name
Set oTinSurface = oAeccDocument.Surfaces
  .AddTinSurface(oTinData)
```

Creating a Grid Surface

Creating a Surface From a DEM File

A grid surface can be generated from a DEM file using the `AeccSurfaces.ImportDEM` method.

NOTE

The conversion process between the raster information and a surface can be slow. Be sure to indicate this to the user.

```
Dim oGridSurface As AeccGridSurface
Dim sFileName As String
sFileName = "C:\My Documents\file.dem"
oGridSurface = oAeccDocument.Surfaces.ImportDEM(sFileName)
```

Creating a Surface From AddGridSurface

A blank grid surface can be created using the `AeccSurfaces.AddGridSurface` method. Before you can use this method you need to prepare an object of type `AeccGridCreationData`, which describes the nature of the surface to be created. It is important to specify every property of the `AeccGridCreationData` object to avoid errors. Units for `XSpacing`, `YSpacing` and `Orientation` are specified in the ambient settings.

```
Dim oGridSurface As AeccGridSurface
Dim oGridCreationData As New AeccGridCreationData

oGridData.Name = "Sample Grid Surface"
oGridData.Description = "Grid Surface"
oGridData.BaseLayer = oAeccDocument.Layers.Item(0).Name
oGridData.Layer = oAeccDocument.Layers.Item(0).Name
oGridData.Orientation = 0#
oGridData.XSpacing = 100#
oGridData.YSpacing = 100#
oGridData.Style = oAeccDocument.SurfaceStyles.Item(0).Name
Set oGridSurface = oAeccDocument.Surfaces _
  .AddGridSurface(oGridData)
```

Creating a Volume Surface

A volume surface represents the mathematical difference between two TIN surfaces or between two grid surfaces in the document. It is created using the `AeccSurfaces.AddTinVolumeSurface` or `AeccSurfaces.AddGridVolumeSurface` methods. Each of these methods require the creation of objects (`AeccGridVolumeCreationData` or `AeccTinVolumeCreationData`) that describe the new volume surface. It is important to specify every property of these objects to avoid errors. Units for `XSpacing`, `YSpacing` and `Orientation` are specified in the ambient settings.

This sample demonstrates the creation of a TIN volume surface from two existing surfaces, `oTinSurfaceL` and `oTinSurfaceH`:

```
' Get the names of the layer and style to be used.
Dim sLayerName as String
sLayerName = g_oAeccDocument.Layers.Item(0).Name
Dim sStyleName as String
```

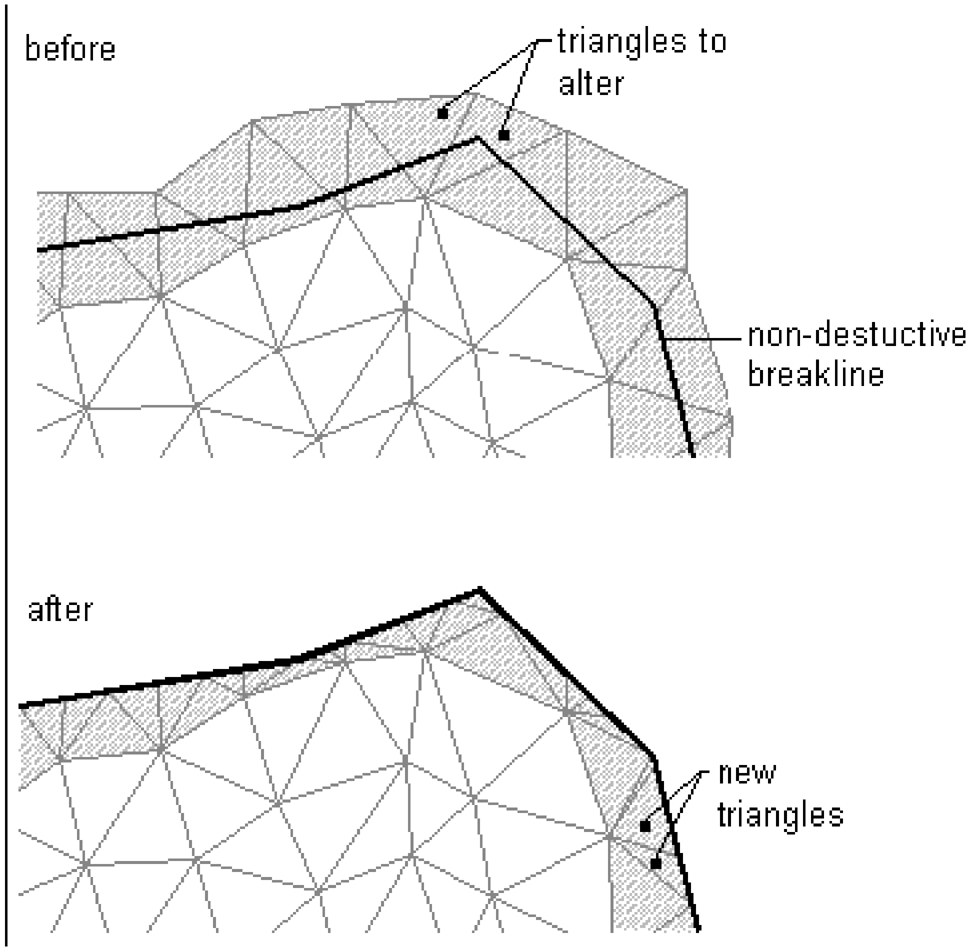
```
sStyleName = oAeccDocument.SurfaceStyles.Item(0).Name

' Create a AeccTinVolumeCreationData object and set all
its
' properties.
Dim oTinVolumeCreationData As New AeccTinVolumeCreationData
oTinVolumeCreationData.Name = "VS"
oTinVolumeCreationData.BaseLayer = sLayerName
oTinVolumeCreationData.Layer = sLayerName
Set oTinVolumeCreationData.BaseSurface = oTinSurfaceL
Set oTinVolumeCreationData.ComparisonSurface = oTinSurfaceH
oTinVolumeCreationData.Style =
oTinVolumeCreationData.Description = "Volume Surface"

' Create a new TIN volume surface.
Dim oTinVolumeSurface As AeccTinVolumeSurface
Set oTinVolumeSurface = oAeccDocument.Surfaces _
  .AddTinVolumeSurface(oTinVolumeCreationData)
```

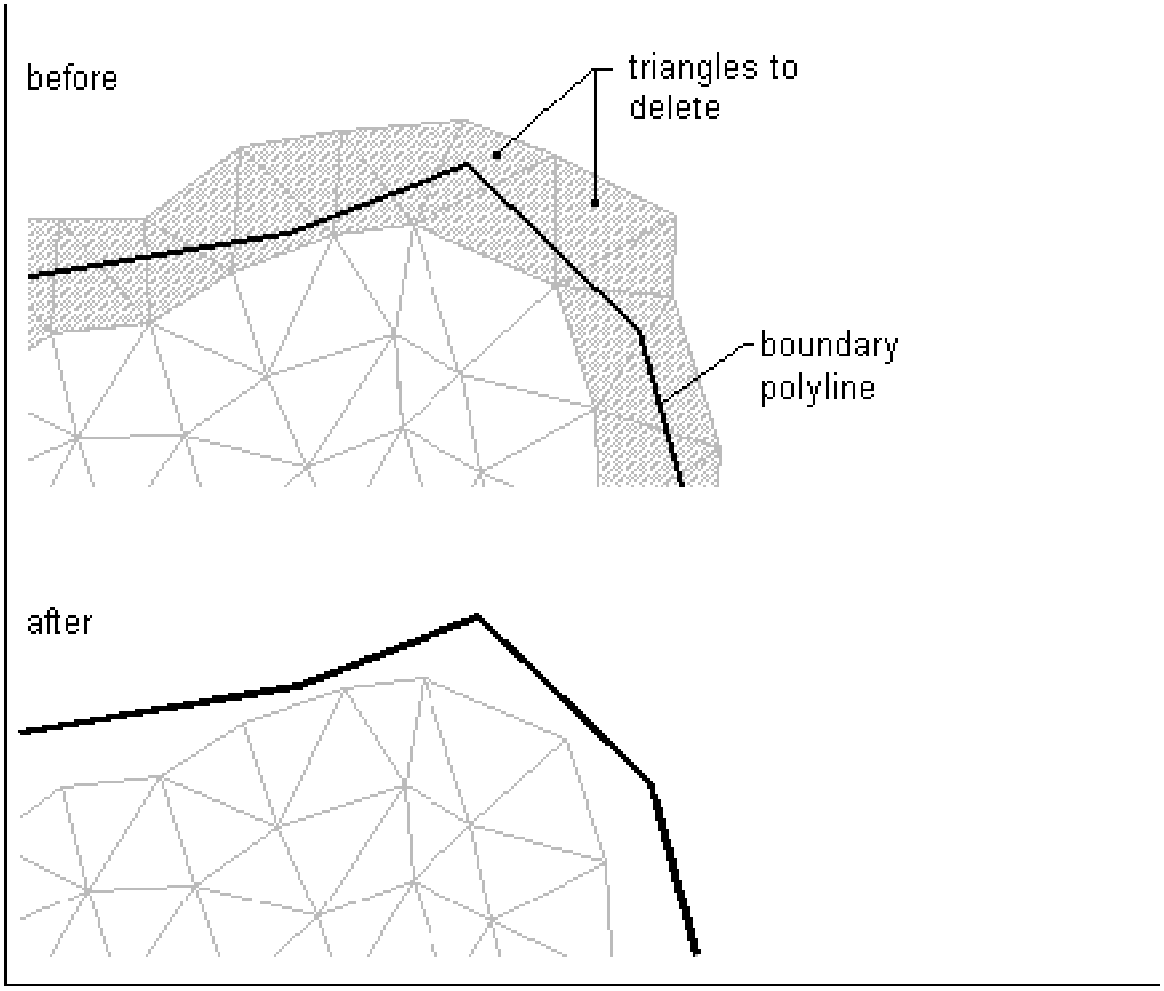
Working with Surfaces

This section covers the various methods for modifying or examining all types of surfaces. This includes adding a boundary, adding information to an existing surface from a DEM file, and using snapshots to improve surface performance.

Adding a Boundary

A boundary is a closed polyline that defines the visibility of surface triangles within it. A boundary can hide all triangles outside it, hide all triangles inside it, or show triangles inside it that would otherwise be hidden. Boundaries also change surface statistics such as area and number of triangles. Boundaries can either be “destructive” (totally hiding triangles that cross the boundary) or “non-destructive” (clipping triangle edges at the point where they cross the boundary). Normally, TIN surfaces use non-destructive boundaries, while grid surfaces can only have destructive boundaries:

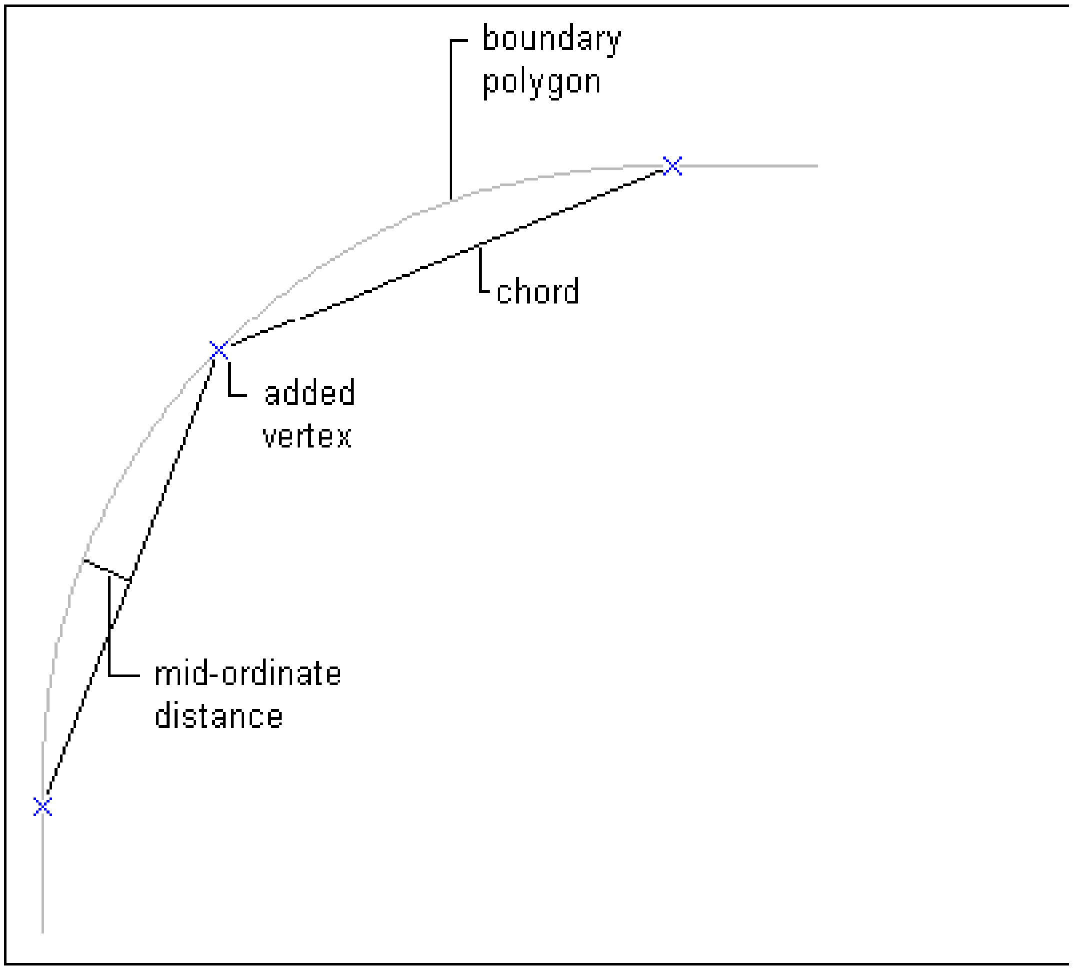

before
triangles to alter
non-destuctive breakline
after
new triangles



Non-destructive Boundary

Destructive Boundary

All boundaries applied to a surface are stored in the `AeccSurface.Boundaries` collection. The boundary itself is defined by an AutoCAD entity such as a closed POLYLINE or POLYGON. The height of the entity plays no part in how surface triangles are clipped, so you can use 2D entities. This entity can also contain curves, but the boundary always consists of lines. How these lines are tessellated is defined by the mid-ordinate distance, which is the maximum distance between a curve and the lines that are generated to approximate it:



Mid-ordinate Distance

This sample adds a square outside boundary to a surface:

```
' First we need an AutoCAD entity (in this case a polyline)
' which describes the boundary location.
Dim oPoly As AcadPolyline
Dim dPoints(0 To 11) As Double
dPoints(0) = 1000: dPoints(1) = 1000: dPoints(2) = 0
dPoints(3) = 1000: dPoints(4) = 4000: dPoints(5) = 0
dPoints(6) = 4000: dPoints(7) = 4000: dPoints(8) = 0
dPoints(9) = 4000: dPoints(10) = 1000: dPoints(11) = 0
```

```
Set oPoly = oAeccDocument.Database.ModelSpace _
  .AddPolyline(dPoints)
oPoly.Closed = True

' The name of the boundary object.
Dim sName as String
sName = "Sample Boundary"
' The third parameter describes what the boundary does
' to triangles inside it. The fourth parameter is True
' if you want non-destructive boundary or false otherwise.
' The final parameter is the mid-ordinate distance.
Dim oNewBoundary As AeccSurfaceBoundary
Set oNewBoundary = oSurface.Boundaries.Add(oPoly, sName,
_
  aeccBoundaryOuter, True, 10.5)
```

NOTE

Any operation that causes the formation of new triangles (such as adding points or breaklines to a TIN surface) may result in triangles that cross existing boundary lines. Boundaries should always be added after every other operation that adds points to a surface.

Adding Data from DEM Files

Any number of DEM files can be added to existing grid and TIN surfaces. When a DEM file is added to the `AeccGridSurface.DEMFiles` or `AeccTinSurface.DEMFiles` collection, its information is converted to an evenly spaced lattice of triangles that is added to the surface.

```
oTinSurface.DEMFiles.Add("C:\My Documents\file.dem")
```

Improving Performance By Using Snapshots

A surface is made up of all the operations that modified the surface's triangles. If you rebuild the surface, re-performing all these operations can be slow. Snapshots can improve performance by recording all the triangles in a surface. Subsequent rebuilds start from the data of the snapshot, thus saving time by not performing complicated calculations that have already been done once.

Surfaces have `CreateSnapshot`, `RebuildSnapshot`, and `RemoveSnapshot` methods. Both `CreateSnapshot` and `RebuildSnapshot` will overwrite an existing snapshot.

TIP

`RebuildSnapshot` will cause an error if the snapshot does not exist. `CreateSnapshot` and `RebuildSnapshot` can also cause errors if the surface is out-of-date.

Finding Intersections

When working with surfaces, it can be useful to determine where a vector intersects with a surface, which you can do with the surface's `IntersectPointWithSurface()` method. For example, you can determine if the top of a structure is visible from a point on the surface, or whether one point on the surface is visible from another point. This example tests whether a vector starting at (20424.7541, 20518.0409, 100) pointing in direction (0.6, 0.4, -0.5) intersects with the first surface in the drawing, and if it does, it prints out the intersection location:

```
Dim objSurf As AeccSurface
Dim varStartPt As Variant, varDir As Variant, varIntPt As
 Variant
Dim darrStart(2) As Double
Dim darrDir(2) As Double
darrStart(0) = 20424.7541
darrStart(1) = 20518.0409
darrStart(2) = 100
darrDir(0) = 0.6
darrDir(1) = 0.4
darrDir(2) = -0.5
varStartPt = darrStart
varDir = darrDir
Set objSurf = g_oAeccDoc.Surfaces(0)
varIntPt = objSurf.IntersectPointWithSurface(varStartPt,
varDir)
If UBound(varIntPt) = 2 Then
   Debug.Print varIntPt(0), varIntPt(1), varIntPt(2)
End If
```

Working with TIN Surfaces

This section covers the various methods for modifying or examining existing TIN surfaces. This includes adding new point data directly, adding breaklines, and adding contours.

Adding Point Data to a TIN Surface

There are two techniques for adding points that are unique to TIN surfaces: using point files and using point groups.

Adding Points Using Point Files

The `AeccTinSurface.PointFiles` collection contains the names of text files that contain point information. These text files must consist only of lines containing the point number, easting, northing, and elevation delineated by spaces. Except for comment lines beginning with "#", any other information will result in an error. Unlike TIN or LandXML files, text files do not contain a list of faces - the points are automatically joined into a series of triangles based on the settings of the `AeccTinSurface.DefinitionProperties` property.

```
' Add points from a .txt file to an existing TIN surface.
sFileName = "C:\My Documents\points.txt"
oTinSurface.PointFiles.Add sFileName
```

Adding Points Using Point Groups

You can manually add points to a TIN surface by adding point groups to the `AeccTinSurface.PointGroups` collection.

```
' Make a surface consisting of 30 random points. To do
' this, first add 30 points to the document's collection
' of points, then make a point group from those points.
' Finally, add that point group to the surface.
Dim i As Integer
For i = 0 To 29
    Dim pt(0 To 2) As Double
```

```
        pt(0) = Int(5000 * Rnd())
        pt(1) = Int(5000 * Rnd())
        pt(2) = Int(200 * Rnd())
        Dim oPoint As AeccPoint
        Set oPoint = oAeccDocument.Points.Add(pt)
        oPoint.Name = "Point" & CStr(i)
    Next i

    Dim oPtGroup As AeccPointGroup
    Set oPtGroup = oAeccDocument.PointtGroups.Add("Random
    group")
    ' Add all points with a name beginning with "Point"
    oPtGroup.QueryBuilder.IncludeNames = "point*"
    ' Add the point group to the surface.
    oTinSurface.PointGroups.Add oPtGroup
    oTinSurface.Update
```

You can also add points to a surface by adding contour lines. See Adding Contours to a TIN Surface (page 263).

All points that make up a surface can be retrieved from the read-only `AeccTinSurface.Points` property, which is an array of locations.

```
    ' Print the location (easting, northing, and elevation)
    ' of the 1000th point.
    Dim vLocation as Variant
    vLocation = oTinSurface.Points(1000)

    ' Now vLocation contains a 3 element array of doubles.
    Debug.Print "Easting:"; vLocation(0)
    Debug.Print "Northing:"; vLocation(1)
    Debug.Print "Elevation:"; vLocation(2)
```

Adding a Breakline to a TIN Surface

Breaklines are used to shape the triangulation of a TIN surface. Each TIN surface has a collection of breaklines contained in the `AeccTinSurface.Breaklines` property. There are different kinds of breaklines, and each is created in a slightly different way.

NOTE

For more information about breaklines, see Breaklines in the AutoCAD Civil 3D User's Guide.

Adding a Standard Breakline

A standard breakline consists of an array of 3D lines or polylines. Each line endpoint becomes a point in the surface and surface triangles around the breakline are redone. If the polyline contains curves, then the curve is tessellated based on the mid-ordinate distance parameter.

```
' Create the polyline basis for the breakline.
Dim o3DPoly as Acad3DPolyline
Dim dPoints(0 To 8) As Double
dPoints(0) = 1200: dPoints(1) = 1200: dPoints(2) = 150
dPoints(3) = 2000: dPoints(4) = 3000: dPoints(5) = 120
dPoints(6) = 3000: dPoints(7) = 2000: dPoints(8) = 100
Set o3DPoly = oAeccDocument.Database.ModelSpace _
  .Add3DPoly(dPoints)
o3DPoly.Closed = False
Dim oBreakline As AeccSurfaceBreakline
Dim vBLines As Variant
' This has to be an array, even if we only have one entity.
Dim oEntityArray(0) As AcadEntity
Set oEntityArray(0) = oAeccDocument.Database.ModelSpace _
  .Add3DPoly(dPoints)
Set oBreakline = oTinSurface.Breaklines.AddStandardBreakline
 _
  (oEntityArray, "Sample Standard Break", 1#)
```

Adding a Proximity Breakline

A proximity breakline does not add new points to a surface. Instead, the nearest surface point to each breakline endpoint is used. The triangles that make up a surface are then recomputed making sure those points are connected. A proximity breakline is created using the same fashion as standard breaklines except you call `AeccSurfaceBreaklines.AddProximityBreakline` instead of `AeccSurfaceBreaklines.AddStandardBreakline`.

```
Set oBreakline =
oTinSurface.Breaklines.AddProximityBreakline( _
  oEntityArray, _
  "Sample Proximity Break", _
  1#)
```

Adding a Non-destructive Breakline

A non-destructive breakline does not remove any triangle edges. It places new points along the breakline at each intersection with a triangle edge and new triangles are computed. Again, it is created like standard breaklines are created except you call the `AeccSurfaceBreaklines.AddNonDestructiveBreakline` method.

```
Set oBreakline = oTinSurface.Breaklines _
  .AddNonDestructiveBreakline( _
    oEntityArray, _
    "Sample Non-Destructive Break", _
    1#)
```

Adding a Wall Breakline

A wall breakline is used when the height of the surface on one side of the breakline is different than the other side. This method creates two breaklines, one for the top of the wall and one for the bottom. However, you cannot have a perfectly vertical wall in a TIN surface. The first breakline is placed along the path specified by the BreaklineEntities parameter, and the second breakline is very slightly offset to one side and raised or lowered by a relative elevation. Among the parameters of the wall breakline creation method are an array of wall elevations and an array describing to which side the height-adjusted breakline is placed. The wall at each entity endpoint is offset to the right if the value is set to `True` and to the left of the value is set to `False` where “right” and “left” refer to directions as you walk along the breakline from the start point to the end.

```
' This is an array of arrays of elevations, one array of
' elevations per entity.
Dim vElevations(0) As Variant
```

```
' These are the elevations of the wall at each endpoint in
' the polyline entity.
Dim dElevations(3) As Double
' This is an array of ooleans, one for each entity.
Dim bOffsets(0) As Boolean

dElevations(0) = 30.5: dElevations(1) = 93.3
dElevations(2) = 93.3: dElevations(3) = 46.2
vElevations(0) = dElevations
' Raise the surface at the right side of the breakline.
bOffsets(0) = True: bOffsets(1) = True
bOffsets(2) = True: bOffsets(3) = True

Set oBreakline = oTinSurf.Breaklines.AddWallBreakline _
  (oEntityArray, _
  "Sample Wall Break", _
  1#, _
  vElevations, _
  bOffsets)
```

Importing Breaklines from a File

You can import breaklines from a file in *.FLT* format, using `AeccSurfaceBreaklines.AddBreaklineFromFile()`. When you import the file, you need to specify whether to maintain the file link, or break the link:

- `aeccBreaklineFileMaintainLink`: Reads the breaklines from the FLT file when they are added and when the surface is rebuilt.
- `aeccBreaklineFileBreakLink`: All breaklines in the FLT file are copied into the surface as Add Breakline operations.

For more information about these options, see Importing Breaklines from a File in the .

This sample shows how to import breaklines from a file named *eg1.flt*, and to get the first newly created breakline:

```
Dim oBrkLine As AeccSurfaceBreakline
Set oBrkLine = brkLines.AddBreaklineFromFile("New
Breakline", "C:\eg1.flt", 10#,
aeccBreaklineFileBreakLink)(0)
```

Adding Contours to a TIN Surface

A contour is an open or closed entity that describes the altitude of the surface along the entity. Contours must have a constant altitude. The z value of the first point of the entity is used as the altitude of entire entity, no matter what is specified in the following points. Contours also have settings that can adjust the number of points added to the surface - when you create a contour, you specify a weeding distance, a weeding angle, and a distance parameter. Points in the contour are removed if the distance between the points before and after is less than the weeding distance and if the angle between the lines before and after is less than the weeding angle. Each line segment is split into equal sections with a length no greater than the supplemental distance parameter. Any curves in the entity are also tessellated according to the mid-ordinate distance, just like breaklines. The supplemental distance value has precedence over the weeding values, so it is possible that the final contour will have line segments smaller than the weeding parameters specify.

For more information about weeding and countours, see Weeding and Supplementing Factors for Contours in the .

A TIN surface has a collection of contours in the `AeccTinSurface.Contours` property. The following sample demonstrates adding a contour to a surface:

```
Dim dPoints(0 To 8) As Double ' 3 points
Dim o3DPoly As AcadPolyline

dPoints(0) = 2500: dPoints(1) = 1500: dPoints(2) = 100
dPoints(3) = 2600: dPoints(4) = 1600: dPoints(5) = 100
' It does not matter that we specify a Z value of 50. It
' is still located at an altitude of 100, just like
' the first point.
dPoints(6) = 2400: dPoints(7) = 1600: dPoints(8) = 50
Set o3DPoly = oAeccDocument.Database.ModelSpace _
  .AddPolyline(dPoints)
o3DPoly.Closed = False
Dim oEntities(0) As AcadEntity
Set oEntities(0) = o3DPoly

Dim dWeedDist as Double
Dim dWeedAngle as Double
Dim dDist as Double
Dim dMidOrdDist as Double
dWeedDist = 55.5
```

```
dWeedAngle = 0.0698 ' 0.0698 radians = 4 degrees
dDist = 85.5
dMidOrdDist = 1#
Dim oNewContour As AeccSurfaceContour
Set oNewContour = oTinSurf.Contours.Add(oEntities, _
  "Sample Contour", dWeedDist, dWeedAngle, dDist,
dMidOrdDist)
```

Extracting Contours from a TIN Surface

You can extract a contour (or contours) from a surface in a given elevation range as AutoCAD entities. This example extracts the contours between 90 and 95, and prints out the entity type for each one.

NOTE

The contours that you wish to extract must be visible in the drawing for this example to work.

```
...
Dim z As Double
Dim objSurf As AeccSurface
Set objSurf = g_oAeccDoc.Surfaces(0)
Dim varObjects As Variant
Dim objEnt As AcadEntity
Dim iCtr As Integer, iLow As Integer, iHigh As Integer
varObjects =
objSurf.ExtractContour(aeccDisplayOrientationPlan,
aeccSFMajorContours, 90, 95)
iLow = LBound(varObjects)
iHigh = UBound(varObjects)
For iCtr = iLow To iHigh
   Set objEnt = varObjects(iCtr)
   Debug.Print TypeName(objEnt)
Next iCtr
```

Surface Style

This section explains the creation and application of surface styles.

Creating a Style

All surface styles are stored in the `AeccDocument.SurfaceStyles` collection, an object of type `AeccSurfaceStyles`. To create a new style, call the `AeccSurfaceStyles.Add` method with the name of your new style. It is initially set according to the document's ambient settings.

```
Dim oSurfaceStyle As AeccSurfaceStyle
oSurfaceStyle = oDocument.SurfaceStyles.Add("New Style")
```

Changing a Surface Style

A surface style consists of different objects governing the appearance of boundaries, contours, direction analysis, elevation analysis, grids, points, slope arrows, triangles, and watershed analysis. Usually a single style only displays some of these objects. When initially created, a style is set according to the document's ambient settings and may show some unwanted style elements. Always set the visibility properties of all style elements to ensure the style behaves as you expect.

```
' Change the style so that it displays surface triangles,
' major contour lines, and any boundaries along the outside
' edge, but nothing else.
oSurfaceStyle.TriangleStyle.DisplayStylePlan.Visible = True
oSurfaceStyle.BoundaryStyle.DisplayExteriorBoundaries =
True
oSurfaceStyle.BoundaryStyle.DisplayStylePlan.Visible = True
oSurfaceStyle.ContourStyle.MajorContourDisplayStylePlan _
  .Visible = True

oSurfaceStyle.PointStyle.DisplayStylePlan.Visible = False
oSurfaceStyle.BoundaryStyle.DisplayInteriorBoundaries =
False
oSurfaceStyle.ContourStyle.MinorContourDisplayStylePlan _
  .Visible = False
oSurfaceStyle.ContourStyle.UserContourDisplayStylePlan _
  .Visible = False
oSurfaceStyle.GridStyle.DisplayStylePlan.Visible = False
oSurfaceStyle.DirectionStyle.DisplayStylePlan.Visible =
False
```

```
oSurfaceStyle.ElevationStyle.DisplayStylePlan.Visible =
False
oSurfaceStyle.SlopeStyle.DisplayStylePlan.Visible = False
oSurfaceStyle.SlopeArrowStyle.DisplayStylePlan.Visible =
False
oSurfaceStyle.WatershedStyle.DisplayStylePlan.Visible =
False

' This must be repeated for all Model display styles as
well.
```

Assigning a Style to a Surface

Set the `AeccSurface.Style` property to the name of the style you wish to use.

```
' The surface is displayed according to the
' oSurfaceStyle style we have just created.
oSurface.Style = oSurfaceStyle.Name
```

Performing Surface Analysis

This section shows you how to perform an elevation analysis and a watershed analysis.

Creating an Elevation Analysis

An elevation analysis creates a 2-dimensional projection of a surface and then add bands of color indicating ranges of altitude. The analysis is managed by an object of type `AeccSurfaceAnalysisElevation`, located in the `AeccSurface.SurfaceAnalysis.ElevationAnalysis` property. This object contains a method for creating elevation regions and a read-only collection containing these regions. The method, `CalculateElevationRegions`, creates a series of contiguous regions each representing a portion of the surface’s total elevation. Each time it is called it discards all existing elevation regions for the surface and creates a new collection of regions. The collection, `ElevationRegions`, lets you modify the color, minimum elevation, and maximum elevation of each region. Always check the number of regions through the `AeccSurfaceAnalysisElevation.ElevationRegions.Count` property as `CalculateElevationRegions` may create fewer regions than specified by the first parameter.

`CalculateElevationRegions` creates regions according to the statistical method specified in the `AeccSurfaceStyle.ElevationStyle.GroupValuesBy` property. The elevation style object also contains other means of modifying how elevation analyses are made, such as using one of the preset color schemes.

This sample creates an elevation analysis for a surface using shades of green:

```
Dim oSurfaceAnalysisElevation As
AeccSurfaceAnalysisElevation
Set oSurfaceAnalysisElevation = oSurface.SurfaceAnalysis
_
  .ElevationAnalysis
Dim oElevationRegions As AeccElevationRegions
Set oElevationRegions = oSurfaceAnalysisElevation _
  .CalculateElevationRegions(6, False)
' See exactly how many regions were created.
Debug.Print oSurfaceAnalysisElevation.ElevationRegions.Count

oElevationRegions.Item(0).Color = 80 ' bright green
oElevationRegions.Item(1).Color = 82
oElevationRegions.Item(2).Color = 84
oElevationRegions.Item(3).Color = 86
oElevationRegions.Item(4).Color = 88 ' dark green

' Adjust the range of one of the regions.
oElevationRegions.Item(2).MaximumElevation = _
  oElevationRegions.Item(2).MaximumElevation - 5
```

Many elevation analysis features can be modified through the surface style - see the Surface Style (page 264) section. For example, you can choose from among a number of pre-set color schemes.

Creating a Watershed Analysis

A watershed analysis predicts how water will flow over and off a surface. The analysis is managed by an object of type `AeccSurfaceAnalysisWatershed` held in the `AeccSurface.SurfaceAnalysis.WatershedAnalysis` property. The analysis is created by calling the `AeccSurfaceAnalysisWatershed.CalculateWatersheds` method. This splits the surface into separate regions, each with its own drain target or targets. The set of all these regions are held in the `AeccSurfaceAnalysisWatershed.WatershedRegions` collection.

You have some control over how the regions are split. If the boolean `AeccSurfaceAnalysisWatershed.MergeAdjacentBoundaries` property is set to `True`, then regions along the boundary are merged into one region if their boundary points or segments touch. If a depression on the surface has a minimum average depth smaller than the value of the `AeccSurfaceAnalysisWatershed.MergeDepression` property, then the depression does not become its own region and is combined with the region it drains into.

```
oSurface.SurfaceAnalysis.WatershedAnalysis _
  .MergeAdjacentBoundaries = True
oSurface.SurfaceAnalysis.WatershedAnalysis _
  .MergeDepression = 10.65
```

Types of Watershed Regions

Depending on the nature of the drain target, each watershed region is a different type derived from `AeccWatershedRegion`. (For more information about watershed region types, see Types of Watersheds in the). By checking the `Type` property of each object in the `AeccSurfaceAnalysisWatershed.WatershedRegions` collection, you can then determine the specific type of each region.

```
' Compute water drainage over the surface.
oSurface.SurfaceAnalysis.WatershedAnalysis _
  .CalculateWatersheds

' Extract information from each watershed region.
' Loop through all the regions in the WatershedRegions
' collection. For each region, determine its
' specific type. Once we cast each region object to this
' specific type, we can learn how water drains over the
' surface.
Dim oWSAnalysis As AeccWatershedRegions
Set oWSAnalysis = oSurface.SurfaceAnalysis.WatershedAnalysis
 _
  .WatershedRegions

Dim i as Integer
```

```
For i = 0 To oWSAnalysis.Count - 1
   Select Case (oWSAnalysis.Item(i).Type)
   Case aeccWatershedBoundaryPoint
      Dim oWSPoint As AeccWatershedRegionBoundaryPoint
      Set oWSPoint = oWSAnalysis.Item(i)

   Case aeccWatershedBoundarySegment
      Dim oWSSegment As AeccWatershedRegionBoundarySegment
      Set oWSSegment = oWSAnalysis.Item(i)

   Case aeccWatershedDepression
      Dim oWSDepression As AeccWatershedRegionDepression
      Set oWSDepression = oWSAnalysis.Item(i)

   Case aeccWatershedFlat
      Dim oWSFlat As AeccWatershedRegionFlat
      Set oWSFlat = oWSAnalysis.Item(i)

   Case aeccWatershedMultiDrain
      Dim oWSMulti As AeccWatershedRegionMultiRegionDrain
      Set oWSMulti = oWSAnalysis.Item(i)

   Case aeccWatershedMultiDrainNotch
      Dim oWSNotch As
AeccWatershedRegionMultiRegionDrainNotch
      Set oWSNotch = oWSAnalysis.Item(i)
   Case Else 'aeccWatershedUnknownSink

   End Select
Next i
```

Objects derived from `AeccWatershedRegion` have other common features. They all have an identification number in the `AeccWatershedRegion.Id` property. They also have a `AeccWatershedRegion.Boundary` property, which contains a 2-dimensional array containing the points of a closed polygon enclosing the region.

```
Dim vBound As Variant
vBound = oWSAnalysis.Item(i).BoundaryLine
For j = 0 To UBound(vBound)
   ' Print the X, Y and Z coordinates of a border point.
   Debug.Print vBound(j, 0), vBound(j, 1), vBound(j, 2)
Next j
```

Boundary Point Regions

In a region of type `AeccWatershedBoundaryPoint`, water reaches the boundary of a surface at a single point. The X, Y, and Z coordinates of this point are held in a variant array in the `AeccWatershedBoundaryPoint.BoundaryDrainPoint` property.

```
Dim oWSPoint As AeccWatershedRegionBoundaryPoint
Set oWSPoint = oWSAnalysis.Item(i)
Dim vDrainPoint As Variant
vDrainPoint = oWSPoint.BoundaryDrainPoint
Debug.Print "This region drains to point: " & vDrainPoint(0)
 _
  & ", " & vDrainPoint(1) & ", " & vDrainPoint(2)
```

Boundary Segment Regions

Regions of type `aeccWatershedBoundarySegment` represent areas where water flows out of a surface along a series of line segments. The end points of these line segments are held in a 2-dimensional array of doubles in the `aeccWatershedBoundarySegment.BoundaryDrainSegment` property. The first dimension of this array represents each point and the second dimension are the X, Y, and Z coordinates of the points.

```
Dim oWSSegment As AeccWatershedRegionBoundarySegment
Set oWSBoundarySegment = oWSAnalysis.Item(i)
Dim vDrainSegments as Variant
vDrainSegments = oWSBoundarySegment.BoundaryDrainSegment

Dim j as Integer
Debug.Print "This region drains through the following"
Debug.Print "line segments:"
For j = 0 To UBound(vDrainSegments, 1) - 1
   Debug.Print vDrainSegments(j, 0) & ", " _
     & vDrainSegments(j, 1) & ", " _
     & vDrainSegments(j, 2) & "  to  ";
   Debug.Print vDrainSegments(j + 1, 0) & ", " _
     & vDrainSegments(j + 1, 1) _
     & ", " & vDrainSegments(j + 1, 2)
Next j
```

Depression Regions

A region of type `aeccWatershedDepression` represents an area of the surface that water does not normally leave. It is possible for the depression to fill and then drain into other regions. The lowest points on the region edge where this overflow may take place and the regions that the water drains into are kept in the `aeccWatershedDepression.Drains` collection.

```
Dim oWSDepression As AeccWatershedRegionDepression
Set oWSDepression = oWSAnalysis.Item(i)
Dim oDrains As AeccWatershedDrains
Set oDrains = oWSDepression.Drains

For j = 0 To oDrains.Count - 1
   ' See what kind of drain targets we have.
   If (UBound(oDrains.Item(j).Targets) = -1) Then
      ' This depression drains outside the surface.
      Debug.Print "Drain through point: " & _
        oDrains.Item(j).Location(0) & ", " & _
        oDrains.Item(j).Location(1) & ", " & _
      oDrains.Item(j).Location(2) & _
        " to the surface boundary."
   Else
      ' This depression can drain into other regions.
      Dim lTargets() As Long
      lTargets = oDrains.Item(j).Targets
      sTargets = CStr(lTargets(0))
      Dim k as Integer
      For k = 1 To UBound(lTargets)
         sTargets = sTargets & ", " & CStr(lTargets(k))
      Next k
      Debug.Print "Drain through point: " & _
        oDrains.Item(j).Location(0) & ", " & _
        oDrains.Item(j).Location(1) & ", " & _
        oDrains.Item(j).Location(2) & _
        " into the following regions: " & sTargets
   Endif
Next j
```

Flat Regions

A flat area that only drains into one region is combined into that region. If a flat surface drains into multiple regions, then it is created as a separate region of type `AeccWatershedRegionFlat`. The only feature of flat regions is an array of all drain targets.

```
Dim oWSFlat As AeccWatershedRegionFlat
Set oWSFlat = oWSAnalysis.Item(i)

varDrainsInto = oWSFlat.DrainsInto
sTargets = CStr(varDrainsInto(0))
For k = 1 To UBound(varDrainsInto)
   sTargets = sTargets & ", " & CStr(varDrainsInto(k))
Next k
Debug.Print "This region drains into regions " & sTargets
```

Multiple Drain Regions (Point)

A region of the surface may drain through a point into many different regions. Such regions are represented by an object of type `AeccWatershedRegionMultiRegionDrain`. These regions have properties containing the point water drains through and a collection of all regions into which water flows.

```
Dim oWSMulti As AeccWatershedRegionMultiRegionDrain
Set oWSMulti = oWSAnalysis.Item(i)

' vDrainPoint is a single point, like BoundaryPoint
vDrainPoint = oWSMulti.DrainPoint
' varDrainsInto is an array of variants, each a region ID.
varDrainsInto = oWSMulti.DrainsInto
sTargets = CStr(varDrainsInto(0))
For k = 1 To UBound(varDrainsInto)
   sTargets = sTargets & ", " & CStr(varDrainsInto(k))
Next k

Debug.Print "This region drains to point: " & vDrainPoint(0)
 _
  & ", " & vDrainPoint(1) & ", " & vDrainPoint(2) _
  & " and into the following regions: " & sTargets
```

Multiple Drain Regions (Notch)

A region can also drain into many other regions through a series of line segments. These regions are represented by an object of type

`AeccWatershedRegionMultiRegionDrainNotch` and they keep both a list of all regions this region drains into and a list of all segments this region drains through.

```
Dim oWSNotch As AeccWatershedRegionMultiRegionDrainNotch
Set oWSNotch = oWSAnalysis.Item(i)
' vDrainSegments is a 2-dimensional array, like
BoundarySegment.
Dim vDrainSegments As Variant
vDrainSegments = oWSNotch.DrainSegment
' varDrainsInto is an array of region IDs.
Dim varDrainsInto As Variant
varDrainsInto = oWSNotch.DrainsInto

sTargets = CStr(varDrainsInto(0))
For k = 1 To UBound(varDrainsInto)
   sTargets = sTargets & ", " & CStr(varDrainsInto(k))
Next k
Debug.Print "This region drains through these segments: "
For j = 0 To UBound(vDrainSegments, 1) - 1
   Debug.Print vDrainSegments(j, 0) & ", " _
     & vDrainSegments(j, 1) & ", " _
     & vDrainSegments(j, 2) & "  to      ";
   Debug.Print vDrainSegments(j + 1, 0) & ", " _
     & vDrainSegments(j + 1, 1) _
     & ", " & vDrainSegments(j + 1, 2)
Next j

' Display each region this drains into.
Debug.Print "and into the following regions: " & sTargets
```

Sample Programs

SurfacePointsSample.dvb

<installation-directory>\Sample\Civil 3D API\Vba\SurfacePoints\SurfacePointsSample.dvb

This sample program demonstrates the creation of surfaces using point data loaded from a file and through use of point groups. Surface styles are created and applied.

SurfaceOperations.dvb

<installation-directory>\Sample\Civil 3D API\Vba\SurfaceOperations\SurfaceOperationsSample.dvb

This sample demonstrates the elevation analysis and watershed analysis features. The watershed analysis includes a full report of all watershed regions. Breaklines of all kinds, contours, and borders are applied to the surfaces as well.

CorridorSample.dvb

<installation-directory>\Sample\Civil 3D API\Vba\Corridor\CorridorSample.dvb

A TIN volume surface is created between the corridor datum surface and the ground surface. Cut and fill information is obtained from this volume surface.

Sites and Parcels in COM

Object Hierarchy

Sites

This section explains the creation and use of sites, which are used as containers for parcels and alignments (for information about alignments, see Chapter 6: Alignments (page 284)).

Creating Sites

All sites in a document are held in the `AeccDocument.Sites` collection, an object of type `AeccSites`. The `AeccSites.Add` method creates a new empty site with the specified name.

```
' Create a new site.
Dim oSites As AeccSites
Set oSites = oAeccDocument.Sites
Dim oSite As AeccSite
Set oSite = oSites.Add("Sample Site")
```

Using Sites

A site represents a distinct group of alignments and parcels. Besides containing collections of parcels and alignments, the `AeccSite` object also contains properties describing how the site objects are numbered and displayed.

```
' NextAutoCounterParcel sets the starting identification
' number for newly created parcels.  The first parcel
' created from parcel segments will be 300, the next 301,
' and so on.
oSite.NextAutoCounterParcel = 300
```

Parcels

This section covers the creation and use of parcels. Parcel segments, parcel loops, and parcel styles and label styles are also explained.

Creating Parcels with Parcel Segments

While a site contains a collection of parcels, this collection has no `Add` method. Instead, parcels are automatically generated from the parcel segments added to the `AeccSite.ParcelSegments` collection. A parcel segment is a 2-dimensional line, curve, or AutoCAD entity. Once a closed structure can be formed from the segments in the site, a parcel is automatically formed. Each additional parcel segment that forms new closed structures creates additional parcels. This may affect the shape of existing parcels - if an existing parcel is bisected by a new segment, the existing parcel is reduced in size and a new parcel is formed.

```
Dim oSegments as AeccParcelSegments
Set oSegments = oSite.ParcelSegments

' Parcel 1
Call oSegments.AddLine(0, 0, 0, 200)
Call oSegments.AddCurve(0, 200, -0.5, 200, 200)
Call oSegments.AddLine(200, 200, 200, 0)
Call oSegments.AddLine(200, 0, 0, 0)

' Parcel 2
Call oSegments.AddCurve2(200, 200, 330, 240, 400, 200)
Call oSegments.AddLine(400, 200, 400, 0)

' This will complete parcel 2, as well as form parcel 3.
Dim oPolyline As AcadPolyline
Dim dPoints(0 To 8) As Double
dPoints(0) = 400: dPoints(1) = 0: dPoints(2) = 0
dPoints(3) = 325: dPoints(4) = 25: dPoints(5) = 0
dPoints(6) = 200: dPoints(7) = 0: dPoints(8) = 0
Set oPolyline = oAeccDocument.Database.ModelSpace_
  .AddPolyline(dPoints)
oPolyline.Closed = True
' Passing True as the second parameter deletes the
' polyline entity once the parcel segment has been created.
Call oSegments.AddFromEntity(oPolyline, True)
```

About Parcel Segments

Each parcel segment is a collection of parcel segment elements, which are represented by objects derived from the `AeccParcelSegmentElement` base class. A segment element is an undivided part of a segment that can be used to create a parcel. When an element is intersected by another parcel segment, the element is split into two contiguous elements:

```
Dim oSegments as AeccParcelSegments
Set oSegments = oSite.ParcelSegments

Dim oSegment1 as AeccParcelSegment

' Segment1 consists of 1 element, a line with endpoints
' at 500,100 to 600,100
Set oSegment1 = oSegments.AddLine(500, 100, 600, 100)
' We can tell this by looking at the number of elements:
Debug.Print oSegment1.Count ' returns 1
' If we cross the segment element with another segment,
' then the elements get split.
Call oSegments.AddLine(550, 150, 550, 50)

Debug.Print oSegment1.Count ' returns 2
```

The `AeccParcelSegment.Item` method returns each element as an object of type `AeccParcelSegmentElement`. This object has no `Type` property, so to determine what kind of element it represents you need to directly check the object type with the `TypeOf` operator:

```
' Loop through all elements of the parcel segment "oSegment"

Dim i as Integer
For i = 0 to oSegment.Count - 1
    Dim oElement As AeccParcelSegmentElement
    Set oElement = oSegment.Item(i)

    Debug.Print "Element " & i & ": " _
      & oElement.StartX & "," & oElement.StartY & " to "
_
      & oElement.EndX & ", " & oElement.EndY

    If (TypeOf oElement Is AeccParcelSegmentLine) Then
```

```
            Dim oSegmentLine As AeccParcelSegmentLine
            Set oSegmentLine = oElement
            Debug.Print " is a line. "
       ElseIf (TypeOf oElement Is AeccParcelSegmentCurve) Then
            Dim oSegmentCurve As AeccParcelSegmentCurve
            Set oSegmentCurve = oElement
            Debug.Print " is a curve with a radius of:" _
              & oSegmentCurve.Radius
        End If
    Next i
```

Determining Parcel Loops

You can determine which parcel segment elements make up a parcel by using the `AeccParcel.ParcelLoops` collection. This collection stores objects of type `AeccParcelSegmentElement`. Each parcel segment element contains a reference to the parent segment, so you can also determine which segments are used to create a parcel.

```
' Loop through all elements used to make parcel "oParcel"

Dim i as Integer
For i = 0 to oParcel.ParcelLoops.Count - 1
    Dim oElement As AeccParcelSegmentElement
    Set oElement = oParcel.ParcelLoops.Item(i)

    Debug.Print "Element " & i _
      & " of segment " & oElement.ParcelSegment.Name & ":
 " _
      & oElement.StartX & "," & oElement.StartY & " to "
_
      & oElement.EndX & ", " & oElement.EndY

    If (TypeOf oElement Is AeccParcelSegmentLine) Then
        Dim oSegmentLine As AeccParcelSegmentLine
        Set oSegmentLine = oElement
        Debug.Print " is a line. "
   ElseIf (TypeOf oElement Is AeccParcelSegmentCurve) Then
        Dim oSegmentCurve As AeccParcelSegmentCurve
        Set oSegmentCurve = oElement
        Debug.Print " is a curve with a radius of:" _
```

```
            & oSegmentCurve.Radius
      End If
  Next i
```

Parcel Style and Parcel Segment Style

The collection of all parcel styles is held in the `AeccDocument.ParcelStyles` collection. The parcel style controls how a parcel and the parcel segments are displayed. Among the features is an option to only fill the area close to a parcel's borders. A parcel style can be assigned to a parcel through the `AeccParcel.Style` property.

This sample creates a parcel style, sets the style properties, and assigns the style to the parcel object "oParcel":

```
  Dim oParcelStyles As AeccParcelStyles
  Set oParcelStyles = oDocument.ParcelStyles
  Dim oParcelStyle As AeccParcelStyle
  Set oParcelStyle = oParcelStyles.Add("Sample Style")
  oParcelStyle.ObservePatternFillDistance = True
  oParcelStyle.PatternFillDistance = 20
  oParcelStyle.SegmentsDisplayStylePlan.color = 20 '
  red-orange
  oParcelStyle.AreaFillDisplayStylePlan.color = 20
  oParcelStyle.AreaFillDisplayStylePlan.Visible = True
  oParcelStyle.AreaFillDisplayStylePlan.Lineweight = 20
  oParcelStyle.AreaFillHatchDisplayStylePlan.UseAngleOfObject
   = True
  oParcelStyle.AreaFillHatchDisplayStylePlan.ScaleFactor =
  3.8
  oParcelStyle.AreaFillHatchDisplayStylePlan.Spacing = 1.5
  oParcelStyle.AreaFillHatchDisplayStylePlan.Pattern =
  "AR-SAND"
  oParcelStyle.AreaFillHatchDisplayStylePlan.HatchType =
  aeccHatchPreDefined

  ' Assign the "Sample Style" style to a single parcel.
  oParcel.Style = oParcelStyle.Name
```

The style of individual parcel segments depends on the style of the parent parcel, but segments may be shared by different parcels. This conflict is decided by the `AeccParcels.Properties.SegmentDisplayOrder` property, which is a collection of all parcel styles currently in use. These styles are arranged according to priority level. When two parcels with different styles share a segment, the segment is displayed with the higher priority style. Among these styles is the global site parcel style, set through the `AeccParcels.Properties.SiteParcelStyle` property. The site parcel style defines the outside boundary style of parcels within the site, given a high enough priority.

This sample displays the current order of parcel styles in the site and then modifies the order:

```
' List all styles used by parcels in this site with their
' priority
Dim oSegmentDisplayOrder As AeccSegmentDisplayOrder
Set oSegmentDisplayOrder = _
  oSite.Parcels.Properties.SegmentDisplayOrder

Debug.Print "Number styles used:";
oSegmentDisplayOrder.Count
Debug.Print "Priority of each style for affecting segments:"
Dim i as Integer
For i = 0 To oSegmentDisplayOrder.Count - 1
    Debug.Print i; " & oSegmentDisplayOrder.Item(i).Name
Next i

' Set the style with the highest priority to the lowest
' priority.
Dim lLowestPosition as Long
lLowestPosition = oSegmentDisplayOrder.Count - 1
oSegmentDisplayOrder.SetPriority 0, lLowestPosition
```

Parcel Label Style

The style of text labels or graphical markers displayed with parcels and parcel segments are set by assigning a label style object to the `AeccParcel.AreaLabelStyle` property. All such label styles are held in the `AeccDocument.ParcelLabelStyles.AreaLabelStyles` property, a collection of `AeccLabelStyle` objects.

Parcel label styles can use the following property fields in the contents of any text component:

Valid property fields for AeccLabelStyleTextComponent.Contents
<[Name(CU)]>
<[Description(CP)]>
<[Parcel Area(Usq_ft|P2|RN|AP|Sn|OF)]>
<[Parcel Number(Sn)]>
<[Parcel Perimeter(Uft|P3|RN|AP|Sn|OF)]>
<[Parcel Address(CP)]>
<[Parcel Site Name(CP)]>
<[Parcel Tax ID(Sn)]>

Label styles are described in detail in the chapter 1 section Label Styles (page 211).

Parcel User-Defined Properties

Parcel objects can have user-defined properties associated with them, and the properties can be organized into user-defined classifications, or are put into an “Unclassified” classification. You can create new classifications and user-defined properties via the API, though you can’t access the values of existing user-defined properties attached to parcels. For more information about user-defined properties and classifications, see User-Defined Property Classifications in the .

This sample creates a new user-defined property classification for parcels called “Example”, and then adds a new user-defined property with upper and lower bounds and a default value:

```
Dim oApp As AcadApplication
```

```
Set oApp = ThisDrawing.Application
' NOTE - Always specify the version number.
Const sAppName = "AeccXUiLand.AeccApplication.6.0"
Set g_vCivilApp = oApp.GetInterfaceObject(sAppName)
Set g_oDocument = g_vCivilApp.ActiveDocument
Set g_oAeccDb = g_oDocument.Database
Dim oUDPClass As AeccUserDefinedPropertyClassification
Dim oUDPProp As AeccUserDefinedProperty
'Create a user-defined parcel property classification
Set oUDPClass =
g_oAeccDb.ParcelUserDefinedPropertyClassifications.Add("Example")
' Add a Property to our new classification An integer using
 upper
' and lower bound limits of 10 and 20 with a default value
 of 15
Set oUDPProp = oUDPClass.UserDefinedProperties.Add("Extra
 Data", _
   "Some Extra Data", aeccUDPPropertyFieldTypeInteger,
True, False, 10, True, _
   False, 20, True, 15, Null)
```

Accessing Daylight Feature Lines

The `AeccSite:FeatureLines` property is a collection of grading feature lines in the drawing. This collection only contains the types of feature lines available through the Prospector on the AutoCAD Civil 3D user interface. The collection doesn't contain daylight or projection feature lines. However, you can still get information about these types of feature lines programmatically by prompting the user to select feature line object in a drawing. This code sample prompts the user to select a feature line, and then prints out the number of points it contains:

```
Dim objPart As AeccLandFeatureLine
Dim objEnt As AcadObject
Dim objAcadEnt As AcadEntity
Dim varPick As Variant
ThisDrawing.Utility.GetEntity objEnt, varPick, "Select the
 polyline/feature line"
If TypeOf objEnt Is AeccLandFeatureLine Then
   Set objPart = objEnt
   Debug.Print TypeName(objPart)
```

```
    Dim varArray As Variant
    varArray = objPart.GetPoints()
    Debug.Print "Number of points = " & UBound(varArray)
ElseIf TypeOf objEnt Is AcadEntity Then
    Set objAcadEnt = objEnt
    Debug.Print TypeName(objAcadEnt)
    If (g_oAeccDoc.Sites.Count = 0) Then
        g_oAeccDoc.Sites.Add "TestSite"
    End If
    Set objPart =
g_oAeccDoc.Sites(0).FeatureLines.AddFromPolyline(objAcadEnt.ObjectID,
 "Standard")
End If
```

Sample Program

ParcelSample.dvb

<installation-directory>\Sample\Civil 3D API\COM\Vba\Parcel\ParcelSample.dvb

This sample program accesses the active document, creates a site, and then creates three parcels by adding line segments, curve segments, and entity segments. A new parcel style is composed and applied to one of the parcels.

Alignments in COM

This chapter covers creating and using Alignments, Stations, and Alignment styles using the COM API. For information about performing these tasks using the .NET API, see the chapter Using Alignments in .NET (page 65).

Object Hierarchy

Basic Alignment Operations

Creating an Alignment

Alignments are usually created in existing sites. Each `AeccSite` object has its own collection of alignments held in an `AeccAlignments` object in the `AeccSite.Alignments` property. There is also a collection of alignments that are not associated with a site in the `AeccDocument.AlignmentsSiteless` property.

Creating a New Alignment

The `AeccAlignments` object provides two ways of creating new alignments. The `AeccAlignments.Add` method takes an alignment name, a layer to draw to, an alignment style object, and an alignment label style object as parameters and returns a new empty alignment. The `AeccAlignments.AddFromPolyline` method takes the same parameters as well as an AutoCAD polyline entity and flags indicating whether curves should be added between the separate line segments and whether the polyline entity should be erased after the alignment is created.

This code creates an alignment from a 2D polyline, using existing styles:

```
' Create an alignment style with default settings.
Dim oAlignmentStyle as AeccAlignmentStyle
Set oAlignmentStyle = oDocument.AlignmentStyles _
  .Add("Sample style")

' Create an alignment label style with default settings.
Dim oAlignmentLabelStyleSet As AeccAlignmentLabelStyleSet
Set oAlignmentLabelStyleSet =
oAeccDocument.AlignmentLabelStyleSets _
  .Add("Sample label style")

' Get the collection of all siteless alignments.
Dim oAlignments as AeccAlignments
Set oAlignments = oDocument.AlignmentsSiteless
```

```
' Create an empty alignment that draws to layer 0.
Dim oAlignment as AeccAlignment
Set oAlignment = oAlignments.Add("Sample Alignment", "0",
 _
  oAlignmentStyle.Name, oAlignmentLabelStyleSet.Name)


' Create a simple 2D polyline.
Dim oPoly As AcadLWPolyline
Dim dPoints(0 To 5) As Double
dPoints(0) = 0: dPoints(1) = 600
dPoints(2) = 600: dPoints(3) = 0
dPoints(4) = 1200: dPoints(5) = 600
Set oPoly = oDocument.Database.ModelSpace _
  .AddLightWeightPolyline(dPoints)

' Create an alignment from the polyline object. Draw to
' layer 0, erase the polyline when we are done, and
' insert curves between line segments.
Set oAlignment = oAlignments.AddFromPolyline( _
  "Sample Alignment from Polyline", _
  "0", _
  oPoly.ObjectID, _
  oAlignmentStyle.Name, _
  oAlignmentLabelStyleSet.Name, _
  True, _
  True)
```

Creating an Alignment Offset From Another Alignment

Alignments can also be created based on the layout of existing alignments. The `AeccAlignment.Offset` method creates a new alignment with a constant offset and adds it to the same parent site as the original alignment. The new alignment has the same name (followed by a number in parenthesis) and the same style as the original, but it does not inherit any station labels, station equations, or design speeds from the original alignment.

```
' Add an offset alignment 10.5 units to the left of the
' original.
oAlignment.Offset -10.5
```

Defining an Alignment Path Using Entities

An alignment is made up of a series of *entities*, which are individual lines, curves, and spirals that make up the path of an alignment. A collection of entities is held in the `AeccAlignment.Entities` collection. This collection has a wide array of methods for creating new entities.

The following code sample demonstrates some of the entity creation methods:

```
' Define the points used to create the entities.
Dim point1(0 To 2) As Double
Dim point2(0 To 2) As Double
Dim point3(0 To 2) As Double
point1(0) = 200: point1(1) = 800: point1(2) = 0
point2(0) = 600: point1(1) = 400: point1(2) = 0
point3(0) = 1000: point1(1) = 800: point1(2) = 0

' Create a line segment entity that connects two points.
Dim oAlignmentTangent As AeccAlignmentTangent
Set oAlignmentTangent = oAlignment.Entities _
  .AddFixedLine1(point1, point2)

' Print the length of the line segment.
Debug.Print oAlignmentTangent.Length

' Create a curve entity that connects the second endpoint

' of the fixed line to another point. The radius of the
' curve depends on the direction of the fixed line and the
' location of the second endpoint.
Dim oAlignmentArc As AeccAlignmentArc
Set oAlignmentArc = oAlignment.Entities _
  .AddFloatingCurve6(oAlignmentTangent.id, point3)

' Print the angle of direction at the second endpoint.
Debug.Print oAlignmentArc.EndDirection
```

Determining Entities Within an Alignment

Each of the entities in the `AeccAlignment.Entities` collection is a type derived from the `AeccAlignmentEntity`. By checking the `AeccAlignmentEntity.Type` property, you can determine the specific type of each entity and cast the reference to the correct type.

The following sample loops through all entities in an alignment, determines the type of entity, and prints one of its properties.

```
Debug.Print "Number of Entities: ";
oAlignment.Entities.Count

Dim i as Integer
For i = 0 To oAlignment.Entities.Count - 1
    Select Case (oAlignment.Entities.Item(i).Type)
    Case aeccTangent
        Dim oTangent As AeccAlignmentTangent
        Set oTangent = oAlignment.Entities.Item(i)
        Debug.Print "Tangent length:" & oTangent.Length
    Case aeccArc
        Dim oArc As AeccAlignmentArc
        Set oArc = oAlignment.Entities.Item(i)
        Debug.Print "Arc radius:" & oArc.Radius
    Case aeccSpiral
        Dim oSpiral As AeccAlignmentSpiral
        Set oSpiral = oAlignment.Entities.Item(i)
        Debug.Print "Spiral A value:" & oSpiral.A
    Case aeccSpiralCurveSpiralGroup
        Dim oSCSGroup As AeccAlignmentSCSGroup
        Set oSCSGroup = oAlignment.Entities.Item(i)
        Debug.Print "Radius of curve in SCS group:" _
          & oSCSGroup.Arc.Radius

    ' And so on for AeccAlignmentSTSGroup,
    ' AeccAlignmentSTGroup, AeccAlignmentTSGroup
   ' AeccAlignmentSCGroup, and AeccAlignmentCSGroup types.
    End Select
Next i
```

Each entity has an identification number contained in its `AeccAlignmentEntity.Id` property. Each entity knows the numbers of the entities before and after it in the alignment, and you can access specific entities by identification number through the `AeccAlignmentEntities.EntityAtId` method.

Stations

Modifying Stations with Station Equations

A station is a point along an alignment a certain distance from the start of the alignment. By default the station at the start point of an alignment is 0 and increases contiguously through its length. This can be changed by using *station equations*, which can renumber stations along an alignment. A station equation is an object of type `AeccStationEquation` which contains a location along the alignment, a new station number basis, and a flag describing whether station values should increase or decrease from that location on. A collection of these station equations is contained in the `AeccAlignment.StationEquations` property.

The following code changes an alignment so that at a point 80 units from the beginning, stations will start being numbered from the value 720:

```
Dim oStationEquation As AeccStationEquation
Set oStationEquation = oAlignment.StationEquations _
  .Add(80, 0, 720, aeccIncreasing)
```

NOTE

Some functions, such as `AeccAlignment.InstantaneousRadius`, require a “raw” station value without regard to modifications made by station equations.

Creating Station Sets

Alignment stations are usually labeled at regular intervals. You can compute the number, location, and geometry of stations taken at regular spacings by using the `AeccAlignment.GetStations` method. This function returns a collection of stations based on the desired interval of major and minor stations and the type of station requested. Unless the type of station requested is `aeccEquation`, station values ignore any station equations.

```
Dim oStations As AeccAlignmentStations

' If we were to label major stations placed every 100 units
' and minor stations placed every 20, how many labels
' would this alignment have?
Set oStations = oAlignment.GetStations(aeccAll, 100#, 20#)
Debug.Print "Number of labels: " & oStations.Count

' Print the location of each major station in the set.
Dim i as Integer
For i = 0 To oStations.Count - 1
    If (oStations.Item(i).Type = aeccMajor) Then
        Dim j As Integer
        Dim x As Integer
        Dim y As Integer
        j = oStations.Item(i).Station
        x = oStations.Item(i).Easting
        y = oStations.Item(i).Northing
        Debug.Print "Station " & j & " is at:" & x & ", "
 & y
    End If
Next i
```

Specifying Design Speeds

You can assign design speeds along the length of an alignment to aid in the future design of a roadway based on the alignment. The collection of speeds along an alignment are contained in the `AeccAlignment.DesignSpeeds` property. Each item in the collection is an object of type `AeccDesignSpeed`, which contains a raw station value, a speed to be used from that station on until the next design speed specified or the end of the alignment, and an optional string comment.

```
Dim oDesignSpeed As AeccDesignSpeed
' Starting at station 0 + 00.00
Set oDesignSpeed = oAlignment.DesignSpeeds.Add(0#)
oDesignSpeed.Value = 45
oDesignSpeed.Comment = "Straightaway"

' Starting at station 4 + 30.00
Set oDesignSpeed = oAlignment.DesignSpeeds.Add(430#)
oDesignSpeed.Value = 30
oDesignSpeed.Comment = "Start of curve"

' Starting at station 14 + 27.131 to the end.
```

```
Set oDesignSpeed = oAlignment.DesignSpeeds.Add(1427.131)
oDesignSpeed.Value = 35
oDesignSpeed.Comment = "End of curve"
' Make Alignment speed-based
oAlignment.DesignSpeedBased = True
```

Superelevation

Another setting that can be applied to certain stations of an alignment is the superelevation, used to adjust the angle of roadway section components for corridors based on the alignment. The inside and outside shoulders and road surfaces can be adjusted for both the left and right sides of the road. The collection of all superelevation information for an alignment is stored in the `AeccAlignment.SuperelevationData` property. Note that, unlike most AutoCAD Civil 3D API collections, the `Add` method does not return a new default entity but instead passes a reference to the new object through the second parameter. An individual superelevation data element (type `AeccSuperelevationDataElement`) can be accessed through the `AeccAlignment.SuperelevationAtStation` method.

This code creates a new superelevation data element at station 11+00.00 and sets the properties of that element:

```
Dim oSuperElevationData As AeccSuperElevationData
Dim oSuperElevationElem As AeccSuperElevationDataElem

' Create an element at station 11+00.0.  A new default
' superelevation data element is assigned to our
' oSuperElevationElem variable.
Set oSuperElevationData = oAlignment.SuperelevationData
oSuperElevationData.Add 1100, oSuperElevationElem

oSuperElevationElem.SegmentCrossSlope _
   (aeccSuperLeftOutShoulderCrossSlope) = 0.05
oSuperElevationElem.SegmentCrossSlope _
   (aeccSuperLeftOutLaneCrossSlope) = 0.02
oSuperElevationElem.SegmentCrossSlope _
   (aeccSuperLeftInLaneCrossSlope) = 0.01
oSuperElevationElem.SegmentCrossSlope _
   (aeccSuperLeftInShoulderCrossSlope) = 0.03
oSuperElevationElem.SegmentCrossSlope _
   (aeccSuperRightInShoulderCrossSlope) = 0.03
```

```
oSuperElevationElem.SegmentCrossSlope _
   (aeccSuperRightInLaneCrossSlope) = 0.01
oSuperElevationElem.SegmentCrossSlope _
   (aeccSuperRightOutLaneCrossSlope) = 0.02
oSuperElevationElem.SegmentCrossSlope _
   (aeccSuperRightOutShoulderCrossSlope) = 0.05
oSuperElevationElem.TransPointType = aeccSuperManual
oSuperElevationElem.TransPointDesc = "Manual adjustment"
oSuperElevationElem.RawStation = 1100
```

Each superelevation data element represents a point in the transition of the roadway cross section. A single transition from normal to full superelevation and back is a *zone*. A collection of data elements representing a single zone can be retrieved by calling the `AeccAlignment.SuperelevationZoneAtStation` method.

This sample retrieves the data elements that are part of the superelevation zone starting at station 0+00.00, and prints all their descriptions:

```
Set oSuperElevationData = _
   oAlignment.SuperelevationZoneAtStation(0)

For Each oSuperElevationElem In oSuperElevationData
    Debug.Print oSuperElevationElem.TransPointDesc
Next
```

Alignment Style

Creating an Alignment Style

A style governs many aspects of how alignments are drawn, including direction arrows and curves, spirals, and lines within an alignment. All alignment styles are contained in the `AeccDocument.AlignmentStyles` collection. Alignment styles must be added to this collection before being used by an alignment object. A style is normally assigned to an alignment when it is first created, but it can also be assigned to an existing alignment through the `AeccAlignment.Style` property.

```
Dim oAlignmentStyle As AeccAlignmentStyle
Set oAlignmentStyle = oAeccDocument.AlignmentStyles _
```

```
    .Add("Sample alignment style")

  ' Do not show direction arrows.
  oAlignmentStyle.ArrowDisplayStylePlan.Visible = False
  oAlignmentStyle.ArrowDisplayStyleModel.Visible = False
  ' Show curves in violet.
  oAlignmentStyle.CurveDisplayStylePlan.color = 200 ' violet
  oAlignmentStyle.CurveDisplayStyleModel.color = 200 ' violet
  ' Show straight sections in blue.
  oAlignmentStyle.LineDisplayStylePlan.color = 160 ' blue
  oAlignmentStyle.LineDisplayStyleModel.color = 160 ' blue

  ' Assign the style to an existing alignment.
  oAlignment.Style = oAlignmentStyle.Name
```

Alignment Label Styles

The style of text labels and graphical markers displayed along an alignment are set by passing an `AeccAlignmentLabelSet` object when the alignment is first created with the `AeccAlignments.Add` and `AeccAlignments.AddFromPolyline` methods or by assigning the label set object to the `AeccAlignment.LabelStyle` property. The `AeccAlignmentLabelSet` object consists of separate sets of styles to be placed at major stations, minor stations, and where the alignment geometry, design speed, or station equations change.

Labels at major stations are described in the `AeccAlignmentLabelSet.MajorStationLabelSet` property, which is a collection of `AeccLabelMajorStationSetItem` objects. Each `AeccLabelMajorStationSetItem` object consists of a single `AeccLabelStyle` object and a number of properties describing the limits of the labels and the interval between labels along the alignment.

Labels at minor stations are described in the `AeccAlignmentLabelSet.MinorStationLabelSet` property, which is a collection of `AeccLabelMinorStationSetItem` objects. Each `AeccLabelMinorStationSetItem` object consists of a single `AeccLabelStyle` object and a number of properties describing the limits of the labels and the interval between labels along the alignment. When a new `AeccLabelMinorStationSetItem` is created it must reference a parent `AeccLabelMajorStationSetItem` object.

Labels may be placed at the endpoints of each alignment entity. Such labels are controlled through the `AeccAlignmentLabelSet.GeometryPointLabelSet` property, an `AeccLabelSet`. The label set is a collection of `AeccLabelStyle` objects. Labels at each change in alignment design speeds and station equations (the `AeccAlignmentLabelSet.GeometryPointLabelSet` and `AeccAlignmentLabelSet.GeometryPointLabelSet` properties respectively) are also `AeccLabelSet` objects.

All label styles at alignment stations can draw from the following list of property fields:

Valid property fields for AeccLabelStyleTextComponent.Contents
<[Station Value(Uft|FS|P0|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Raw Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Northing(Uft|P4|RN|AP|Sn|OF)]>
<[Easting(Uft|P4|RN|AP|Sn|OF)]>
<[Design Speed(P3|RN|AP|Sn|OF)]>
<[Instantaneous Direction(Udeg|FDMSdSp|MB|P4|RN|DSn|CU|AP|OF)]>
<[Perpendicular Direction(Udeg|FDMSdSp|MB|P4|RN|DSn|CU|AP|OF)]>
<[Alignment Name(CP)]>
<[Alignment Description(CP)]>
<[Alignment Length(Uft|P3|RN|AP|Sn|OF)]>
<[Alignment Start Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Alignment End Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>

Label styles for minor stations, geometry points, design speeds, and station equations can also use the following property fields:

<[Offset From Major Station(Uft|P3|RN|AP|Sn|OF)]>	Minor stations
<[Geometry Point Text(CP)]>	Geometry points
<[Geometry Point Entity Before Data(CP)]>	Geometry points
<[Geometry Point Entity After Data(CP)]>	Geometry points
<[Design Speed Before(P3|RN|AP|Sn|OF)]>	Design speeds
<[Station Ahead(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>	Station equations
<[Station Back(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>	Station equations
<[Increase/Decrease(CP)]>	Station equations

Label styles are described in detail in the chapter 1 section Label Styles (page 211).

Sample Program

AlignmentSample.dvb

<installation-directory>\Sample\Civil 3D API\Vba\Alignment\AlignmentSample.dvb

The sample code from this chapter can be found in context in the AlignmentSample.dbp project. This project can create alignments based on AutoCAD polyline entities or through layout commands. It demonstrates using styles, label styles, station equations, design speeds, and superelevation.

Profiles in COM

This chapter covers creating and using Profiles using the COM API. For information about performing these tasks using the .NET API, see the chapter Profiles in .NET (page 78).

Object Hierarchy

Profiles

This section covers the creation and style of profiles. Profiles are the vertical analogue to alignments. Together, an alignment and a profile represent a 3D path.

Creating a Profile From a Surface

A profile is an object consisting of elevations along an alignment. Each alignment contains a collection of profiles in its `AeccAlignment.Profiles` property. The `AeccProfiles.AddFromSurface` creates a new profile and derives its elevation information from the specified surface along the path of the alignment.

```
Dim oProfiles As AeccProfiles
Set oProfiles = oAlignment.Profiles
Dim oProfile As AeccProfile

' Add a new profile for the alignment "oAlignment" based
' on the elevations of the surface "oSurface".
Set oProfile = oProfiles.AddFromSurface( _
  "Profile01", _
  aeccExistingGround, _
  oProfileStyle.Name, _
  oSurface.Name, _
  oAlignment.StartingStation, _
  oAlignment.EndingStation, _
  "0")
```

Creating a Profile Using Entities

The `AeccProfiles.Add` method creates a new profile with no elevation information. The vertical shape of a profile can then be specified using entities. Entities are geometric elements - tangents or symmetric parabolas. The collection of all entities that make up a profile are contained in the

AeccProfile.Entities collection. AeccProfile.Entities also contains all the methods for creating new entities.

This sample creates a new profile along the alignment "oAlignment" and then adds three entities to define the profile shape. Two straight entities are added at each end and a symmetric parabola is added in the center to join them and represent the sag of a valley.

```
Dim oProfiles As AeccProfiles
Set oProfiles = oAlignment.Profiles
Dim oProfile As AeccProfile

' NOTE: The second parameter (aeccFinishedGround) indicates
' that the shape of the profile is not drawn from the
existing
' surface. We will define the profile ourselves.
Set oProfile = oProfiles.Add _
  ("Profile03", _
  aeccFinishedGround, _
  oProfileStyle.Name)

' Now add the entities that define the profile.
' NOTE: Profile entity points are not x,y,z point, but
' station-elevation locations.
Dim dLoc1(0 To 1) As Double
Dim dLoc2(0 To 1) As Double
Dim oProfileTangent1 As aeccProfileTangent
dLoc1(0) = oAlignment.StartingStation: dLoc1(1) = -40#
dLoc2(0) = 758.2: dLoc2(1) = -70#
Set oProfileTangent1 = oProfile.Entities.AddFixedTangent
_
  (dLoc1, dLoc2)

Dim oProfileTangent2 As aeccProfileTangent
dLoc1(0) = 1508.2: dLoc1(1) = -60#
dLoc2(0) = oAlignment.EndingStation: dLoc2(1) = -4#
Set oProfileTangent2 = oProfile.Entities.AddFixedTangent
_
  (dLoc1, dLoc2)

Dim dCrestLen As Double
dCrestLen = 900.1
Call oProfile.Entities.AddFreeSymmetricParabolaByLength _
```

```
    (oProfileTangent1.Id, _
    oProfileTangent2.Id, _
    aeccSag, _
    dCrestLen, _
    True)
```

Editing Points of Vertical Intersection

The point where two adjacent tangents would cross (whether they actually cross or not) is called the “point of vertical intersection”, or “PVI.” This location can be useful for editing the geometry of a profile because this one point controls the slopes of both tangents and any curve connecting them. The collection of all PVIs in a profile are contained in the `AeccProfile.PVIs` property. This object lets you access, add, and remove PVIs from a profile, which can change the position and number of entities that make up the profile. Individual PVIs (type AeccProfilePVI) do not have a name or id, but are instead identified by a particular station and elevation. The collection methods `AeccProfilePVIs.ItemAt` and `AeccProfilePVIs.RemoveAt` access or delete the PVI closest to the station and elevation parameters so you do not need the exact location of the PVI you want to modify.

This sample identifies the PVI closest to a specified point. It then adds a new PVI to profile created in the Creating a Profile Using Entities (page 299) topic and adjusts its elevation.

```
Dim oPVI As AeccProfilePVI

' Find the PVI close to station 1000 elevation -70.
Set oPVI = Nothing
Set oPVI = oProfile.PVIs.ItemAt(1000, -70)
Debug.Print "PVI closest to station 1000 is at station: ";

Debug.Print oPVI.Station

' Add another PVI and slightly adjust its elevation.
Set oPVI = oProfile.PVIs.Add(607.4, -64.3,
aeccProfileTangent)
oPVI.Elevation = oPVI.Elevation - 2#
```

Creating a Profile Style

The profile style, an object of type `AeccProfileStyle`, defines the visual display of profiles. The collection of all such styles in a document are stored in the `AeccDocument.LandProfileStyles` property. The style contains objects of type `AeccDisplayStyle` which govern the display of arrows showing alignment direction and of the lines, line extensions, curves, parabolic curve extensions, symmetrical parabolas and asymmetrical parabolas that make up a profile. The properties of a new profile style are defined by the document's ambient settings.

```
Dim oProfileStyle As AeccProfileStyle
Set oProfileStyle = oDocument.LandProfileStyles
  .Add("Profile Style 01")

' For all profiles that use this style, line elements
' will be yellow, curve elements will be shades of green,
' and extensions will be dark grey. No arrows will be shown.
With oProfileStyle
    .ArrowDisplayStyleProfile.Visible = False
    .LineDisplayStyleProfile.Color = 50 ' yellow
    .LineDisplayStyleProfile.Visible = True
    .LineExtensionDisplayStyleProfile.Color = 251 ' grey
    .LineExtensionDisplayStyleProfile.Visible = True
    .CurveDisplayStyleProfile.Color = 80 ' green
    .CurveDisplayStyleProfile.Visible = True
    .ParabolicCurveExtensionDisplayStyleProfile.Color = 251
 ' grey
    .ParabolicCurveExtensionDisplayStyleProfile.Visible =
True
    .SymmetricalParabolaDisplayStyleProfile.Color = 81 '
green
    .SymmetricalParabolaDisplayStyleProfile.Visible = True
    .AsymmetricalParabolaDisplayStyleProfile.Color = 83 '
green
    .AsymmetricalParabolaDisplayStyleProfile.Visible = True
End With
' Properties for 3d display should also be set.
```

Profile Views

This section describes the creation and display of profile views. A profile view is a graph displaying the elevation of a profile along the length of the related alignment.

Creating a Profile View

A profile view, an object of type `AeccProfileView`, is a graphical display of profiles within a graph. A collection of profile views is contained in each alignment's `AeccAlignment.ProfileViews` property.

```
Dim dOriginPt(0 To 2) As Double
dOriginPt(0) = 6000 ' X location of profile view
dOriginPt(1) = 3500 ' Y location
dOriginPt(2) = 0 ' Z location

' Use the first profile view style in the document.
Dim oProfileViewStyle As AeccProfileViewStyle
Set oProfileViewStyle = oDocument.ProfileViewStyles.Item(0)

Dim oProfileView as AeccProfileView
Set oProfileView = oAlignment.ProfileViews.Add( _
  "Profile Style 01", _
  "0", _
  dOriginPt, _
  oProfileViewStyle, _
  Nothing)  ' "Nothing" means do not include data bands.
```

Creating Profile View Styles

The profile view style, an object of type `AeccProfileViewStyle`, governs all aspects of how the graph axes, text, and titles are drawn. Within `AeccProfileViewStyle` are objects dealing with the top, bottom, left, and right axes; lines at geometric locations within profiles; and with the graph as a whole. All profile view styles in the document are stored in the `AeccDocument.ProfileViewStyles` collection. New styles are created using the collection's `Add` method with the name of the new style.

```
Dim oProfileViewStyle As AeccProfileViewStyle
Set oProfileViewStyle = oDocument.ProfileViewStyles _
  .Add("Profile View style 01")
```

Setting Profile View Styles

The profile view style object consists of separate objects for each of the four axes, one object for the graph overall, and an `AeccDisplayStyle` object for grid lines displayed at horizontal geometry points. The axis styles and graph style also contain subobjects for specifying the style of tick marks and titles.

Setting the Axis Style

All axis styles are based on the `AeccAxisStyle` class. The axis style object controls the display style of the axis itself, tick marks and text placed along the axis, and a text annotation describing the axis's purpose. The annotation text, location, and size is set through the `AeccAxisStyle.TitleStyle` property, an object of type `AeccAxisTitleStyle`. The annotation text can use any of the following property fields:

Valid property fields for AeccAxisTitleStyle.Text
<[Profile View Minimum Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[Profile View Maximum Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[Profile View Start Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Profile View End Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>

Axis Tick Marks

Within each axis style are properties for specifying the tick marks placed along the axis. Both major tick marks and minor tick marks are represented by objects of type `AeccTickStyle`. `AeccTickStyle` manages the location, size, and visual style of tick marks through its `Interval`, `Size` and `DisplayStylePlan` properties. Note that while most style properties use drawing units, the `Interval` property uses the actual ground units of the surface. The `AeccTickStyle` object also sets what text is displayed at each tick, including the following property fields:

Valid property fields for AeccTickStyle.Text	Axis
<[Station Value(Uft|FS|P0|RN|AP|Sn|TP|B2|EN|W0|OF)]>	horizontal
<[Raw Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>	horizontal
<[Graph View Abscissa Value(Uft|P4|RN|AP|Sn|OF)]>	horizontal
<[Profile View Point Elevation(Uft|P1|RN|AP|Sn|OF)]>	vertical
<[Graph View Ordinate Value(Uft|P3|RN|AP|Sn|OF)]>	vertical

Setting the Graph Style

The graph is managed by an object of type `AeccGraphStyle`. This object can be used to change the grid and the title of the graph. The grid is controlled by the `AeccGraphStyle.GridStyle` property, an object of type `AeccGridStyle`. The grid style sets the amount of empty space above and below the extents of the section through the `AeccGridStyle.GridsAboveMaxElevation` and `AeccGridStyle.GridsBelowDatum` properties. The grid style also manages the line styles of major and minor vertical and horizontal gridlines with the `AeccGridStyle` properties `MajorVerticalDisplayStylePlan`, `MajorHorizontalDisplayStylePlan`, `MinorVerticalDisplayStylePlan`, and `MinorHorizontalDisplayStylePlan`. The `AeccGridLines.VerticalPosition` and `AeccGridLines.HorizontalPosition` properties tell which axis to use to position the grid lines.

Graph Title

The title of the graph is controlled by the `AeccGraphStyle.TitleStyle` property, an object of type `AeccGraphTitleStyle`. The title style object can adjust the position, style, and border of the title. The text of the title can include any of the following property fields:

Valid property fields for AeccGraphTitleStyle.Text
<[Graph View Name(CP)]>
<[Parent Alignment(CP)]>
<[Drawing Scale(P4|RN|AP|OF)]>
<[Graph View Vertical Scale(P4|RN|AP|OF)]>
<[Graph View Vertical Exageration(P4|RN|AP|OF)]>
<[Profile View Start Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Profile View End Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Profile View Minimum Elevation(Uft|P2|RN|AP|Sn|OF)]>
<[Profile View Maximum Elevation(Uft|P3|RN|AP|Sn|OF)]>

Profile View Style Example

This example takes an existing profile view style and modifies its top axis and title:

```
' Get the first style in the document's collection of
styles.
Dim oProfileViewStyle as AeccProfileViewStyle
Set oProfileViewStyle = oDocument.ProfileViewStyles.Item(0)

' Adjust the top axis. Put station information here, with
' the title at the far left.

With oProfileViewStyle.TopAxis
    .DisplaySyle2d.Visible = True
    ' Modify the text to display meters in decimal
    ' format.
    .MajorTickStyle.Text = "<[Station Value(Um|FD|P1)]> m"
    .MajorTickStyle.Interval = 164.041995
    Dim dPoint(0 To 2) As Double
    dPoint(0) = 0.13
    dPoint(1) = 0#
    .TitleStyle.Offset = dPoint
    .TitleStyle.Text = "Meters"
    .TitleStyle.Position = aeccStart
End With

' Adjust the title to show the alignment name.
oProfileViewStyle.GraphStyle.TitleStyle.Text = _
  "Profile View of:<[Parent Alignment(CP)]>"
```

Sample Programs

ProfileSample.dvb

<installation-directory>\Sample\Civil 3D API\Vba\Profile\ProfileSample.dvb

This sample creates a surface from a file and an alignment using entities. It creates a profile based on the existing surface along the alignment. It then creates a second profile using entities and points of vertical intersection. A profile view is made displaying both profiles. The style and label style of the profile view are set with custom styles. Data bands are added to the profile view.

CorridorSample.dvb

<installation-directory>\Sample\Civil 3D API\Vba\Corridor\CorridorSample.dvb

The `CreateCorridor` method of the Corridor module shows the creation of a profile using entities.

Sections in COM

Object Hierarchy

Sample Lines

This section covers the creation and use of sample lines. Sample lines are line segments placed at regular intervals across an alignment. They can be used to define the location and orientation of surface cross sections that can be studied through section views.

Creating a Sample Line

Sample lines are line segments placed along an alignment, usually perpendicular to the alignment path and at regular intervals. Sample lines represent the location and orientation of surface cross sections that can be studied through section views.

Sample lines are created in groups. All sample line groups for an alignment are held in the `AeccAlignment.SampleLineGroups` collection. The `Add` method for this collection creates a new empty group, and takes as parameters the name of the group, the layer the group will be drawn to, a style object for how groups of graphs are organized, a style object for the sample lines, and a label style object.

```
' Get all the styles we will need for our sample line
object.
' We will use whatever default styles the document contains.
Dim oGroupPlotStyle As AeccGroupPlotStyle
Set oGroupPlotStyle = oDocument.GroupPlotStyles.Item(0)

Dim oSampleLineStyle As AeccSampleLineStyle
Set oSampleLineStyle = oDocument.SampleLineStyles.Item(0)

Dim oSampleLineLabelStyle As AeccLabelStyle
Set oSampleLineLabelStyle = oDocument _
  .SampleLineLabelStyles.Item(0)

Dim oSampleLineGroups As AeccSampleLineGroups
Set oSampleLineGroups = oAlignment.SampleLineGroups

' Create a sample line group using the above styles and
drawn
' to layer "0".
Dim sLayerName as String
```

```
sLayerName = "0"
Dim oSampleLineGroup As AeccSampleLineGroup
Set oSampleLineGroup = oSampleLineGroups.Add( _
  "Example Sample Line Group", _
  sLayerName, _
  oGroupPlotStyle, _
  oSampleLineStyle, _
  oSampleLineLabelStyle)
```

Defining Sample Lines

Setup

The first step in creating sample lines along an alignment is to specify the surface or surfaces being sampled. This is accomplished by adding surfaces to the `AeccSampleLineGroup.SampledSurfaces` collection. The `AeccSampledSurfaces.AddAllSurfaces` method adds one `AeccSampledSurface` object to the collection for each surface in the document. The `AeccSampledSurfaces.Add` method takes a specific surface object and an `AeccSectionStyle` object and returns a reference to the added `AeccSampledSurface` object. It is important to then set the boolean `AeccSampledSurface.Sample` property to `True`.

NOTE

If no `SampledSurface` object added to the `AeccSampledSurfaces` collection has the `Sample` property set to `True`, then no sections will be generated for that sample line.

This sample demonstrates how to add a single surface to a sample line group and how to add all surfaces in the drawing to a sample line group:

```
' Get the section style we need for our sampled surface
' object. We use the default style of the document.
Dim oSectionStyle As AeccSectionStyle
Set oSectionStyle = oDocument.SectionStyles.Item(0)

' Get the surface object we will be sampling from.
' Assume there is a surface with the name "EG".
Dim oSurface as AeccSurface
Set oSurface = oDocument.Surfaces.Item("EG")
```

```
' This section demonstrates adding a single surface to
' the sample line group. An AeccSampleSurface object is
' returned, which needs some properties set.
Dim oSampledSurface As AeccSampledSurface
Set oSampledSurface = oSampleLineGroup.SampledSurfaces _
  .Add(oSurface, oSectionStyle)

oSampledSurface.UpdateMode = aeccSectionStateDynamic
' We need to set the Sample property of the
' SampledSurface object. Otherwise the sampled surface
' will not be used in creating sections.
oSampledSurface.Sample = True


' This section demonstrates adding all surfaces in the
' document to the sample line group.
oSampleLineGroup.SampledSurfaces.AddAllSurfaces
oSectionStyle
' We need to set the Sample property of each
' SampledSurface object. Otherwise the sampled surfaces
' will not be used in creating sections.
Dim i As Integer
For i = 0 To oSampleLineGroup.SampledSurfaces.Count - 1
    oSampleLineGroup.SampledSurfaces.Item(i).Sample = True
Next i
```

At this point you can define sample lines. Sample lines are held in the `AeccSampleLineGroup.SampleLines` collection. There are three methods to add sample lines to the collection: `AeccSampleLines.AddByPolyline`, `AeccSampleLines.AddByStation`, and `AeccSampleLines.AddByStationRange`.

The `AeccSampledSurfaces` collection can contain objects other than surfaces to be sampled in section views, such as pipe networks or corridors. These objects can be added from the AutoCAD Civil 3D user interface. (Currently you can only add surfaces to the `AeccSampledSurfaces` collection via the API.)

Accessing these non-surface objects with the `AeccSampledSurface.Surface` property throws an exception, so you need to use the `AeccSampledSurface.AcadEntity` property, as in this example, where the second item in the collection is a pipe network:

```
Dim SampledSurfaces As AeccSampledSurfaces
SampledSurfaces =
aeccdb.Sites.Item(0).Alignments.Item(1).SampleLineGroups.Item(0).SampledSurfaces
```

```
Dim s1 As AeccSurface
Dim e1 As AcadEntity
Dim s2 As AeccPipeNetwork
s1 = SampledSurfaces.Item(0).Surface
e1 = SampledSurfaces.Item(1).AcadEntity ' Surface would
throw exception
MessageBox.Show(e1.ObjectName)
s2 = e1
MessageBox.Show(s2.Name)
```

Adding a Sample Line from a Polyline

`AddByPolyline` lets you specify the location of a sample line based on an AutoCAD lightweight polyline entity. This gives you great flexibility in designing the sample line. It can consist of many line segments at any orientation, and it does not have to be perpendicular to the alignment or even cross the alignment at all. The `AddByPolyline` method also lets you delete the polyline entity once the sample line has been created. `AddByPolyline` returns the `AeccSampleLine` object created.

```
Dim oPoly As AcadLWPolyline
Dim dPoints(0 To 3) As Double
' Assume these coordinates are in one of the surfaces
' in the oSampleLineGroup.SampledSurfaces collection.
dPoints(0) = 4750: dPoints(1) = 4050
dPoints(2) = 4770: dPoints(3) = 3950
Set oPoly = ThisDrawing.ModelSpace _
  .AddLightWeightPolyline(dPoints)

' Now that we have a polyline, we can create a sample line
' with those coordinates. Delete the polyline when done.
Call oSampleLineGroup.SampleLines.AddByPolyline _
  ("Sample Line 01", oPoly, True)
```

Add a Sample Line at a Station

`AddByStation` creates a single sample line perpendicular to a particular alignment station. The `AddByStation` method takes as parameters the name of the sample line, the station the line crosses, and the length of the line to the left and right sides of the alignment. `AddByStation` returns the `AeccSampleLine` object created.

```
Dim dSwathWidthLeft As Double
Dim dSwathWidthRight As Double
Dim dStation As Double
dSwathWidthRight = 45.5
dSwathWidthLeft = 35.5
dStation = 1100.5

Call oSampleLineGroup.SampleLines.AddByStation( _
  "Sample Line 02", _
  dStation, _
  dSwathWidthLeft, _
  dSwathWidthRight)
```

Add a Range of Sample Lines

`AddByStationRange` creates a series of sample lines along the alignment. The characteristics of the sample lines are defined by an object of type `AeccStationRange`. When first created, the `AeccStationRange` object properties are set to default values, so be sure to set every property to the values you require.

NOTE

The `AeccStationRange.SampleLineStyle` property must be set to a valid object or the `AddByStationRange` method will fail.

The `AeccStationRange` object is then passed to the `AddByStationRange` method along with a flag describing how to deal with duplicates. The sample lines are then generated and added to the `AeccSampleLineGroup.SampleLines` collection. This method has no return value.

This sample adds a series of sample lines along a section of an alignment:

```
' Specify where the sample lines will be drawn.
Dim oStationRange As New AeccStationRange
oStationRange.UseSampleIncrements = True
oStationRange.SampleAtHighLowPoints = False
oStationRange.SampleAtHorizontalGeometryPoints = False
oStationRange.SampleAtSuperelevationCriticalStations =
False
oStationRange.SampleAtRangeEnd = True
oStationRange.SampleAtRangeStart = True
oStationRange.StartRangeAtAlignmentStart = False
oStationRange.EndRangeAtAlignmentEnd = False
' Only sample for 1000 units along part of the
' alignment.
oStationRange.StartRange = 10#
oStationRange.EndRange = 1010#
' sample every 200 units along straight lines
oStationRange.IncrementTangent = 200#
' sample every 50 units along curved lines
oStationRange.IncrementCurve = 50#
' sample every 50 units along spiral lines
oStationRange.IncrementSpiral = 50#
' 50 units to either side of the station
oStationRange.SwathWidthLeft = 50#
oStationRange.SwathWidthRight = 50#
oStationRange.SampleLineDefaultDirection = _
  aeccDirectionFromBaseAlignment
Set oStationRange.SampleLineStyle = oSampleLineStyle

oSampleLineGroup.SampleLines.AddByStationRange _
  "Sample Line 03", _
  aeccSampleLineDuplicateActionOverwrite, _
  oStationRange
```

Creating Sample Line Styles

In creating sample lines, you need to work with three different style objects that control how sample lines are displayed.

Group Plot Styles

The `AeccGroupPlotStyle` style controls how groups of section view graphs are drawn. The style changes the row and column orientation and spacing between multiple graphs. The collection of all `AeccGroupPlotStyle` styles is contained in the `AeccDocument.GroupPlotStyles` property.

```
Dim oGroupPlotStyle As AeccGroupPlotStyle
Set oGroupPlotStyle = oDocument.GroupPlotStyles _
  .Add("Example group plot style")
```

Sample Line Styles

The `AeccSampleLineStyle` style controls how sample lines are drawn on a surface. The collection of all `AeccSampleLineStyle` styles is contained in the `AeccDocument.SampleLineStyles` property.

```
Dim oSampleLineStyle As AeccSampleLineStyle
Set oSampleLineStyle = oDocument.SampleLineStyles _
  .Add("Example sample line style")

' This style just changes the display of the sample line.
oSampleLineStyle.LineDisplayStyleSection.color = 140 '
slate
```

Section Styles

The `AeccSectionStyle` style controls how surface cross sections are displayed in the section view graphs. The collection of all `AeccSectionStyle` styles is contained in the `AeccDocument.SectionStyles` property.

```
Dim oSectionStyle As AeccSectionStyle
Set oSectionStyle = oDocument.SectionStyles _
  .Add("Example cross section style")

' This style just changes the display of cross section
' lines.
oSectionStyle.SegmentDisplayStyleSection.color = 110 '
green/blue
```

Creating Sample Line Label Styles

The style of sample line text labels, lines, and marks are controlled by an `AeccLabelStyle` object. The style can be set through the `AeccSampleLine.LabelStyle` property or by passing an `AeccLabelStyle` object when using the `AeccSampleLineGroups.Add` method. All labels styles for sample lines are stored in the `AeccDocument.SampleLineLabelStyles` collection. See the Root Objects chapter for more detailed information about the `AeccLabelStyle` class.

Text labels for sample lines can use any of the following property fields:

Valid property fields for AeccLabelStyleTextComponent.Contents
<[Sample Line Name(CP)]>
<[Sample Line Number(Sn)]>
<[Left Swath Width(Uft|P3|RN|AP|Sn|OF)]>
<[Right Swath Width(Uft|P3|RN|AP|Sn|OF)]>
<[Distance from Previous Sample Line(Uft|P3|RN|AP|Sn|OF)]>
<[Sample Line Parent Alignment Name(CP)]>
<[Sample Line Group(CP)]>
<[Sample Line Station Value(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Sample Line Raw Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>

Label styles are described in detail in the Chapter 1 section Label Styles (page 211).

Sections

This section covers the creation and use of sections. A section is a cross section of one or more surfaces along a sample line. A series of sections can be used to analyze the changes in a surface along a path.

Creating Sections

A section is a cross section of one or more surfaces taken along a sample line. You can either create sections one at a time using the `AeccSampleLineGroup.CreateSectionsAtSampleLine` method, or all at once using the `AeccSampleLineGroup.CreateSectionsAtSampleLines` method. These methods will cause an error if the sample line group does not reference any surfaces, or if the surface is not located under the sample lines specified.

```
' Create a section at the first sample line in the sample
' line group.
' oSampleLineGroup is of type AeccSampleLineGroup

Dim oSampleLine as AeccSampleLine
Set oSampleLine = oSampleLineGroup.SampleLines(0)
oSampleLineGroup.CreateSectionsAtSampleLine oSampleLine

' Create a section for each sample line in the sample
' line group.
oSampleLineGroup.CreateSectionsAtSampleLines
```

Using Sections

Each sample line contains a collection of sections that were based on that sample line. Each section is represented by object of type `AeccSection` and contains methods for retrieving statistics of the surface along the section. While sections initially have styles based on the `AeccSectionStyle` style passed to the `AeccSampledSurfaces.Add` method, you can also set the style for each section individually through the `AeccSection.Style` property.

```
Dim oSampleLines as AeccSampleLines
Set oSampleLines = oSampleLineGroup.SampleLines

' For each sample line, go through all the sections that
' were created based on it.
Dim i As Integer
Dim j As Integer
For i = 0 To oSampleLines.Count - 1
   Dim oSections As AeccSections
   Set oSections = oSampleLines.Item(i).Sections

   ' For each section, print its highest elevation and set
   ' some of its properties.
   Dim oSection as AeccSection
   For Each oSection in oSections
      Debug.Print "Max Elevation of "; oSection.Name;
      Debug.Print " is: "; oSection.ElevationMax
```

```
        oSection.DataType = aeccSectionDataTIN
       oSection.StaticDynamic = aeccSectionStateDynamic
    Next
Next i
```

Section Views

This section describes the creation and display of section views. A section view is a graph of a single section. Usually a series of section views are displayed to demonstrate a range of cross sections.

Creating Section Views

A section view is a graph of the sections for a single sample. Each sample line contains a collection of section views in its `AeccSampleLine.SectionViews` property. To create a new section view, use the `AeccSectionViews.Add` method, which takes as parameters the name of the new section view, the layer to draw to, the location, the style of the view, and an optional data band set. Each section view is automatically constructed to display the sections at that sample line in the center of an appropriately sized graph. As each sample line may have different lengths and represent different surface altitudes, each section view may be different in size or in what units are displayed along each graph axis.

This sample creates a row of section views from all sample lines in a given alignment:

```
Dim i As Integer
Dim j As Integer

' Use the first section view style in the document.
Dim oSectionViewStyle As AeccSectionViewStyle
Set oSectionViewStyle = oDocument.SectionViewStyles.Item(0)

' Specify the starting location of the row of section
' views.
Dim dX As Double
Dim dY As Double
dX = 6000
dy = 3500
```

```
' We have an alignment with sample lines.  Loop through
' all the sample line groups in the alignment.
For i = 0 To oAlignment.SampleLineGroups.Count - 1
   Dim oSampleLineGroup As AeccSampleLineGroup
   Set oSampleLineGroup =
oAlignment.SampleLineGroups.Item(i)
   Dim oSampleLines As AeccSampleLines
   Set oSampleLines = oSampleLineGroup.SampleLines

   ' Now loop through all the sample lines in the current
   ' sample line group.  For each sample line, we add a
   ' section view at a unique location with a style and
   ' a data band that we defined earlier.
   Dim dOffsetRight As Double
   dOffsetRight = 0
   For j = 0 To oSampleLines.Count - 1
      Dim oSectionView As AeccSectionView
      dOffsetRight = j * 300
      Dim dOriginPt(0 To 2) As Double
      ' To the right of the surface and the previous
      ' section views.
      dOriginPt(0) = dX + 200 + dOffsetRight
      dOriginPt(1) = dY
      Set
oSectionView=oSampleLines.Item(j).SectionViews.Add( _
        "Section View" & CStr(j), _
        "0", _
        dOriginPt, _
        oSectionViewStyle, _
        Nothing) ' "Nothing" means do not display a data
band
   Next j
Next i
```

Creating Section View Styles

The section view style, an object of type `AeccSectionViewStyle`, governs all aspects of how the graph axes, text, and titles are drawn. Within `AeccSectionViewStyle` are objects dealing with the top, bottom, left, center vertical, and right axes and with the graph as a whole. All section view styles in the document are stored in the `AeccDocument.SectionViewStyles` collection.

New styles are created using the collection's `Add` method with the name of the new style.

```
Dim oSectionViewStyle As AeccSectionViewStyle
Set oSectionViewStyle = oDocument.SectionViewStyles _
  .Add("Section View style")
```

Setting Section View Styles

The section view style object consists of separate objects for each of the four axes, one object for the graph overall, and an `AeccDisplayStyle` object for sample lines within the views. The axis styles and graph style also contain subobjects for specifying the style of tick marks and titles.

Setting the Axis Style

All axis styles are based on the `AeccAxisStyle` class. The axis style object controls the display style of the axis itself, tick marks and text placed along the axis, and a text annotation describing the purpose of the axis. The annotation text, location, and size is set through the `AeccAxisStyle.TitleStyle` property, an object of type `AeccAxisTitleStyle`. The annotation text can use any of the following property fields:

Valid property fields for AeccAxisTitleStyle.Text	**Axes**
<[Section View Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>	top, bottom
<[Section View Width(Uft|P2|RN|AP|Sn|OF)]>	top, bottom
<[Left Width(Uft|P2|RN|AP|Sn|OF)]>	top, bottom
<[Right Width(Uft|P2|RN|AP|Sn|OF)]>	top, bottom
<[Elevation Range(Uft|P2|RN|AP|Sn|OF)]>	left, right, center
<[Minimum Elevation(Uft|P2|RN|AP|Sn|OF)]>	left, right, center

Valid property fields for AeccAxisTitleStyle.Text	Axes
<[Maximum Elevation(Uft|P2|RN|AP|Sn|OF)]>	left, right, center

Axis Tick Marks

Within each axis style are properties for specifying the tick marks placed along the axis. Both major tick marks and minor tick marks are represented by objects of type `AeccTickStyle`. `AeccTickStyle` manages the location, size, and visual style of tick marks through its `Interval`, `Size` and `DisplayStylePlan` properties.

NOTE

While most style properties use drawing units, the `Interval` property uses surface units.

The `AeccTickStyle` object also sets what text is displayed at each tick, including any of the following property fields:

Valid property fields for AeccTickStyle.Text
<[Section View Point Offset Side(CP)]>
<[Section View Point Offset(Uft|P3|RN|Sn|OF|AP)]>
<[Graph View Abscissa Value(Uft|P3|RN|AP|Sn|OF)]>

Setting Graph Styles

The graph is managed by an object of type `AeccGraphStyle`. This object can be used to change the grid and the title of the graph. The grid is controlled by the `AeccGraphStyle.GridStyle` property, an object of type `AeccGridStyle`. The grid style sets the amount of empty space above and below the extents of the section through the `AeccGridStyle.GridsAboveMaxElevation` and `AeccGridStyle.GridsBelowDatum` properties. The grid style also manages the line styles of major and minor vertical and horizontal gridlines with the `AeccGridStyle` properties `MajorVerticalDisplayStylePlan`, `MajorHorizontalDisplayStylePlan`, `MinorVerticalDisplayStylePlan`, and `MinorHorizontalDisplayStylePlan`. The `AeccGridLines.HorizontalPosition` and `AeccGridLines.VerticalPosition` properties tell which tick marks to use to position the grid lines for the axis.

Graph Title

The title of the graph is controlled by the `AeccGraphStyle.TitleStyle` property, an object of type `AeccGraphTitleStyle`. The title style object can adjust the position, style, and border of the title. The text of the title can include any of the following property fields:

Valid property fields for AeccGraphTitleStyle.Text
<[Section View Description(CP)]>
<[Section View Name(CP)]>
<[Parent Alignment(CP)]>
<[Section View Station(Uft|FS|P3|RN|Sn|OF|AP|B2|TP|EN|W0|DZY)]>
<[Section View Datum Value(Uft|P3|RN|AP|Sn|OF)]>
<[Section View Width(Uft|P3|RN|AP|Sn|OF)]>
<[Left Width(Uft|P3|RN|AP|Sn|OF)]>
<[Right Width(Uft|P3|RN|AP|Sn|OF)]>
<[Drawing Scale(P3|RN|AP|OF)]>
<[Graph View Vertical Scale(P3|RN|AP|OF)]>
<[Graph View Vertical Exageration(P3|RN|AP|OF)]>
<[Sample Line Name(CP)]>
<[Sample Line Group(CP)]>

Valid property fields for AeccGraphTitleStyle.Text
<[Sample Line Number(Sn)]>

Section View Style Example

This example takes an existing section view style and modifies its top axis and title.

```
' We assume a section view style with the name
' "Section View style" already exists.
Dim oSectionViewStyle As AeccSectionViewStyle
Set oSectionViewStyle = oDocument.SectionViewStyles _
  .Item("Section View style")

''''''''''''''''''''''''''''''''''''''
' Adjust the top axis.
With oSectionViewStyle.TopAxis.MajorTickStyle
    .AnnotationDisplayStylePlan.color = 23 ' light brown
    .Height = 0.004 ' text height
    .AnnotationDisplayStylePlan.Visible = True ' show text
    .Interval = 15 ' Major ticks 15 ground units apart
    ' Each major tick is marked with distance from the
    ' centerline in meters.
    .Text = "<[Section View Point
Offset(Um|P1|RN|Sn|AP|OF)]>m"
    .DisplayStylePlan.Visible = True ' show ticks
End With
With oSectionViewStyle.TopAxis.TitleStyle
    .DisplayStylePlan.color = 23 ' light brown
    .Height = 0.008 ' text height
    .Text = "Meters"
    ' Position the title slightly higher.
    dOffset(0) = 0#
    dOffset(1) = 0.02
    .Offset = dOffset
    .DisplayStylePlan.Visible = True ' show title
End With

''''''''''''''''''''''''''''''''''''''
```

```
' Adjust the graph and graph title.
With oSectionViewStyle.GraphStyle
    .VerticalExaggeration = 4.1
    ' The lowest grid with any part of the section
    ' line in it will have one empty grid between
    ' it and the bottom axis.
    .GridStyle.GridsBelowDatum = 1
    ' Increase the empty space above the section
    ' line to make room for any section line labels.
    .GridStyle.GridsAboveMaxElevation = 2

    ' Show major lines, but not minor lines.
    .GridStyle.MajorHorizontalDisplayStylePlan.Visible =
True
   .GridStyle.MajorVerticalDisplayStylePlan.Visible = True
    .GridStyle.MinorHorizontalDisplayStylePlan.Visible =
False
    .GridStyle.MinorVerticalDisplayStylePlan.Visible =
False

    ' Move the title above the top axis.
    dOffset(0) = 0#
    dOffset(1) = 0.045
    .TitleStyle.Offset = dOffset
    ' Set the title to display the station number of each
    ' section.
    Dim sTmp as String
   sTmp = "EG at Station <[Section View Station(Uft|FS)]>"
    .TitleStyle.Text = sTmp
End With
```

Sample Program

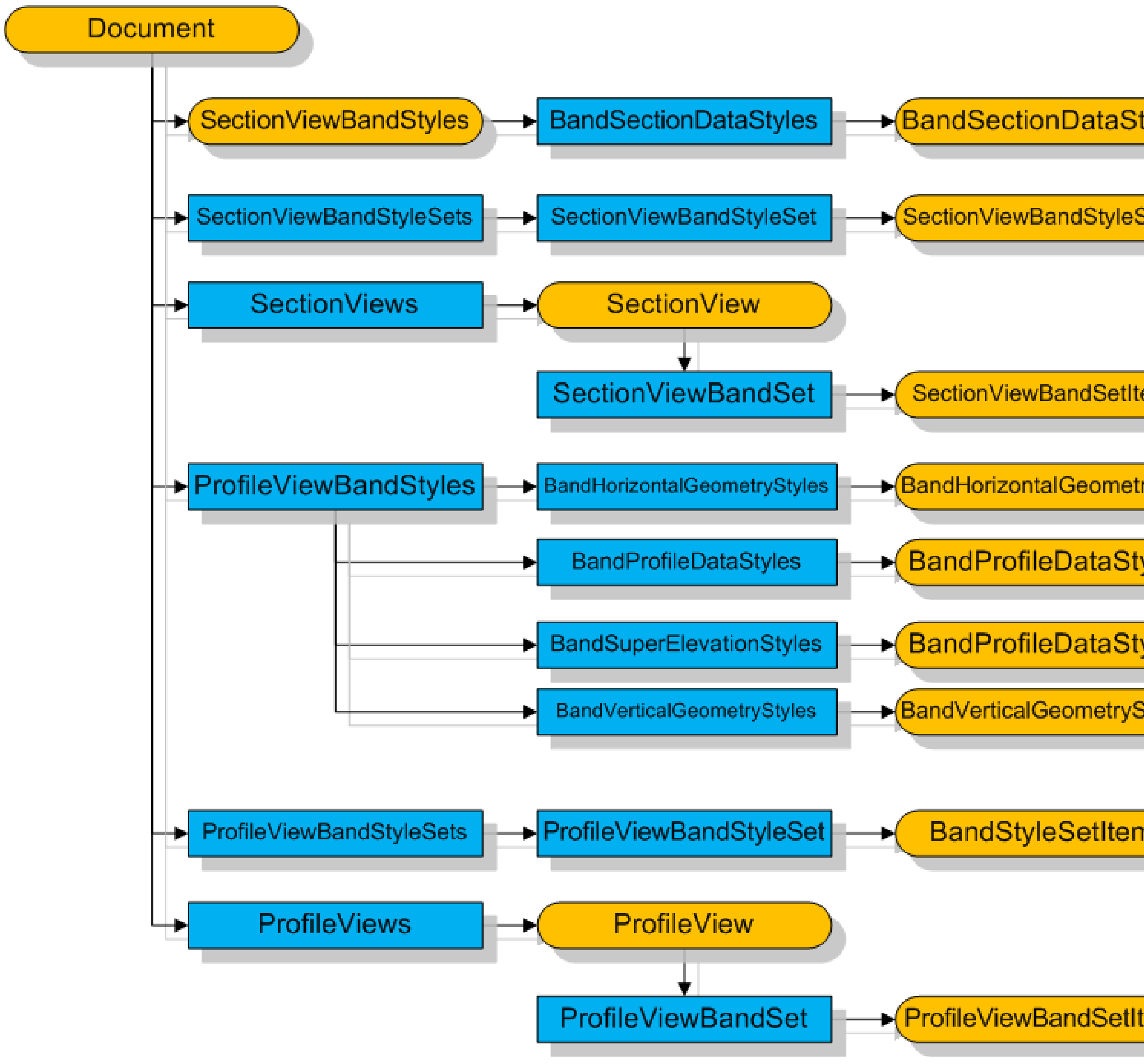
SectionSample.dvb

<installation-directory>\Sample\Civil 3D API\Vba\Section\SectionSample.dvb

This sample program creates a surface from a file and an alignment from a polyline. Sample lines (using group sample lines, individual sample lines, and sample lines based on polyline entities) are created across the alignment. Sections are created from each sample line, and section views of each surface cross section are drawn. Styles and label styles for sample lines and section views are created, as are data bands for the section view graphs.

Data Bands in COM

Object Hierarchy



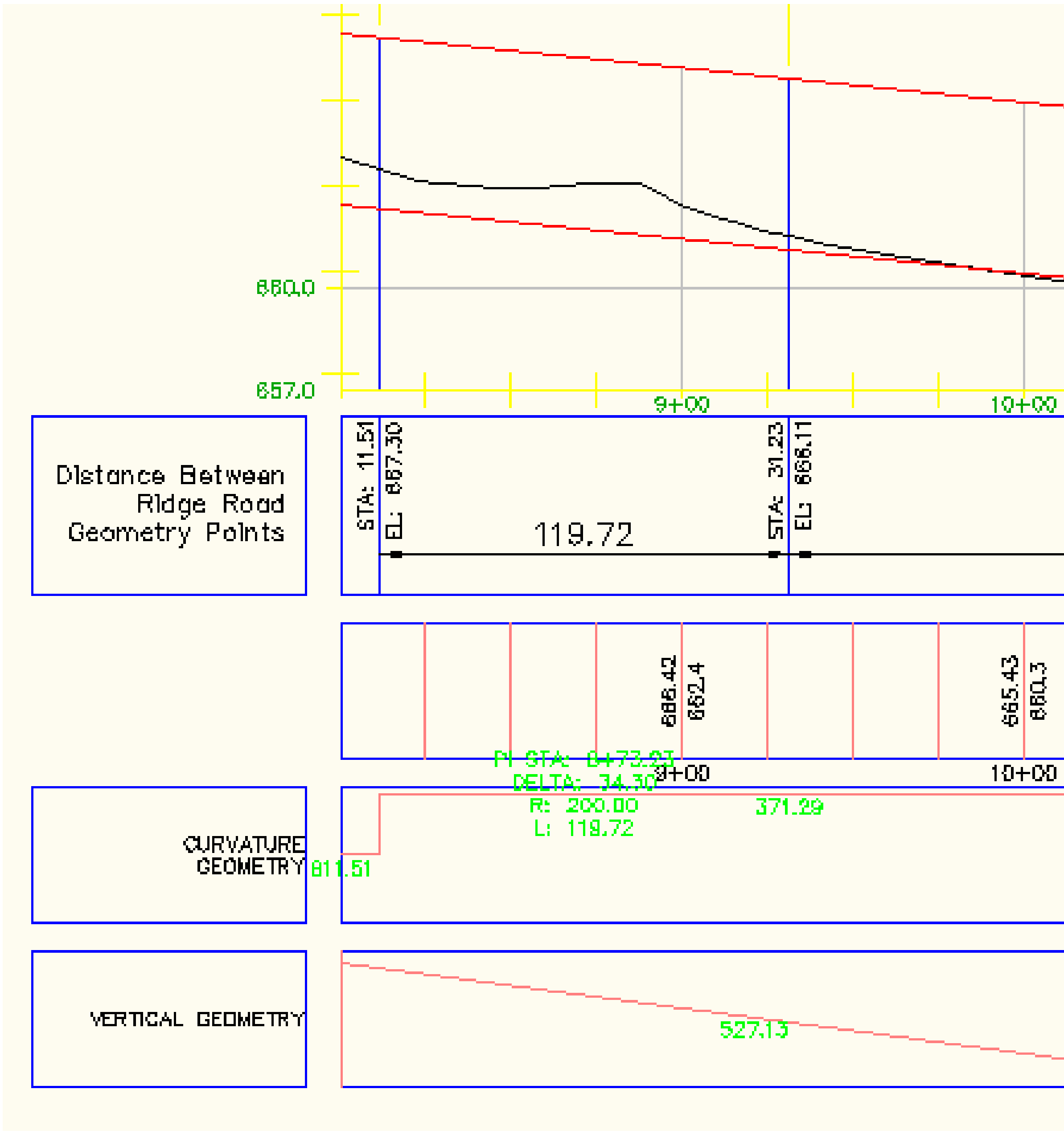
Object Model for Data Bands

Defining a Data Band Style

This section explains the creation and definition of data band style objects. These objects are used with profile view and section view graphs and represent a single band of graphical and text information.

Data Band Concepts

Data bands are a way to display more information with profile views and section views, including differences between sections, profile geometry, and superelevation along an alignment. Data bands consist of one or more strips at the top or bottom of each graph with tick marks, graphics, and labels describing particular features of the subject of the graph.



The bottom of a profile view with four data bands

A band is described by an object derived from the `AeccBandStyle` type: `AeccBandSegmentDataStyle`, `AeccBandProfileDataStyle`, `AeccBandHorizontalGeometryStyle`, `AeccBandVerticalGeometryStyle`, or `AeccBandSuperElevationStyle`. All such styles in the document are stored in collections depending on the band type:

Band Style Type	**Collection of Band Style Objects**
AeccBandSegmentDataStyle	AeccDocument.SectionViewBandStyles. SectionDataBandStyles
AeccBandProfileDataStyle	AeccDocument.ProfileViewBandStyles. ProfileDataBandStyles
AeccBandHorizontalGeometryStyle	AeccDocument.ProfileViewBandStyles. HorizontalGeometryBandStyles
AeccBandVerticalGeometryStyle	AeccDocument.ProfileViewBandStyles. VerticalGeometryBandStyles
AeccBandSuperElevationStyle	AeccDocument.ProfileViewBandStyles. SuperElevationBandStyles

Each collection has an `Add` method for creating new band styles.

The location of information displayed in the band depends on which band style objects are visible. Each data location (for example, at profile stations or at horizontal geometry points) consists of multiple style elements (text labels, tick marks, lines, or blocks). Different band styles have different locations where information can be displayed, and will display information with different graphical effects.

Each information location is managed by a set of three style objects that control:

- The visual style of the text label (properties ending with “LabelDisplayStylePlan“, objects of type `AeccDisplayStyle`).
- The contents and nature of the text , tick marks, lines, and blocks (properties ending with “LabelStyle“, objects of type `AeccBandLabelStyle`).
- The visual style of the tick mark (properties ending with “TickDisplayStylePlan“, objects of type `AeccDisplayStyle`).

The display styles have priority over the label style when setting the color or linetype.

NOTE

If either the display style or the label style element is not set to be visible, then the data element is not visible.

A title can be displayed on the left side of each band. The style of the text and box around the text are controlled by the `AeccBandStyle.TitleBoxTextDisplayStylePlan` and `AeccBandStyle.TitleBoxDisplayStylePlan` properties, both object of type `AeccDisplayStyle`. The text of the title and its location are controlled through the `AeccBandStyle.TitleStyle` property, an object of type `AeccBandTitleStyle`. The `AeccBandTitleStyle.Text` property contains the actual string to be displayed, which may include property fields from the following list:

Valid property fields for AeccBandTitleStyle.Text
<[Parent Alignment(CP)]>
<[Section1 Name(CP)]>
<[Section1 Type(CP)]>
<[Section1 Left Length(Uft|P3|RN|AP|Sn|OF)]>
<[Section1 Right Length(Uft|P3|RN|AP|Sn|OF)]>
<[Section2 Name(CP)]>
<[Section2 Type(CP)]>
<[Section2 Left Length(Uft|P3|RN|AP|Sn|OF)]>
<[Section2 Right Length(Uft|P3|RN|AP|Sn|OF)]>
<[Sample Line Name(CP)]>

Valid property fields for AeccBandTitleStyle.Text
<[Sample Line Group(CP)]>
<[Sample Line Number(Sn)]>
<[Profile1 Name(CP)]>
<[Profile2 Name(CP)]>

The following code sets the title for a section view data band showing two sections:

```
oBandSectionDataStyle.TitleStyle.Text = _
  "<[Section1 Name(CP)]> and <[Section1 Name(CP)]>"
```

Profile Data Band Style

Data bands for general information about profiles are defined by the `AeccBandProfileDataStyle` type. Information in this data band can be displayed at the major and minor grid marks of the base graph, at points where the alignment station equation changes, and at points where the vertical or horizontal geometry change.

Each label can use any of the following property fields:

Valid property fields for use with AeccBandLabelStyle text components
<[Station Value(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Raw Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Profile1 Elevation(Uft|P2|RN|AP|Sn|OF)]>
<[Profile2 Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[Profile1 Elevation Minus Profile2 Elevation(Uft|P3|RN|AP|Sn|OF)]>

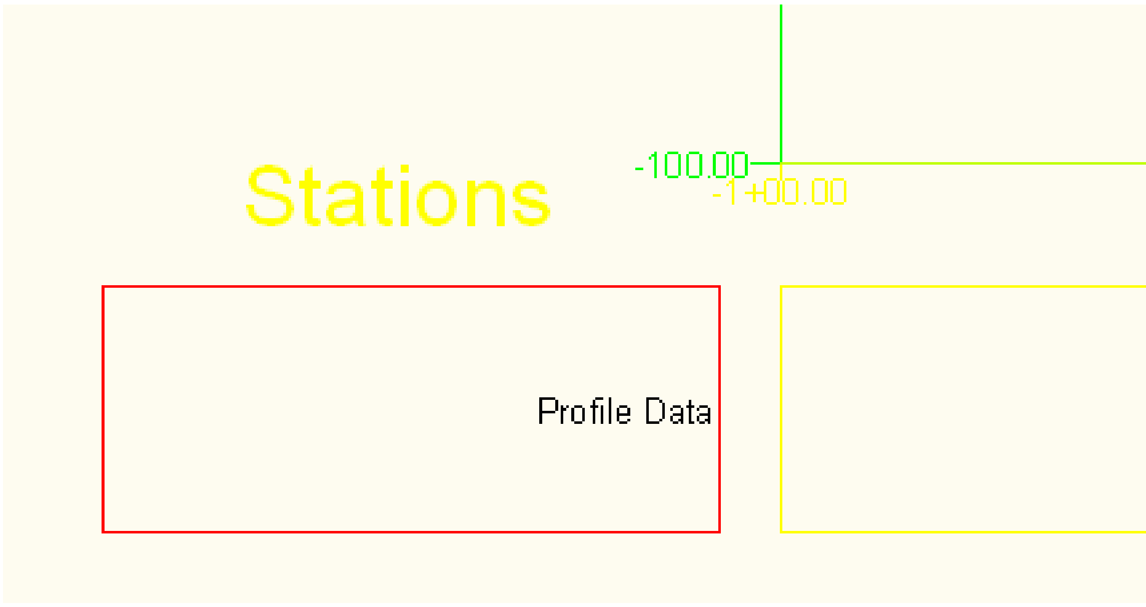
Valid property fields for use with AeccBandLabelStyle text components
<[Profile2 Elevation Minus Profile1 Elevation(Uft|P3|RN|AP|Sn|OF)]>

This sample demonstrates the creation of a data band style displaying section elevation data at two different locations:

```
Dim oBandProfileDataStyle As AeccBandProfileDataStyle
Set oBandProfileDataStyle = oDocument.ProfileViewBandStyles
 _
  .ProfileDataBandStyles.Add("Profile Band")


With oBandProfileDataStyle
    ' Add ticks and labels to each horizontal
    ' geography location.
    .HGPLabelDisplayStylePlan.Visible = True
    .HGPTickDisplayStylePlan.Color = 10 ' red
    .HGPTickDisplayStylePlan.Visible = True
    .HGPLabelStyle.TextComponents.Item(0).Contents = _
     "<[Station Value(Uft|FS|P0|RN|AP|Sn|TP|B2|EN|W0|OF)]>"
    .HGPLabelStyle.TextComponents.Item(0).Color = 11 ' red
    .HGPLabelStyle.TextComponents.Item(0).Visibility = True

    ' Modify how the title is displayed.
    .TitleBoxDisplayStylePlan.Color = 10 ' red
    .TitleBoxDisplayStylePlan.Linetype = "DOT"
    .TitleBoxDisplayStylePlan.Visible = True
    .TitleBoxTextDisplayStylePlan.Color = 80 ' green
    .TitleBoxTextDisplayStylePlan.Visible = True
    .TitleStyle.Text = "Profile Info"
    .TitleStyle.TextHeight = 1.0
    .TitleStyle.TextBoxWidth = 2.0

    ' Hide the rest of the information locations.
    .VGPLabelStyle.TextComponents.Item(0).Visibility =
False
    .MajorIncrementLabelStyle.TextComponents.Item(0). _
      Visibility = False
    .MajorStationLabelDisplayStylePlan.Visible = False
    .MajorTickDisplayStylePlan.Visible = False
    .MinorIncrementLabelStyle.TextComponents.Item(0). _
```

```
          Visibility = False
        .MinorStationLabelDisplayStylePlan.Visible = False
        .MinorTickDisplayStylePlan.Visible = False
        .VGPLabelDisplayStylePlan.Visible = False
        .VGPTickDisplayStylePlan.Visible = False
        .StationEquationLabelStyle.TextComponents.Item(0). _
          Visibility = False
        .StationEquationLabelDisplayStylePlan.Visible = True
        .StationEquationTickDisplayStylePlan.Visible = True
  End With
```

This style produces a data band that looks like this:



Horizontal Geometry Data Band Style

The `AeccBandHorizontalGeometryStyle` type is used to display features of the horizontal geometry of alignments in profile views. Tangents and curves in the alignment are displayed as stylized line segments and curve segments, and a label can be displayed over each segment.

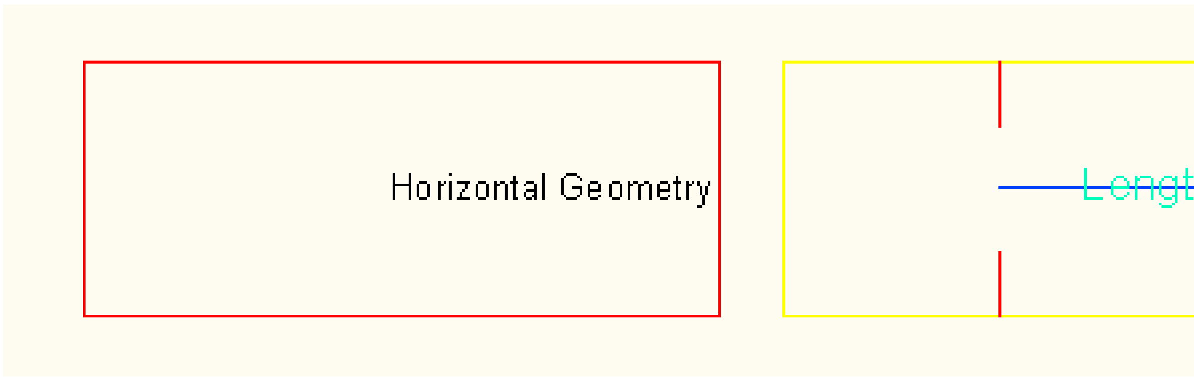
Each label style can use any of the following property fields:

Valid property fields for use with AeccBandLabelStyle text components
<[Length(Uft|P2|RN|Sn|OF|AP)]>
<[Tangent Direction(Udeg|FDMSdSp|MB|P6|RN|DSn|CU|AP|OF)]>
<[Start Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Start Easting(Uft|P4|RN|AP|Sn|OF)]>
<[Start Northing(Uft|P4|RN|AP|Sn|OF)]>
<[End Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[End Easting(Uft|P4|RN|AP|Sn|OF)]>
<[End Northing(Uft|P4|RN|AP|Sn|OF)]>

This sample demonstrates the creation of a data band style displaying information about alignment geometry with a title:

```
Dim oBandProfileDataStyle As AeccBandProfileDataStyle
Set oBandProfileDataStyle = oDocument.ProfileViewBandStyles
 _
  .ProfileDataBandStyles.Add("Horizontal Band")


With oBandHorizontalGeometryStyle
    ' Add displays and labels for alignment tangents.
    .TangentGeometryDisplayStylePlan.Visible = True
    .TangentGeometryDisplayStylePlan.Color = 160 ' blue
    .TangentLabelDisplayStylePlan.Visible = True
    .TangentLabelStyle.TextComponents.Item(0).Contents =
_
      "Length = <[Length(Uft|P2|RN|Sn|OF|AP)]>"
    .TangentLabelStyle.TextComponents.Item(0).Color = 120
    .TangentLabelStyle.TextComponents.Item(0). _
      Visibility = True
    ' Add displays and labels for alignment curves.
    .CurveGeometryDisplayStylePlan.Visible = True
    .CurveGeometryDisplayStylePlan.Color = 160 ' blue
    .CurveLabelDisplayStylePlan.Visible = True
    .CurveLabelStyle.TextComponents.Item(0).Contents = _
      "Length = <[Length(Uft|P2|RN|Sn|OF|AP)]>"
    .CurveLabelStyle.TextComponents.Item(0).Color = 120
    .CurveLabelStyle.TextComponents.Item(0). _
      Visibility = True

    ' Add tick marks at each horizontal geometry point,
    ' the location where different segments of the
    ' alignment meet.
    .TickDisplayStylePlan.Color = 10 ' red
    .TickDisplayStylePlan.Visible = True

    ' Modify how the title is displayed.
    .TitleBoxDisplayStylePlan.Color = 10 ' red
    .TitleBoxDisplayStylePlan.Linetype = "DOT"
    .TitleBoxDisplayStylePlan.Visible = True
    .TitleBoxTextDisplayStylePlan.Color = 80 ' green
    .TitleBoxTextDisplayStylePlan.Visible = True
    .TitleStyle.Text = "Alignment Geometry"
    .TitleStyle.TextHeight = 0.0125
    .TitleStyle.TextBoxWidth = 0.21
    ' Hide the rest of the information locations and
    ' graphical displays.
    .SpiralGeometryDisplayStylePlan.Visible = False
    .SpiralLabelDisplayStylePlan.Visible = False
End With
```

This style produces a data band that looks like this:



Vertical Geometry Data Band Style

The `AeccBandProfileDataStyle` type is used to display features of the vertical geometry of alignments in profile views. The style of graphical markers displayed at each curve and tangent segment can be modified, as well as the labels placed at crest, sag, uphill, and downhill segments of the profile.

Downhill and uphill labels can use any of the following property fields:

Valid property fields for use with AeccBandLabelStyle text components
<[Tangent Horizontal Length(Uft|P2|RN|Sn|OF|AP)]>
<[Tangent Slope Length(Uft|P2|RN|AP|Sn|OF)]>
<[Tangent Grade(FP|P2|RN|AP|Sn|OF)]>
<[Tangent Start Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Tangent Start Elevation(Uft|P2|RN|AP|Sn|OF)]>
<[Tangent End Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Tangent End Elevation(Uft|P2|RN|AP|Sn|OF)]>
<[Tangent Elevation Change(Uft|P2|RN|AP|Sn|OF)]>

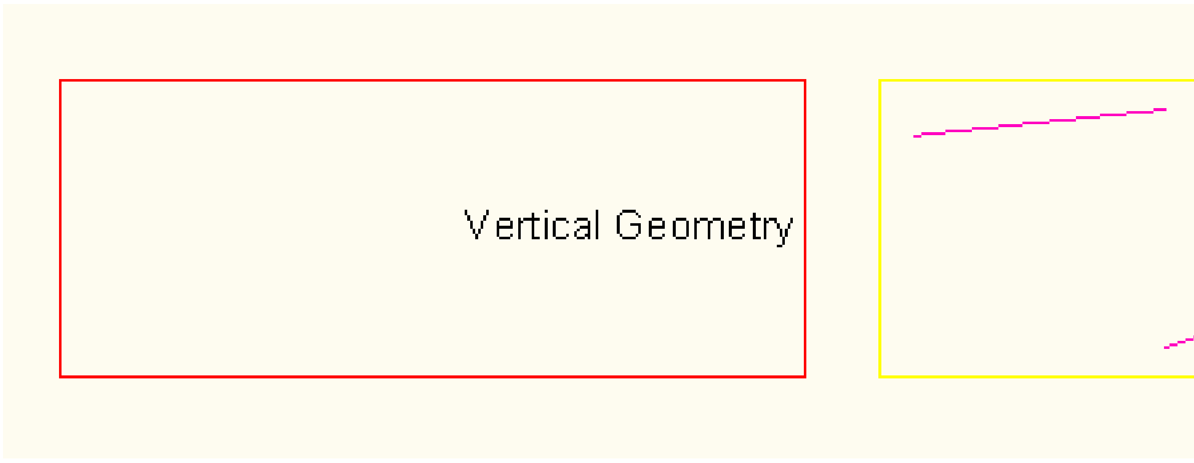
Valid property fields for use with AeccBandLabelStyle text components
<[PVI Before Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[PVI Before Elevation(Uft|P2|RN|AP|Sn|OF)]>
<[PVI After Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[PVI After Elevation(Uft|P2|RN|AP|Sn|OF)]>

This sample demonstrates the creation of a data band style displaying the direction of slope for all segments of a profile with a title:

```
Dim oBandVerticalGeometryStyle As
AeccBandVerticalGeometryStyle
Set oBandVerticalGeometryStyle =
oDocument.ProfileViewBandStyles _
  .VerticalGeometryBandStyles.Add("Vertical Band")


 With oBandVerticalGeometryStyle
    ' Add graphical marks that show the uphill or downhill
    ' directions and the lengths of the vertical segments
    ' of the profile.  On uphill sections the label of the
    ' length of the segment will be in white, on downhill
    ' it will be pale yellow. The graphical element that
    ' shows direction will be pink.
    .DownhillTangentLabelStyle.TextComponents.Item(0).
      Contents = "<[Tangent Horizontal Length(Uft|P2)]>"
    .DownhillTangentLabelStyle.TextComponents.Item(0). _
      Color = 51 ' pale yellow
    .DownhillTangentLabelStyle.TextComponents.Item(0). _
      Visibility = True
    .UphillTangentLabelStyle.TextComponents.Item(0).
      Contents = "<[Tangent Horizontal Length(Uft|P2)]>"
    .UphillTangentLabelStyle.TextComponents.Item(0). _
      Color = 255  ' white
    .UphillTangentLabelStyle.TextComponents.Item(0). _
      Visibility = True
    .TangentGeometryDisplayStylePlan.Color = 220 ' pink
    .TangentGeometryDisplayStylePlan.Visible = True
```

```
        .TangentLabelDisplayStylePlan.Visible = True
        .TangentGeometryDisplayStylePlan.Visible = True

        ' Modify how the title is displayed.
        .TitleBoxDisplayStylePlan.Color = 10 ' red
        .TitleBoxDisplayStylePlan.Linetype = "DOT"
        .TitleBoxDisplayStylePlan.Visible = True
        .TitleBoxTextDisplayStylePlan.Color = 80 ' green
        .TitleBoxTextDisplayStylePlan.Visible = True
        .TitleStyle.Text = "Profile Geometry"
        .TitleStyle.TextHeight = 0.0125
        .TitleStyle.TextBoxWidth = 0.21

        ' Hide the rest of the information locations and
        ' graphical displays.
        .CurveGeometryDisplayStylePlan.Visible = False
        .CurveLabelDisplayStylePlan.Visible = False
        .TickDisplayStylePlan.Visible = False
    End With
```

This style produces a data band that looks like this:



Superelevation Data Band Style

An `AeccBandProfileDataStyle` data band displays information related to the alignment superelevation. It can display slopes of superelevation elements as lines, the distance above or below the centerline representing the amount of slope. The superelevation elements that can be represented this way are:

- Left inside pavement
- Left inside shoulder
- Left outside pavement
- Left outside shoulder
- Right inside pavement
- Right inside shoulder
- Right outside pavement
- Right outside shoulder

You can also display a reference line through the center of the data band to help users interpret the element lines.

The data band can also display tick marks and text labels at points of change in the superelevation of the alignment. The following can be marked on the data band:

- Full superelevation
- Level crown
- Normal crown
- Reverse crown
- Shoulder break over
- Transition segment

The label styles for text labels can use any of the following property fields:

Valid property fields for use with AeccBandLabelStyle text components
<[Station Value(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Superelevation critical point text(CP)]>
<[Cross slope - Left outside pavement(FP|P2|RN|AP|Sn|OF)]>
<[Cross slope - Left inside pavement(FP|P2|RN|AP|Sn|OF)]>
<[Cross slope - Right outside pavement(FP|P2|RN|AP|Sn|OF)]>

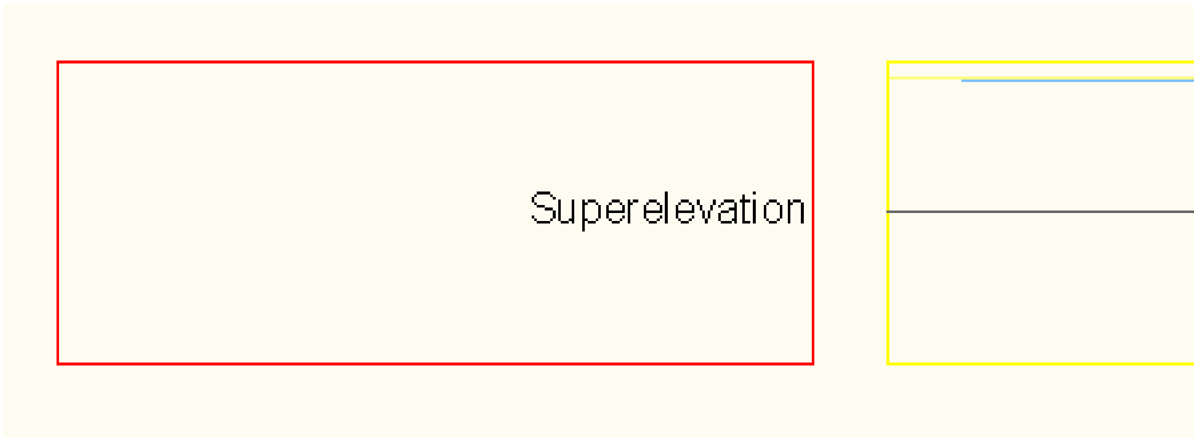
Valid property fields for use with AeccBandLabelStyle text components
<[Cross slope - Right inside pavement(FP|P2|RN|AP|Sn|OF)]>
<[Cross slope - Left outside shoulder(FP|P2|RN|AP|Sn|OF)]>
<[Cross slope - Left inside shoulder(FP|P2|RN|AP|Sn|OF)]>
<[Cross slope - Right outside shoulder(FP|P2|RN|AP|Sn|OF)]>
<[Cross slope - Right inside shoulder(FP|P2|RN|AP|Sn|OF)]>

This sample demonstrates the creation of a data band style displaying the slopes of the outside shoulders - the right shoulder in yellow and the left in blue. A gray reference line is also added.

```
Dim oBandSuperElevationStyle As AeccBandSuperElevationStyle
Set oBandSuperElevationStyle =
oDocument.ProfileViewBandStyles _
  .SuperElevationBandStyles.Add("Superelevation Band")


With oBandSuperElevationStyle
   ' Add graphical display of the slope of the left and
right
   ' outside shoulders.  If the line is above the
centerline,
   ' then the slope is positive.
   .LeftOutsideShoulderLineDisplayStylePlan.Visible = True
   .LeftOutsideShoulderLineDisplayStylePlan.color = 151
   ' Color 151 = pale blue
   .RightOutsideShoulderLineDisplayStylePlan.Visible = True
   .RightOutsideShoulderLineDisplayStylePlan.color = 51
   ' Color 51 = pale yellow

   ' Add a reference line through the center of the data
band.
   .ReferenceLineDisplayStylePlan.Visible = True
   .ReferenceLineDisplayStylePlan.color = 252 ' gray
```

```
    ' Modify how the title is displayed.
    .TitleBoxDisplayStylePlan.color = 10 ' red
    .TitleBoxDisplayStylePlan.Linetype = "DOT"
    .TitleBoxDisplayStylePlan.Visible = True
    .TitleBoxTextDisplayStylePlan.color = 80 ' green
    .TitleBoxTextDisplayStylePlan.Visible = True
    .TitleStyle.Text = "Profile Geometry"
    .TitleStyle.TextHeight = 0.0125
    .TitleStyle.TextBoxWidth = 0.21

    ' Hide the rest of the information locations and
    ' graphical displays.
    .FullSuperLabelDisplayStylePlan.Visible = False
    .FullSuperTickDisplayStylePlan.Visible = False
    .LeftInsidePavementLineDisplayStylePlan.Visible = False
    .LeftInsideShoulderLineDisplayStylePlan.Visible = False
    .LeftOutsidePavementLineDisplayStylePlan.Visible = False
    .LevelCrownLabelDisplayStylePlan.Visible = False
    .LevelCrownTickDisplayStylePlan.Visible = False
    .NormalCrownLabelDisplayStylePlan.Visible = False
    .NormalCrownTickDisplayStylePlan.Visible = False
    .ReverseCrownLabelDisplayStylePlan.Visible = False
    .ReverseCrownTickDisplayStylePlan.Visible = False
    .RightInsidePavementLineDisplayStylePlan.Visible = False
    .RightInsideShoulderLineDisplayStylePlan.Visible = False
    .RightOutsidePavementLineDisplayStylePlan.Visible =
False
    .ShoulderBreakOverLabelDisplayStylePlan.Visible = False
    .ShoulderBreakOverTickDisplayStylePlan.Visible = False
    .TransitionSegmentLabelDisplayStylePlan.Visible = False
End With
```

This style produces a data band that looks like this:



Section Data Band Style

Data bands for section views are described by an object of type `AeccBandSectionDataStyle`. Information in this data band can be displayed at major and minor tick marks, at the centerline, at each section grade break, and at each sample line vertex. The centerline is the location where the sample line crosses the alignment. If the sample line does not cross the alignment, the centerpoint is where the sample line would cross the alignment if the sample line were extended. Unless the `AeccSampleLines.AddByPolyline` method was used to create a multi-segment sample line, placing information at each sample line vertex simply places tick marks and labels at the section endpoints.

Each label style can use any of the following property fields:

Valid property fields for use with AeccBandLabelStyle text components
<[Distance from Centerline(Uft|P2|RN|Sn|OF|AP)]>
<[Distance from Centerline Side(CP)]>
<[Offset from Centerline(Uft|P3|RN|AP|Sn|OF)]>
<[Offset from Centerline Side(CP)]>

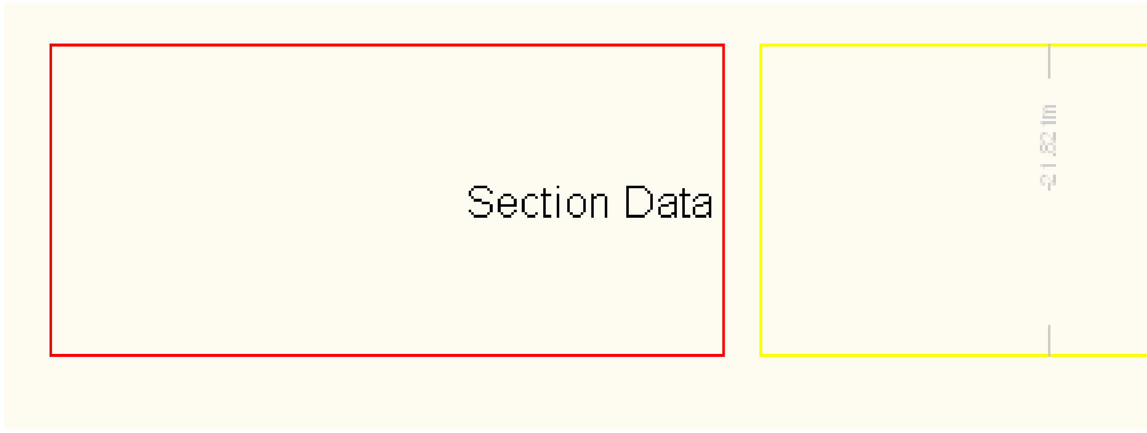
Valid property fields for use with AeccBandLabelStyle text components
<[Section1 Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[Section2 Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[Section1 Elevation Minus Section2 Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[Section2 Elevation Minus Section1 Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[Section Segment Grade In(FP|P2|RN|AP|Sn|OF)]>
<[Section Segment Grade Out(FP|P2|RN|AP|Sn|OF)]>
<[Section Segment Grade Change(FP|P2|RN|AP|Sn|OF)]>

This sample demonstrates the creation of a data band style displaying section elevation data at two different locations:

```
Dim oBandSectionDataStyle As AeccBandSectionDataStyle
Set oBandSectionDataStyle = oDocument.SectionViewBandStyles.
 _
  SectionDataBandStyles.Add("Segment Band")


' Display at every major grid line a tick mark and a label
' that shows the section elevation at that point.
With oBandSectionDataStyle
    .MajorOffsetLabelDisplayStylePlan.Color = 255 ' white
    .MajorOffsetLabelDisplayStylePlan.Visible = True
    .MajorOffsetTickDisplayStylePlan.Color = 255 ' white
    .MajorOffsetTickDisplayStylePlan.Visible = True
End With
With oBandSectionDataStyle.MajorIncrementLabelStyle. _
  TextComponents.Item(0)
    .Contents = "<[Section1 Elevation(Um|P3|RN|AP|Sn|OF)]>m"
    .Color = 255 ' white
    .Visibility = True
    ' Shift the label to the high side of the band.
    .YOffset = 0.015
```

```
End With


' Display a red tick mark and a red label showing section

' elevation at each vertex endpoint in the sample line.
' Make the tick mark large, and only at the top of the
' band.
With oBandSectionDataStyle
    .SampleLineVertexLabelDisplayStylePlan.Color = 20 '
red
    .SampleLineVertexLabelDisplayStylePlan.Visible = True
    .SampleLineVertexTickDisplayStylePlan.Color = 20 ' red
    .SampleLineVertexTickDisplayStylePlan.Visible = True
End With
With oBandSectionDataStyle.SampleLineVerticesLabelStyle.
_
  TextComponents.Item(0)
    .Contents = "<[Section1 Elevation(Um|P3|RN|AP|Sn|OF)]>m"
    .Color = 20 ' red
    .Visibility = True
    .YOffset = 0.08
End With
With oBandSectionDataStyle.SampleLineVerticesLabelStyle.
_
  TickStyle
    .IncrementSmallTicksAtTop = True
    .IncrementSmallTicksAtMiddle = False
    .IncrementSmallTicksAtBottom = False
    .SmallTicksAtTopSize = 0.015
End With


' Hide all other data locations in the data band.
With oBandSectionDataStyle
    .CenterLineLabelDisplayStylePlan.Visible = False
    .CenterLineTickDisplayStylePlan.Visible = False
    .GradeBreakLabelDisplayStylePlan.Visible = False
    .GradeBreakTickDisplayStylePlan.Visible = False
    .MinorOffsetLabelDisplayStylePlan.Visible = False
    .MinorOffsetTickDisplayStylePlan = False
End With
```

This style produces a data band that looks like this:



Creating a Data Band Set

This section explains data band sets, which are groups of individual data bands displayed around a profile view or section view graph.

Creating Data Band Sets for Profile Views

Individual band styles can be grouped together into a set, which can then be assigned to a graph. Profile band sets are `AeccProfileViewBandStyleSet` objects stored in the `AeccDocument.ProfileViewBandStyleSet` collection.

The following example demonstrates creating a profile band style set and adding a band style to it:

```
Dim oProfileViewBandStyleSet As AeccProfileViewBandStyleSet
Set oProfileViewBandStyleSet = _
  oDocument.ProfileViewBandStyleSets.Add("Profile Band
set")

' Add a band style we have already created to the
' band set.
Call oProfileViewBandStyleSet.Add(oBandProfileDataStyle)

' Now we have a band set consisting of one band.
```

Data band sets are used when profile views are first created. The following sample code is taken from the topic Creating a Profile View (page 303), but this time a data band set is passed in the last parameter.

```
Set oProfileView = oAlignment.ProfileViews.Add( _
   "Profile Style 01", _
   "0", _
   dOriginPt, _
   oProfileViewStyle, _
   oProfileViewBandStyleSet)
```

Creating Data Band Sets for Section Views

Individual band styles can be grouped together into a set, which can then be assigned to a graph. Band sets for section graphs are objects of type `AeccSectionViewBandStyleSet`, and all such sets are stored in the `AeccDocument.SectionViewBandStyleSet` collection.

The following example demonstrates creating a section band style set and adding a band style to it:

```
Dim oSectionViewBandStyleSet As AeccSectionViewBandStyleSet
Set oSectionViewBandStyleSet = _
  oDocument.SectionViewBandStyleSets.Add("Section Band
Set")

' Add a band style we have already created to the
' band set.
Call oSectionViewBandStyleSet.Add(oBandSectionDataStyle)

' Now we have a band set consisting of one band.
```

Data band sets are used when section views are first created. The following sample code is taken from the Creating Section Views (page 318) section of the Sections (page 308) chapter, but this time a data band set is passed in the last parameter:

```
Set oSectionView=oSampleLines.Item(j).SectionViews.Add( _
   "Section View" & CStr(j), _
   "0", _
   dOriginPt, _
```

```
    oSectionViewStyle, _
    oSectionViewBandStyleSet)
```

Using Data Bands

This section explains how data band sets are added to profile view and section view graphs.

Adding Data Bands to a Profile View

Every profile view contains a collection of bands in its `AeccProfileView.BandSet` property, an object of type `AeccProfileViewBandSet`. When a profile view is first created, all band styles from the `AeccProfileViewBandStyleSet` parameter are added to this collection. Individual band styles can be added to a section view through the `AeccProfileViewBandSet` collection using its `Add` or `Insert` methods. Both of these methods take an `AeccProfileDataBandStyle` style, a parent alignment, and two `AeccProfile` objects, allowing comparison between profiles.

TIP

If you only want to display information from a single profile in the band, pass the same profile object to both parameters.

The order of bands in the band set is also the order in which the bands are displayed. `AeccProfileViewBandSet.Add` places the new band at the bottom of the list while `AeccProfileViewBandSet.Insert` places the new band at the specified index.

TIP

You can swap the location of two bands with the `AeccProfileViewBandSet.Swap` method.

This sample adds a data band to a profile view that describes the single profile “oProfile” based on the alignment “oAlignment”:

```
Dim oProfileViewBandSetItem As AeccProfileViewBandSetItem
Set oProfileViewBandSetItem = oProfileView.BandSet.Add( _
  oBandStyle, _
  oAlignment, _
```

```
    oProfile, _
    oProfile)

' Now oProfileView has another data band.
```

Adding Data Bands to a Section View

Every section view contains a collection of bands in its `AeccSectionView.BandSet` property, an object of type `AeccSectionViewBandSet`. When a section view is first created, all band styles from the `AeccSectionViewBandStyleSet` parameter are added to this collection. Individual band styles can be added to a section view through the `AeccSectionViewBandSet` collection using its `Add` or `Insert` methods. Both of these methods take an `AeccSectionDataBandStyle` style and two `AeccSection` objects, allowing comparison between sections.

TIP

If you only want to display a single section in the band, pass the same section object to both parameters.

The order of bands in the band set is also the order in which the bands are displayed. `AeccSectionViewBandSet.Add` places the new band at the bottom of the list while `AeccSectionViewBandSet.Insert` places the new band at the specified index.

TIP

You can swap the location of two bands with the AeccSectionViewBandSet.Swap method.

This sample adds a data band to a section view that describes the single section "oSection":

```
Dim oSectionViewBandSetItem As AeccSectionViewBandSetItem
Set oSectionViewBandSetItem = oSectionView.BandSet.Add( _
  oBandSectionDataStyle, _
  oSection, _
  oSection)

' Now oSectionView has another data band.
```

Sample Programs

Profiles.dvb

<installation-directory>\Sample\Civil 3D API\Vba\Profile\ProfileSample.dvb

See the `ProfileViewStyle` module for an example of the creation of a data band style, the definition of a data band style set, and the use of that data band set with a profile view.

Sections.dvb

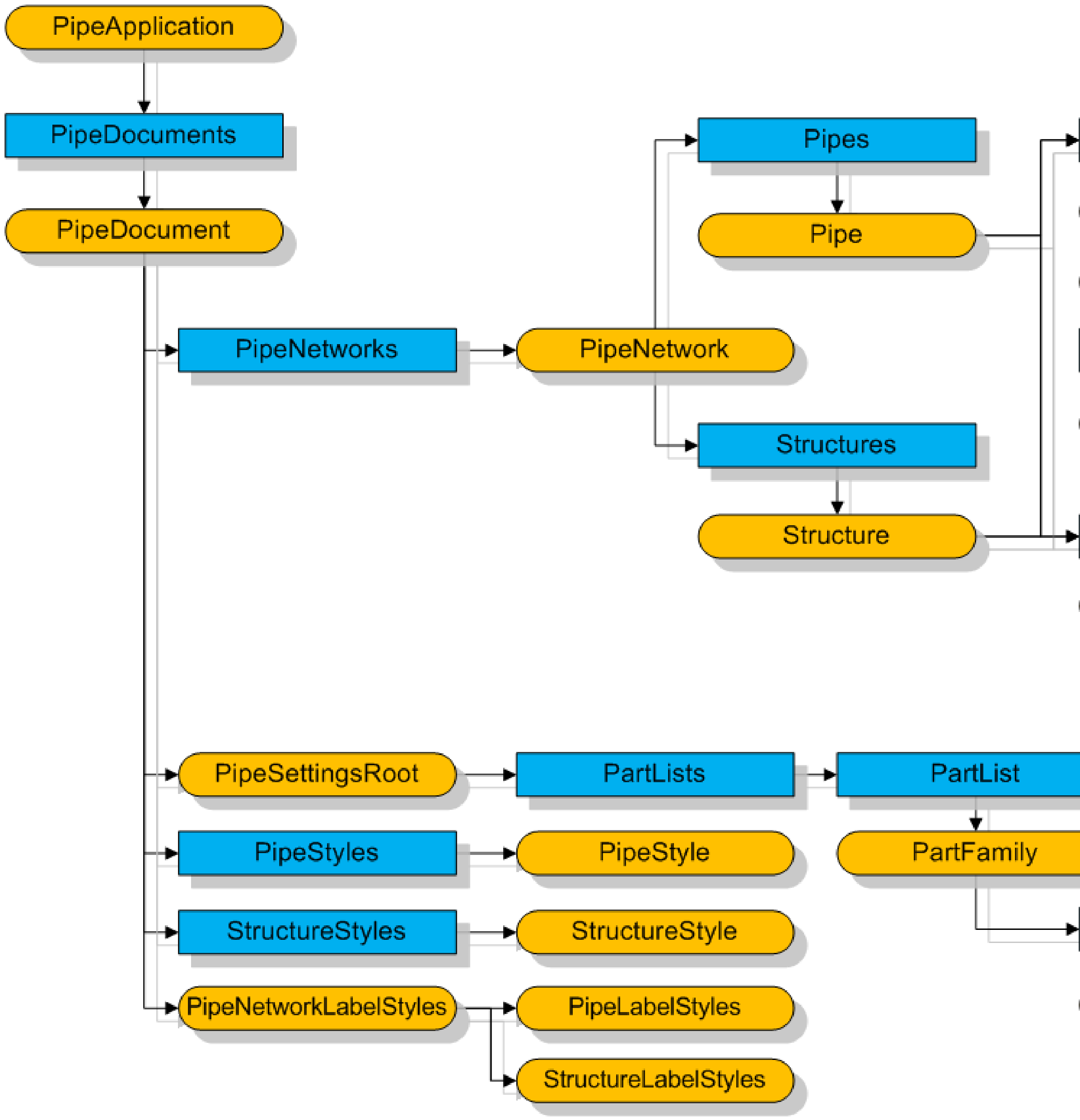
<installation-directory>\Sample\Civil 3D API\Vba\Section\SectionSample.dvb

See the `SectionViewStyle` module for an example of the creation of a data band style, the definition of a data band style set, and the use of that data band set with a section view.

Pipe Networks in COM

This chapter covers working with pipe networks with the COM API. For information about using pipe networks with the .NET API, see Pipe Networks in .NET (page 92).

Object Hierarchy



Base Objects

This section explains how to get the base objects required for using the pipe network API classes.

Accessing Pipe Network-Specific Base Objects

Applications that access pipe networks require special versions of the base objects representing the application and document. The `AeccPipeApplication` object is identical to the `AeccApplication` it is inherited from except that its `AeccPipeApplication.ActiveDocument` property returns an object of type `AeccPipeDocument` instead of `AeccDocument`. The `AeccPipeDocument` object contains collections of pipe network-related items, such as pipe networks, pipe styles, and interference checks. It also contains all of the methods and properties of `AeccDocument`.

When using pipe network root objects, be sure to reference the “Autodesk Civil Engineering Pipe 6.0 Object Library” (AeccXPipe.tlb) and “Autodesk Civil Engineering UI Pipe 6.0 Object Library” (AeccXUIPipe.tlb) libraries.

This sample demonstrates how to retrieve the pipe network root objects:

```
Dim oApp As AcadApplication
Set oApp = ThisDrawing.Application
Dim sAppName As String
sAppName = "AeccXUiPipe.AeccPipeApplication"
Dim oPipeApplication As AeccPipeApplication
Set oPipeApplication = oApp.GetInterfaceObject(sAppName)

' Get a reference to the currently active document.
Dim oPipeDocument As AeccPipeDocument
Set oPipeDocument = oPipeApplication.ActiveDocument
```

Pipe-Specific Ambient Settings

Ambient settings allow you to get and set the units and default property settings of pipe network objects as well as access the catalog of all pipe and structure parts held in the document. Ambient settings for a pipe document are held in the `AeccPipeDocument.Settings` property, an object of type `AeccPipeSettingsRoot`. `AeccPipeSettingsRoot` inherits all the properties of the `AeccSettingsRoot` class.

Among the properties of `AeccPipeSettingsRoot` are `InterferenceSettings`, `PipeSettings`, and `StructureSettings`. Each of these properties consist of an `AeccSettingsAmbient` object, which describes the default units of measurement for interference, pipe, and structure objects. The `AeccPipeSettingsRoot.PipeNetworkSettings` property contains the name of the default styles for pipe and structure objects as well as the default label placement, units, and naming conventions for pipe networks as a whole.

```
' Get the default set of pipe rules used in this document.
With oSettings.PipeNetworkSettings.RulesSettings
    Debug.Print "Using pipe rules:"; .PipeDefaultRules.Value
End With

' Set the default units used for pipes in this document.
With oSettings.PipeSettings.AmbientSettings
     .AngleSettings.Unit = aeccAngleUnitRadian
     .CoordinateSettings.Unit = aeccCoordinateUnitFoot
     .DistanceSettings.Unit = aeccCoordinateUnitFoot
End With
```

The `AeccPipeSettingsRoot` object also has a `PipeNetworkCommandsSettings`property, which contains properties that affect pipe network-related commands. Each sub-property contains an AmbientSettings property which describes the default units of measurement for interference, pipe, and structure objects, plus other properties specific to the command.

Listing and Adding Dynamic Part Properties

Each type of pipe and structure has many unique attributes (such as size, geometry, design, and composition) that cannot be stored in the standard pipe and structure properties. To give each part appropriate attributes, pipe and structure objects have sets of dynamic properties. A single property is represented by an `AeccPartDataField` object. Data fields are held in collections of type `AeccPartDataRecord`. You can reach these collections through the `PartDataRecord` property of `AeccPartSizeFilter`, `AeccPipe`, and `AeccStructure` objects. Each data field contains an internal variable name, a text description of the value, a global context used to identify the field, data type, and the data value itself.

This sample enumerates all the data fields contained in a pipe object “oPipe” and displays information from each field.

```
Dim oPartDataField As AeccPartDataField

Debug.Print "All data fields for this pipe:"
Debug.Print "======"
For Each oPartDataField In oPipe.PartDataRecord
   Debug.Print "Context name: ";
oPartDataField.ContextString
   Debug.Print "Description:  "; oPartDataField.Description
   Debug.Print "Internal name:"; oPartDataField.Name
   Debug.Print "Value:        "; oPartDataField.Tag
   Debug.Print "Type of value:"; oPartDataField.Type
   Debug.Print "------"
Next
```

To create your own dynamic properties, you first create a custom parameter describing the type and name of the property. You do this by using the pipe network catalog definitions object `AeccPipeNetworkCatDef`, which you access through the ambient property `AeccPipeSettingsRoot.PipeNetworkCatDef`. The `AeccPipeNetworkCatDef` object creates new parameters using the `AeccPipeNetworkCatDef.DeclareNewParameter` method. `DeclareNewParameter` takes some strings describing the parameter data type:

- a global context (the identification string used to access the parameter type)
- a context description
- a parameter name (the internally used name of the parameter)
- a parameter description (the public name of the parameter used by the user interface, such as in the **Part Properties** tab of the **Pipe and Structure Properties** dialog boxes).

Once a parameter has been created, it can be made into a property available for use in parts through the `AeccPipeNetworkCatDef.DeclarePartProperty` method.

NOTE

The parameter name cannot contain spaces or punctuation characters.

This sample demonstrates declaring a parameter and making a property based on that parameter available to any pipe objects:

```
Dim oSettings As AeccPipeSettingsRoot
Dim oPipeNetworkCatDef As AeccPipeNetworkCatDef

Set oSettings = oPipeDocument.Settings
Set oPipeNetworkCatDef = oSettings.PipeNetworkCatDef
oPipeNetworkCatDef.DeclareNewParameter _
  "Global Context 01", _
  "Context Description", _
  "TParam", _
  "Test Parameter", _
  aeccDoubleGeneral, _
  aeccDouble, _
  "", _
  True, _
  False

oPipeNetworkCatDef.DeclarePartProperty
  "Global Context 01", aeccDomPipe, 10
```

You can now choose from among those properties available to the part's domain and create a data field.

```
' Make a data field based on the "Global Context 01"
' property and add it to a pipe object "oPipe". Set
' the value of the data field to "6.5".
Dim oPartDataField As AeccPartDataField
Set oPartDataField = oPipe.PartDataRecord.Append
  ("Global Context 01", 0)
oPartDataField.Tag = 6.5
```

Retrieving the Parts List

`AeccPipeSettingsRoot` also contains the `PartLists` property, a read-only collection of all the lists of part types available in the document. Each list is an object of type `AeccPartList`, a read-only collection of part families. A part family represents a broad category of parts, and is identified by a GUID (Globally Unique Identification) value. A part family can only contain parts from one domain - either pipes or structures but not both. Part families contain a read-only collection of part filters (`AeccPartSizeFilter`), which are the particular sizes of parts. A part filter is defined by its `AeccPartSizeFilter.PartDataRecord` property, a collection of fields describing various aspects of the part.

This sample prints the complete listing of all parts in a document.

```
Dim oSettings As AeccPipeSettingsRoot
Set oSettings = oPipeDocument.Settings
' Get a reference to all the parts lists in the drawing.
Dim oPartLists As AeccPartLists
Set oPartLists = oSettings.PartLists
Debug.Print "Number of part lists: "; oPartLists.Count

Dim oPartList As AeccPartList
For Each oPartList In oPartLists
   Dim oPartFamily As AeccPartFamily
   Dim oSizeFilters As AeccPartSizeFilters
   Dim oSizeFilter As AeccPartSizeFilter
   Dim sPipeGuid As String
   Dim sStructureGuid As String
   Dim oPipeFilter As AeccPartSizeFilter
   Dim oStructureFilter As AeccPartSizeFilter

   Debug.Print: Debug.Print
   Debug.Print "PART LIST - "; oPartList.Name
   Debug.Print "------------------------------------------"

   ' From the part list, looking at only those part families
   ' that are pipes, print all the individual parts.
   Debug.Print "  Pipes"
   Debug.Print "  ====="
   For Each oPartFamily In oPartList
      ' Look for only pipe families.
      If (oPartFamily.Domain = aeccDomPipe) Then
         sPipeGuid = oPartFamily.guid
         Debug.Print "  Family: "; oPartFamily.Name
         ' Go through each part in this family.
         For Each oPipeFilter In oPartFamily.SizeFilters
            Debug.Print "    Part: "; oPipeFilter.Name
         Next
      End If
   Next
```

```
    ' From the part list, looking at only those part families
    ' that are structures, print all the individual parts.
    Debug.Print
    Debug.Print "  Structures"
    Debug.Print "  =========="
    For Each oPartFamily In oPartList
       ' Look for only structure families.
       If (oPartFamily.Domain = aeccDomStructure) Then
          sStructureGuid = oPartFamily.guid
          Debug.Print "  Family: "; oPartFamily.Name
          ' Go through each part in this family.
          For Each oPipeFilter In oPartFamily.SizeFilters
             Debug.Print "    Part: "; oPipeFilter.Name
          Next
       End If
    Next
 Next
```

Creating a Pipe Network

A pipe network is a set of interconnected or related parts. The collection of all pipe networks is held in the `AeccPipeDocument.PipeNetworks` property. A pipe network, an object of type `AeccPipeNetwork`, contains the collection of pipes and the collection of structures which make up the network. `AeccPipeNetwork` also contains the method `AeccPipeNetwork.FindShortestNetworkPath` for determining the path between two network parts.

The `AeccPipeNetwork.ReferenceAlignment` is used by pipe and structure label properties. For example, you can create a label that shows the station and offset from the alignment. The `AeccPipeNetwork.ReferenceSurface` is used primarily for Pipe Rules. For example, you can have a rule that places the structure rim at a specified elevation from the surface. Labels may also refer to the `ReferenceSurface` property.

```
' Get the collection of all networks.
Dim oPipeNetworks as AeccPipeNetworks
Set oPipeNetworks = oPipeDocument.PipeNetworks
```

```
' Create a new pipe network
Set oPipeNetwork = oPipeNetworks.Add("Network Name")
```

Pipes

This section explains the creation and use of pipes. Pipes represent the conduits within a pipe network.

Creating Pipes

Pipe objects represent the conduits of the pipe network. Pipes are created using the pipe network's `AeccPipeNetwork.Pipes` collection. This collection has methods for creating either straight or curved pipes. Both methods require you to specify a particular part family (using the GUID of a family) and a particular part size filter object as well as the starting and ending points of the pipe. The order of the start and end points may have meaning in describing flow direction.

This sample creates a straight pipe between two hard-coded points using the first pipe family and pipe size filter it can find in the part list:

```
Dim oPipe as AeccPipe
Dim oSettings As AeccPipeSettingsRoot
Dim oPartLists As AeccPartLists
Dim oPartList As AeccPartList
Dim sPipeGuid As String
Dim oPipeFilter As AeccPartSizeFilter

' Go through the list of part types and select the first
' pipe found.
Set oSettings = oPipeDocument.Settings
' Get all the parts list in the drawing.
Set oPartLists = oSettings.PartLists
' Get the first part list found.
Set oPartList = oPartLists.Item(0)
For Each oPartFamily In oPartList
   ' Look for a pipe family.
   If (oPartFamily.Domain = aeccDomPipe) Then
      sPipeGuid = oPartFamily.guid
      ' Get the first size filter list from the family.
      Set oPipeFilter = oPartFamily.SizeFilters.Item(0)
   Exit For
   End If
Next

Dim dStartPoint(0 To 2) As Double
Dim dEndPoint(0 To 2) As Double
dStartPoint(0) = 100: dStartPoint(1) = 100
dEndPoint(0) = 200: dEndPoint(1) = 100

' Assuming a valid AeccNetwork object "oNetwork".
Set oPipe = oNetwork.Pipes.Add(sPipeGuid, oPipeFilter,
dStartPoint, dEndPoint)
```

Using Pipes

To make a new pipe a meaningful part of a pipe network, it must be connected to structures or other pipes using the `AeccPipe.ConnectToStructure` or `AeccPipe.ConnectToPipe` methods, or structures must be connected to it using the `AeccStructure.ConnectToPipe` method. Connecting pipes together directly creates a new virtual `AeccStructure` object to serve as the joint. If a pipe end is connected to a structure, it must be disconnected before attempting to connect it to a different structure. After a pipe has been connected to a network, you can determine the structures at either end by using the `StartStructure` and `EndStructure` properties or by using the `Connections` property, which is a read-only collection of network parts. There are methods and properties for setting and determining the flow direction, getting all types of physical measurements, and for accessing collections of user-defined properties for custom descriptions of the pipe.

```
' Given a pipe and a structure, join the second endpoint
' of the pipe to the structure.
oPipe.ConnectToStructure aeccPipeEnd, oStructure

' Set the flow direction for the pipe.
oPipe.FlowDirectionMethod =
aeccPipeFlowDirectionMethodBySlope

' Add a custom property to the pipe and assign a value.
Call oPipe.ParamsLong.Add("Custom", 9.2)
Debug.Print "Custom prop:"; oPipe.ParamsLong.Value("Custom")
```

Creating Pipe Styles

A pipe style controls the visual appearance of pipes in a document. All pipe style objects in a document are stored in the `AeccPipeDocument.PipeStyles` collection. Pipe styles have four display methods and three hatch methods for controlling general appearance attributes and three properties for controlling display attributes that are specific to pipes. The methods `DisplayStyleModel|Profile|Section|Plan`, and `HatchStylePlan|Profile|Section` all take a parameter describing the feature being modified, and return a reference to the `AeccDisplayStyle` or `AeccHatchDisplayStyle` object controlling common display attributes, such as line styles and color. The properties `PlanOption` and `ProfileOption` set the size of the inner wall, outer wall, and end lines according to either the physical properties of the pipe, a custom sizes using drawing units, or a certain percentage of its previous drawing size. The `HatchOption` property sets the area of the pipe covered by any hatching used. A pipe object is given a style by assigning the `AeccPipe.Style` property to a `AeccPipeStyle` object.

This sample creates a new pipe style object, sets some of its properties, and assigns it to a pipe object:

```
' Create a new pipe style object.
Dim oPipeStyle As AeccPipeStyle
Set oPipeStyle = oPipeDocument.PipeStyles.Add("Pipe Style
 01")
With oPipeStyle.PlanOption
    ' Set the display size of the pipes in plan view, using
    ' absolute drawing units for the inside, outside, and
    ' ends of each pipe.
    .InnerDiameter = 2.1
    .OuterDiameter = 2.4
    .EndLineSize = 2.1

    ' Indicate that we will use our own measurements for
    ' the inside and outside of the pipe, and not base
    ' the drawing on the actual physical measurements of
    ' the pipe.
    .WallSizeType = aeccUserDefinedWallSize

    ' Indicate what kind of custom sizing to use.
    .WallSizeOptions = aeccPipeUseAbsoluteUnits
End With
```

```
' Modify the colors of pipes using this style when shown
in
' plan view.
oPipeStyle.DisplayStylePlan(aeccDispCompPipeOutsideWalls)
 _
  .Color = 40 ' orange
oPipeStyle.DisplayStylePlan(aeccDispCompPlanInsideWalls)
_
  .Color = 255 ' white
oPipeStyle.DisplayStylePlan(aeccDispCompPipeEndLine) _
  .color = 255 ' white

' Set a pipe to use this style.
Set oPipe.Style = oPipeStyle
```

Creating Pipe Label Styles

The collection of all pipe label styles in a document is found in the `AeccPipeDatabase.PipeNetworkLabelStyles.PipeLabelStyles` property, which is a standard `AeccLabelStyles` object. For more information, see Label Styles (page 211).

NOTE

The label style of a particular pipe cannot be set using the API.

Pipe label styles can use the following property fields in the contents of any text component.

Valid property fields for AeccLabelStyleTextComponent.Contents in pipes
<[Cross Sectional Shape(CP)]>
<[Wall Thickness(Uin|P3|RN|AP|Sn|OF)]>
<[Material(CP)]>
<[Minimum Curve Radius(Uft|P3|RN|AP|Sn|OF)]>
<[Manning Coefficient(P3|RN|AP|Sn|OF)]>

Valid property fields for AeccLabelStyleTextComponent.Contents in pipes
<[Hazen Williams Coefficient(P3|RN|AP|Sn|OF)]>
<[Darcy Weisbach Factor(P3|RN|AP|Sn|OF)]>
<[Inner Pipe Diameter(Uin|P3|RN|AP|Sn|OF)]>
<[Inner Pipe Width(Uin|P3|RN|AP|Sn|OF)]>
<[Inner Pipe Height(Uin|P3|RN|AP|Sn|OF)]>
<[Name(CP)]>
<[Description(CP)]>
<[Network Name(CP)]>
<[Reference Alignment Name(CP)]>
<[Pipe Start Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Start Offset(Uft|P3|RN|AP|Sn|OF)]>
<[Start Offset Side(CP)]>
<[Pipe End Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[End Offset Side(Uft|P3|RN|AP|Sn|OF)]>
<[End Offset(CP)]>
<[Reference Surface Name(CP)]>
<[Pipe Slope(FP|P2|RN|AP|Sn|OF)]>

Valid property fields for AeccLabelStyleTextComponent.Contents in pipes
<[Pipe Start Structure(CP)]>
<[Pipe Start Northing(Uft|P4|RN|AP|Sn|OF)]>
<[Pipe Start Easting(Uft|P4|RN|AP|Sn|OF)]>
<[Start Invert Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[Start Centerline Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[Start Crown Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[Pipe End Structure(CP)]>
<[Pipe End Northing(Uft|P4|RN|AP|Sn|OF)]>
<[Pipe End Easting(Uft|P4|RN|AP|Sn|OF)]>
<[End Invert Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[End Centerline Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[End Crown Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[2D Length - Center to Center(Uft|P3|RN|AP|Sn|OF)]>
<[3D Length - Center to Center(Uft|P3|RN|AP|Sn|OF)]>
<[2D Length - To Inside Edges(Uft|P3|RN|AP|Sn|OF)]>
<[3D Length - To Inside Edges(Uft|P3|RN|AP|Sn|OF)]>
<[Pipe Bearing(Udeg|FDMSdSp|MB|P6|RN|DSn|CU|AP|OF)]>

Valid property fields for AeccLabelStyleTextComponent.Contents in pipes
<[Pipe Start Direction in plan(Udeg|FDMSdSp|MB|P6|RN|DSn|CU|AP|OF)]>
<[Pipe End Direction in plan(Udeg|FDMSdSp|MB|P6|RN|DSn|CU|AP|OF)]>
<[Pipe Radius(Uft|P3|RN|AP|Sn|OF)]>
<[Pipe Chord Length(Uft|P3|RN|AP|Sn|OF)]>
<[Radius Point Northing(Uft|P4|RN|AP|Sn|OF)]>
<[Radius Point Easting(Uft|P4|RN|AP|Sn|OF)]>
<[Minimum Cover(Uft|P3|RN|AP|Sn|OF)]>
<[Maximum Cover(Uft|P3|RN|AP|Sn|OF)]>
<[Pipe Outer Diameter or Width(Uin|P3|RN|AP|Sn|OF)]>
<[Pipe Inner Diameter or Width(Uin|P3|RN|AP|Sn|OF)]>
<[Drop Across Span(Uft|P3|RN|AP|Sn|OF)]>
<[Total Slope Across Span(FP|P2|RN|AP|Sn|OF)]>
<[Number of Pipes in Span(Sn)]>

Structures

This section describes the creation and use of structures. Structures are the connectors within a pipe network.

Creating Structures

Structures represent physical objects such as manholes, catch basins, and headwalls. Logically, structures are used as connections between pipes at pipe endpoints. In cases where two pipes connect directly, an `AeccStructure` object not representing any physical object is still created to serve as the joint. Any number of pipes can connect with a structure. Structures are represented by objects of type `AeccStructure`, which are created by using the `Add` method of the `Surfaces` collection of `AeccPipeNetwork`.

This sample uses the first structure family and size filter it can find in the part list and creates a new structure based on that part type.

```
Dim oStructure as AeccStructure
Dim oSettings As AeccPipeSettingsRoot
Dim oPartLists As AeccPartLists
Dim oPartList As AeccPartList
Dim sStructureGuid As String
Dim oStructureFilter As AeccPartSizeFilter

' Go through the list of part types and select the first
' structure found.
Set oSettings = oPipeDocument.Settings
' Get all the parts list in the drawing.
Set oPartLists = oSettings.PartLists
' Get the first part list found.
Set oPartList = oPartLists.Item(0)
For Each oPartFamily In oPartList
   ' Look for a structure family that is not named
   ' "Null Structure".
   If (oPartFamily.Domain = aeccDomStructure) And _
     (oPartFamily.Name = "Null Structure") Then
      sStructureGuid = oPartFamily.guid
      ' Get the first size filter list from the family.
     Set oStructureFilter = oPartFamily.SizeFilters.Item(0)
      Exit For
   End If
Next

Dim dPoint(0 To 2) As Double
dPoint(0) = 100: dPoint(1) = 100
```

```
' Assuming a valid AeccNetwork object "oNetwork".
Set oStructure = oNetwork.Structures.Add( _
  sStructureGuid, _
  oStructureFilter, _
  dPoint, _
  5.2333) ' 305 degrees in radians
```

Using Structures

To make the new structure a meaningful part of a pipe network, it must be connected to pipes in the network using the `AeccStructure.ConnectToPipe` method or pipes must be connected to it using the `AeccPipe.ConnectToStructure` method. After a structure has been connected to a network, you can determine the pipes connected to it by using the `Connections` property, which is a read-only collection of network parts. There are also methods and properties for setting and determining all types of physical measurements for the structure and for accessing collections of user-defined properties for custom descriptions of the structure.

```
' Given a pipe and a structure, join the second endpoint
' of the pipe to the structure.
oStructure.ConnectToPipe oPipeNew, aeccPipeEnd

' Determine flow directions from all pipes connected
' to a structure.
Dim i As Integer
For i = 0 To oStructure.ConnectedPipesCount - 1
   If (oStructure.IsConnectedPipeFlowingIn(i) = True) Then
      Debug.Print "Pipe "; i; " flows into structure"
   Else
      Debug.Print "Pipe "; i; " does not flow into
structure"
   End If
Next i
```

Creating Structure Styles

A structure style controls the visual appearance of structures in a document. All structure style objects are stored in the `AeccPipeDocument.StructureStyles` collection. Structure styles have four methods for controlling general appearance attributes and three properties for controlling display attributes that are specific to structures. The methods `DisplayStylePlan|Profile|Section|Model` and `HatchStylePlan|Profile|Section` all take a parameter describing the feature being modified and return a reference to the `AeccDisplayStyle` or `AeccHatchDisplayStyle` object controlling common display attributes such as line styles and color. The properties `PlanOption`, `ProfileOption`, and `ModelOption` set the display size of the structure and whether the structure is shown as a model of the physical object or only symbolically. A structure object is given a style by assigning the `AeccStructure.Style` property to a `AeccStructureStyle` object.

This sample creates a new structure style object, sets some of its properties, and assigns it to a structure object:

```
' Create a new structure style object.
Dim oStructureStyle As AeccStructureStyle
Set oStructureStyle =
oPipeDocument.StructureStyles.Add("Structure Style 01")

oStructureStyle.DisplayStylePlan(aeccDispCompStructure).color
 = 30
oStructureStyle.PlanOption.MaskConnectedObjects = True

' Set a structure to use this style.
Set oStructure.Style = oStructureStyle
```

Creating Structure Label Styles

The collection of all structure label styles in a document is found in the `AeccPipeDatabase.PipeNetworkLabelStyles.StructureLabelStyles` property, which is a standard `AeccLabelStyles` object. For more information, see Label Styles (page 211).

NOTE

The label style of a particular structure cannot be set using the API.

Structure label styles can use the following property fields in the contents of any text component:

Valid property fields for AeccLabelStyleTextComponent.Contents in structures
<[Name(CP)]>
<[Description(CP)]>
<[Network Name(CP)]>
<[Structure Rotation Angle(Udeg|FD|P4|RN|AP|OF)]>
<[Reference Alignment Name(CP)]>
<[Structure Station(Uft|FS|P2|RN|AP|Sn|TP|B2|EN|W0|OF)]>
<[Structure Offset(Uft|P3|RN|AP|Sn|OF)]>
<[Structure Offset Side(CP)]>
<[Reference Surface Name(CP)]>
<[Connected Pipes(Sn)]>
<[Structure Northing(Uft|P4|RN|AP|Sn|OF)]>
<[Structure Easting(Uft|P4|RN|AP|Sn|OF)]>
<[Automatic Surface Adjustment]>
<[Insertion Rim Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[Sump Elevation(Uft|P3|RN|AP|Sn|OF)]>
<[Surface Adjustment Value(Uft|P3|RN|AP|Sn|OF)]>

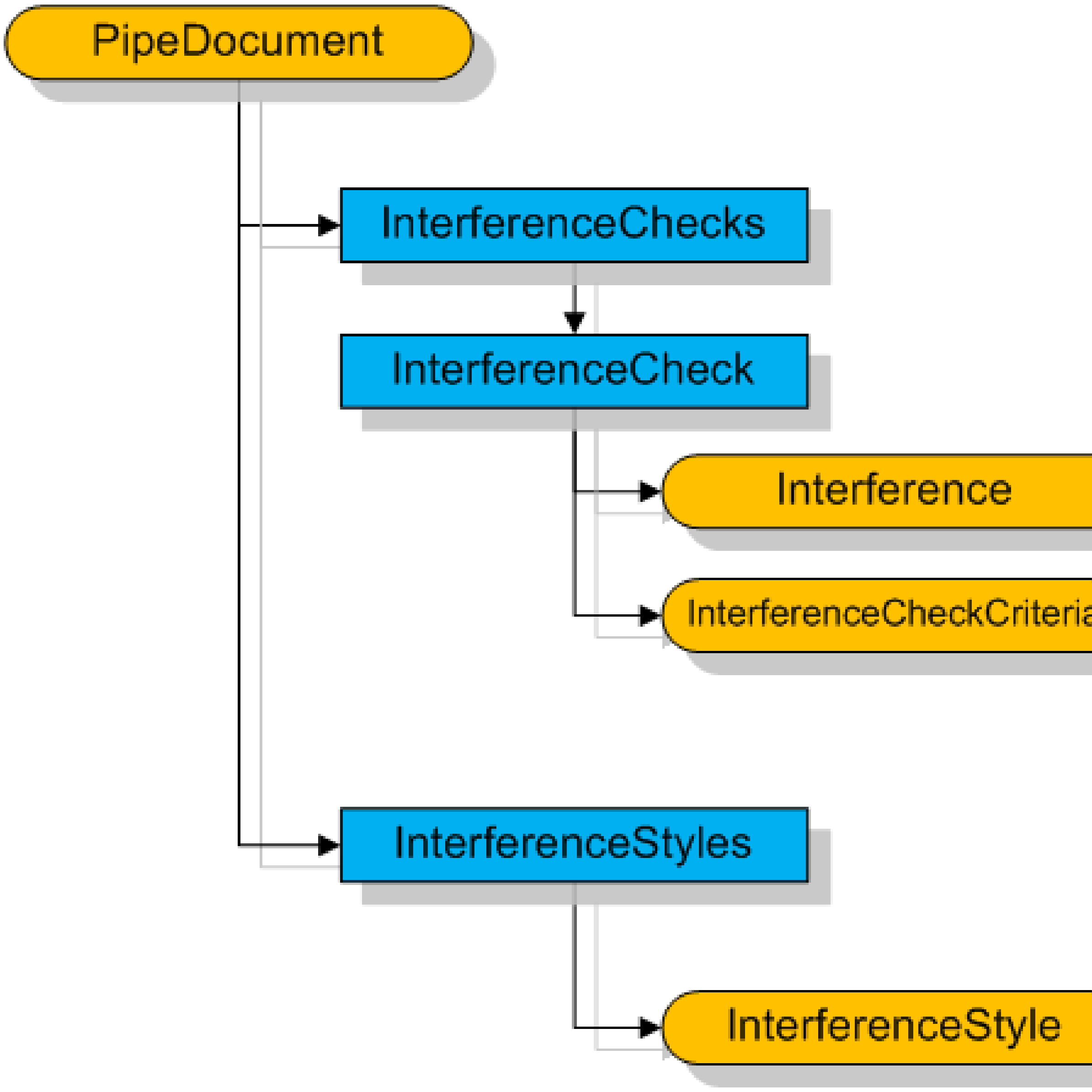
Valid property fields for AeccLabelStyleTextComponent.Contents in structures
<[Control Sump By:(CP)]>
<[Sump Depth(P3|RN|AP|Sn|OF)]>
<[Surface Elevation At Insertion Point(Uft|P3|RN|AP|Sn|OF)]>
<[Structure Shape(CP)]>
<[Vertical Pipe Clearance(Uin|P3|RN|AP|Sn|OF)]>
<[Rim to Sump Height(Uft|P3|RN|AP|Sn|OF)]>
<[Wall Thickness(Uin|P3|RN|AP|Sn|OF)]>
<[Floor Thickness(Uin|P3|RN|AP|Sn|OF)]>
<[Material(CP)]>
<[Frame(CP)]>
<[Grate(CP)]>
<[Cover(CP)]>
<[Frame Height(Uin|P3|RN|AP|Sn|OF)]>
<[Frame Diameter(Uin|P3|RN|AP|Sn|OF)]>
<[Frame Length(Uin|P3|RN|AP|Sn|OF)]>
<[Frame Width(Uin|P3|RN|AP|Sn|OF)]>
<[Barrel Height(Uft|P3|RN|AP|Sn|OF)]>

Valid property fields for AeccLabelStyleTextComponent.Contents in structures
<[Barrel Pipe Clearance(Uin|P3|RN|AP|Sn|OF)]>
<[Cone Height(Uin|P3|RN|AP|Sn|OF)]>
<[Slab Thickness(Uin|P3|RN|AP|Sn|OF)]>
<[Inner Structure Diameter(Uin|P3|RN|AP|Sn|OF)]>
<[Inner Structure Length(Uin|P3|RN|AP|Sn|OF)]>
<[Inner Structure Width(Uin|P3|RN|AP|Sn|OF)]>
<[Headwall Base Width(Uin|P3|RN|AP|Sn|OF)]>
<[Headwall Base Thickness(Uin|P3|RN|AP|Sn|OF)]>

Interference Checks

This section explains how to generate and examine an interference check. An interference check is used to determine when pipe network parts are either intersecting or are too close together.

Object Hierarchy



Pipe Network Interference Object Model

Performing an Interference Check

An interference check is used to detect intersections between the pipe parts of two different pipe networks or of pipes of a single network with themselves. The collection of all interference checks, an object of type `AeccInterferenceChecks`, is contained in the document's `AeccPipeDocument.InterferenceChecks` property. A new interference check is made using the `AeccInterferenceChecks.Create` method, which requires an `AeccInterferenceCheckCreationData` parameter. The creation data object holds all the information needed to perform the check, including the type of check to perform, the distance between parts required for an interference, and the pipe networks being checked. A new creation data object can only be made using the `AeccInterferenceChecks.GetDefaultCreationData` method. A valid check requires at least the `Name`, `LayerName`, `SourceNetwork` and `TargetNetwork` properties of the creation data object to be set.

The following sample performs an interference check between two networks:

```
' Get the collection of all interference checks.
Dim oInterferenceChecks As AeccInterferenceChecks
Set oInterferenceChecks = oPipeDocument.InterferenceChecks

' Set up the creation data structure for making an
' interference check.
Dim oCreationData As AeccInterferenceCheckCreationData
Set oCreationData =
oInterferenceChecks.GetDefaultCreationData

' If pipes are closer than 3.5 units apart, count it as an
' intersection.
oCreationData.Criteria.ApplyProximity = True
oCreationData.Criteria.CriteriaDistance = 3.5
oCreationData.Criteria.UseDistanceOrScaleFactor =
aeccDistance

' List the networks being tested. We will compare a network
' with itself, so we list it twice.
Set oCreationData.SourceNetwork = oPipeNetwork1
Set oCreationData.TargetNetwork = oPipeNetwork2

' Assign the check a unique name and a layer to use.
oCreationData.Name = "Test 01"
oCreationData.LayerName = oPipeDocument.Layers.Item(0).Name

' Create a new check of the pipe network.
Dim oInterferenceCheck As AeccInterferenceCheck
Set oInterferenceCheck = _
  oInterferenceChecks.Create(oCreationData)
```

Listing the Interferences

An interference check, the `AeccInterferenceCheck` object returned by the `AeccInterferenceChecks.Create` method, contains a collection of `AeccInterference` objects each representing a single interference found during the check. Each interference holds the point location of the interference center in the `Location` property, a three element array of doubles representing X, Y, and Z coordinates. The bounds of the entire interference area are returned by the `GetExtents` method. The extents are a two-item array of points, together representing the greatest and least corners of a cube containing the intersection area. The `SourceNetworkPart` and `TargetNetworkPart` properties hold the network parts that intersect.

```
Dim oInterference As AeccInterference
For Each oInterference In oInterferenceCheck
    ' Display the 2D x,y location of the interference.
    Dim vLocation As Variant
    Dim sLocation As String
    Dim vExtent As Variant
    vLocation = oInterference.Location
    sLocation = vLocation(0) & ", " & vLocation(1)
    MsgBox "There is an interference at:" & sLocation

    ' Display the greatest and least corners of the 3D
    ' rectangle containing the interference.
    vExtent = oInterference.GetExtents()
    Debug.Print "The interference takes place between:"
    sLocation = vExtent(0)(0) & ", "
    sLocation = sLocation & vExtent(0)(1) & ", "
    sLocation = sLocation & vExtent(0)(2)
    Debug.Print "  "; sLocation; "   and:"
    sLocation = vExtent(1)(0) & ", "
    sLocation = sLocation & vExtent(1)(1) & ", "
    sLocation = sLocation & vExtent(1)(2)
    Debug.Print "  "; sLocation
Next

If (oInterferenceCheck.Count = 0) Then
    MsgBox "There are no interferences in the network."
End If
```

Interference Check Styles

Either a symbol or a model of the actual intersection region can be drawn at each interference location. The display of these intersections is controlled by an `AeccInterferenceStyle` object. The collection of all interference style objects in the document are stored in the `AeccPipeDocument.InterferenceStyles` collection. Set the style of an interference object by assigning an `AeccInterferenceStyle` object to the `AeccInterference.Style` property:

```
Set oInterference.Style = oInterferenceStyle
```

There are three different styles of interference displays you can chose from. First, you can display a 3D model of the intersection region. This is done by setting the `ModelOptions` style property to `aeccTrueSolidInterference`. The `ModelSolidDisplayStyle2D` property, an object of type `AeccDisplayStyle`, controls the visible appearance of the model such as color and line types. Make sure the `ModelSolidDisplayStyle2D.Visible` property is set to `True`.

Another possibility is to draw a 3D sphere at the location of intersection. This is done by setting the `ModelOptions` style property to `aeccSphereInterference`. If the `InterferenceSizeType` property is set to `aeccSolidExtents`, then the sphere is automatically sized to just circumscribe the region of intersection (that is, it is the smallest sphere that still fits the model of the intersection region). You can set the size of the sphere by setting the `InterferenceSizeType` property to `aeccUserDefined`, setting the `ModelSizeOptions` property to use either absolute units or drawing units, and setting the corresponding `AbsoluteModelSize` or `DrawingScaleModelSize` property to the desired value. Again, the `ModelSolidDisplayStyle2D` property controls the visual features such as color and line type. Make sure the `ModelSolidDisplayStyle2D.Visible` property is set to `True`.

The third option is to place a symbol at the location of intersection. Set the `PlanSymbolDisplayStyle2D.Visible` style property to `True` to make symbols visible. The style property `MarkerStyle`, an object of type `AeccMarkerStyle`, controls all aspects of how the symbol is drawn.

This sample creates a new interference style object that displays an X symbol with a superimposed circle at points of intersection:

```
' Create a new interference style object.
Dim oInterferenceStyle As AeccInterferenceStyle
Set oInterferenceStyle = oPipeDocument.InterferenceStyles
 _
  .Add("Interference style 01")

' Draw a symbol of a violet X with circle with a specified
' drawing size at the points of intersection.
oInterferenceStyle.PlanSymbolDisplayStyle2D.Visible = True
With oInterferenceStyle.MarkerStyle
    .MarkerType = aeccUseCustomMarker
    .CustomMarkerStyle = aeccCustomMarkerX
    .CustomMarkerSuperimposeStyle = _
      aeccCustomMarkerSuperimposeCircle
    .MarkerDisplayStylePlan.color = 200 ' violet
    .MarkerDisplayStylePlan.Visible = True
    .MarkerSizeType = aeccAbsoluteUnits
    .MarkerSize = 5.5
End With
' Hide any model display at intersection points.
oInterferenceStyle.ModelSolidDisplayStyle2D.Visible = False
```

Sample Program

PipeSample.dvb

<installation-directory>\Sample\Civil 3D API\Vba\Pipe\PipeSample.dvb

The sample code from this chapter can be found in context in the PipeSample.dvb project. This sample creates a simple pipe network, creates and applies a new style, and performs an interference check.

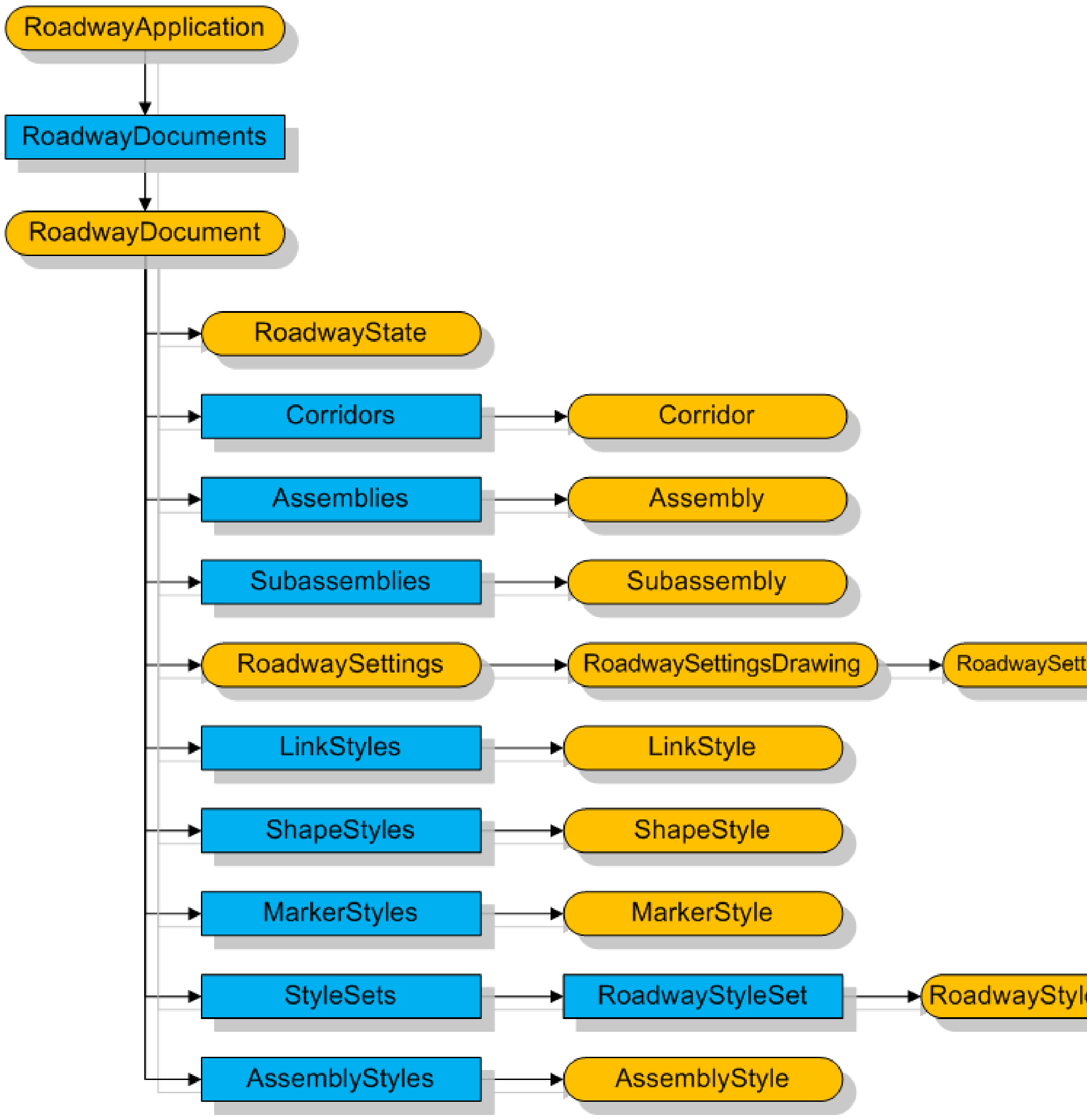
Corridors in COM

This chapter covers creating and managing corridor objects using the COM API. For information about performing these tasks using the .NET API, see Corridors in .NET (page 108).

Root Objects

This section explains how to get the base objects required for using the roadway API classes.

Object Hierarchy



Object Model for Root Corridor Objects

Accessing Corridor-Specific Base Objects

Applications that access corridors require special versions of the base objects representing the application and document. The `AeccRoadwayApplication` object is identical to the `AeccApplication` it is inherited from except that its `AeccRoadwayApplication.ActiveDocument` property returns an object of type `AeccRoadwayDocument` instead of `AeccDocument`. The `AeccRoadwayDocument` object contains collections of road related items, such as corridors, subassemblies, and style objects in addition to all of the methods and properties of `AeccDocument`.

When using corridor root objects, be sure to reference the “Autodesk Civil Engineering Corridor 6.0 Object Library” (AeccXRoadway.tlb) and “Autodesk Civil Engineering UI Corridor 6.0 Object Library” (AeccXUIRoadway.tlb) libraries.

This sample demonstrates how to retrieve the corridor root objects:

```
Dim oApp As AcadApplication
Set oApp = ThisDrawing.Application
Dim sAppName As String
sAppName = "AeccXUiRoadway.AeccRoadwayApplication"
Dim oRoadwayApplication As AeccRoadwayApplication
Set oRoadwayApplication = oApp.GetInterfaceObject(sAppName)

' Get a reference to the currently active document.
Dim oRoadwayDocument As AeccRoadwayDocument
Set oRoadwayDocument = oRoadwayApplication.ActiveDocument
```

Ambient Settings

Ambient settings allow you to get and set the unit and default property settings of roadway objects. Ambient settings for a corridor document are held in the `AeccRoadwayDocument.Settings` property, an object of type `AeccRoadwaySettingsRoot`. `AeccRoadwaySettingsRoot` inheirits all the properties of the `AeccSettingsRoot` class from which it is derived.

The roadway-specific properies of `AeccRoadwaySettingsRoot` let you adjust the settings for corridors, assemblies, subassemblies, and quantity takeoffs:

Corridor Ambient Settings

The corridor ambient settings object allows you to set the default name templates and default styles for corridor-related objects. The name templates allow you to set how new corridors, corridor surfaces, profiles from feature lines, or alignments from feature lines are named. Each template can use elements from the following property fields:

Valid property fields for AeccSettingsCorridor.NameTemplate
<[Corridor First Assembly(CP)]>
<[Corridor First Baseline(CP)]>
<[Corridor First Profile(CP)]>
<[Next Counter(CP)]>

... for AeccSettingsCorridor.CorridorSurfaceNameTemplate
<[Corridor Name(CP)]>
<[Next Corridor Surface Counter(CP)]>

...for AeccSettingsCorridor.ProfileFromFeatureLineNameTemplate
<[Next Counter(CP)]>

... for AeccSettingsCorridor.AlignmentFromFeatureLineNameTemplate
<[Corridor Baseline Name(CP)]>
<[Corridor Feature Code(CP)]>
<[Corridor Name(CP)]>
<[Next Counter(CP)]>

... for AeccSettingsCorridor.AlignmentFromFeatureLineNameTemplate
<[Profile Type]>

This sample sets the corridor name template:

```
' Get the ambient settings root object.
Dim oRoadwaySettings  As AeccRoadwaySettingsRoot
Set oRoadwaySettings = oRoadwayDocument.Settings

' Set the template so that new corridors are named
"Corridor"
' followed by a unique number followed by the name of the
' corridor's first assembly in parenthesis.
oRoadwaySettings.CorridorSettings.NameTemplate = _
  "Corridor <[Next Counter(CP)]>(<[Corridor First
Assembly(CP)]>)"
```

Default styles are set through the `AeccSettingsCorridor.StyleSettings` property. The styles for corridor alignments, alignment labels, code sets, surfaces, feature lines, profiles, profile labels, and slope pattern are accessed through a series of string properties.

This sample sets the style of alignments in a corridor to the first alignment style in the document's collection of styles:

```
' Get a reference to the corridor settings object.
Dim oSettingsCorridor As AeccSettingsCorridor
Set oSettingsCorridor =
oRoadwayDocument.Settings.CorridorSettings

' Get the name of the first alignment style in the
collection.
Dim sName As String
sName = oRoadwayDocument.AlignmentStyles.Item(0).Name

' Assign the name to alignment style property.
oSettingsCorridor.StyleSettings.AlignmentStyle.Value =
sName
```

Assembly Ambient Settings

The assembly ambient settings object allows you to set the default name templates and default styles for assemblies. The name templates allow you to set how new assemblies, offset assemblies, and assembly groups are named. Each template can use elements from the following property fields:

Valid property fields for AeccSettingsAssembly.NameTemplate
<[Next Counter(CP)]>

... for AeccSettingsAssembly.OffsetNameTemplate
<[Corridor Name(CP)]>

...for AeccSettingsAssembly.GroupNameTemplate
<[Next Counter(CP)]>

Default styles are set through the `AeccSettingsCorridor.StyleSettings` property. The styles for assemblies and code sets are accessed through string properties.

Subassembly Ambient Settings

The subassembly ambient settings object allows you to set the default name templates and default styles for subassembly objects. The name templates allow you to set how subassemblies created from entities and subassemblies create from macros are named. Each template can use elements from the following property fields:

... for AeccSettingsSubassembly.CreateFromEntitiesNameTemplate
<[Macro Short Name(CP)]>
<[Next Counter(CP)]>
<[Subassembly Local Name(CP)]>

... for AeccSettingsSubassembly.CreateFromEntitiesNameTemplate
<[Subassembly Side]>

... for AeccSettingsSubassembly.CreateFromMacroNameTemplate
<[Macro Short Name(CP)]>
<[Next Counter(CP)]>
<[Subassembly Local Name(CP)]>
<[Subassembly Side]>

The name of the default code style set is accessed through the `AeccSettingsSubassembly.CodeSetStyle` string property.

Each of these settings properties also contain a standard `AmbientSettings` property of type `AeccSettingsAmbient` for setting the default units of measurement.

Corridors

Corridor Concepts

A corridor represents a path, such as a road, trail, railroad, or airport runway. The geometry of a corridor is defined by a horizontal alignment and a profile. Together, these form the baseline - the centerline of the 3D path of the corridor. Along the length of the baselines are a series of assemblies which define the cross-sectional shape of the alignment. Common points in each assembly are connected to form feature lines. Together the assemblies and feature lines form the 3D shape of a corridor. A corridor also has one or more surfaces which can be compared against an existing ground surface to determine the amount of cut or fill required.

Listing Corridors

The collection of all corridors in a document are held in the `AeccRoadwayDocument.Corridors` property.

The following sample displays the name and the largest possible triangle side of every corridor in a document:

```
Dim oCorridors As AeccCorridors
Set oCorridors = oRoadwayDocument.Corridors

Dim oCorridor As AeccCorridor
For Each oCorridor In oCorridors
    Debug.Print "Corridor: " & oCorridor.Name
    Debug.Print oCorridor.MaximumTriangleSideLength
Next
```

Creating Corridors

The corridors collection includes a `AeccCorridors.Add` method for creating new corridors. This method creates the corridor based on an existing alignment, profile, and assembly.

NOTE

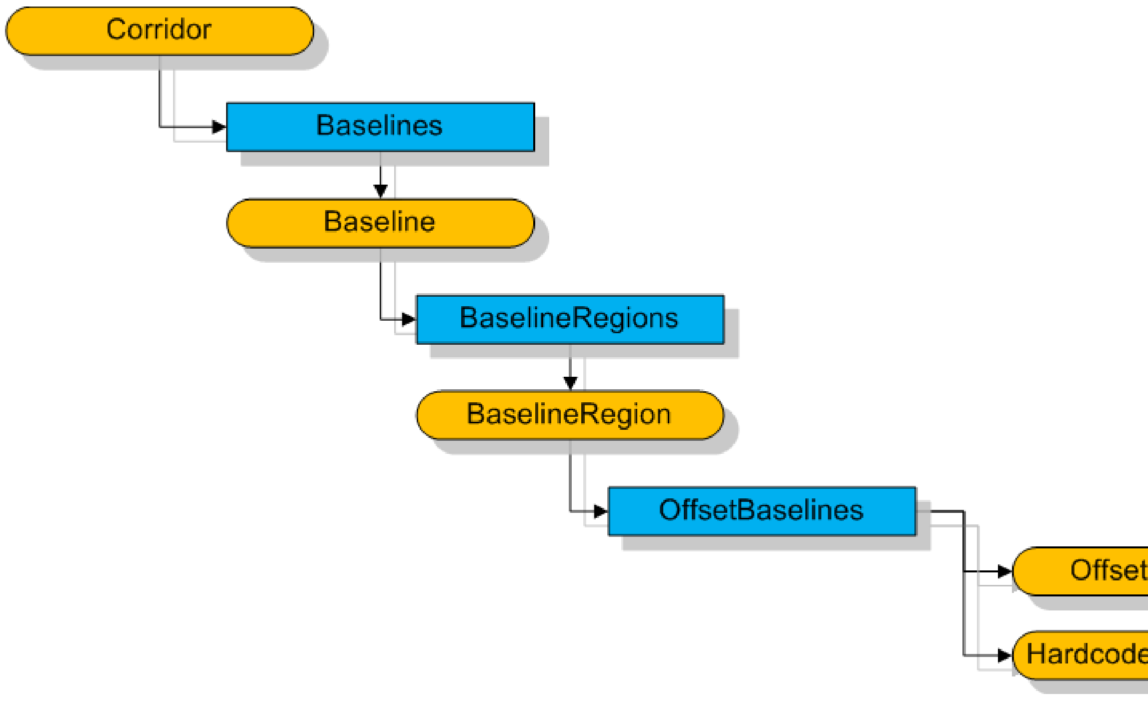
The station distance between assemblies cannot be set through the API, and needs to be set through the property page dialog box before the `AeccCorridors.Add` method is called.

```
' Assuming oAlignment, oProfile, and oAssembly represent
' valid AeccAlignment, AeccProfile, and AeccAssembly
objects.
Dim oCorridors As AeccCorridors
Set oCorridors = oRoadwayDocument.Corridors
Dim oCorridor As AeccCorridor
Set oCorridor = oCorridors.Add( _
  "Corridor01", _
  oAlignment.Name, _
  oProfile.Name, _
  oAssembly.Name)
```

Baselines

A baseline represents the centerline of the path of a corridor. It is based on an alignment (the horizontal component of the path) and a profile (the vertical component of the path). A corridor can contain more than one baseline if the corridor is modeling a complicated shape, such as an intersection. A baseline is made up of one or more baseline regions. Each region has its own assembly (its own cross section), so a corridor can have different shapes at different locations along its length.

Object Hierarchy



Baselines Object Model

Listing Baselines in a Corridor

The collection of all baselines in a corridor are contained in the `AeccCorridor.Baselines` property.

The following sample display information about the underlying alignment and profile for every baseline in a corridor:

```
Dim oBaseline As AeccBaseline
For Each oBaseline In oCorridor.Baselines
    Debug.Print "Baseline information -"
    Debug.Print "Alignment    : " & oBaseline.Alignment.Name
    Debug.Print "Profile      : " & oBaseline.Profile.Name
    Debug.Print "Start station: " & oBaseline.StartStation
    Debug.Print "End station  : " & oBaseline.EndStation
    Debug.Print
Next
```

Adding a Baseline to a Corridor

A baseline can be added to an existing corridor through the `AeccCorridor.AddBaseline` method. The baseline is defined by an existing alignment, profile, and assembly and consists of a single region.

NOTE

The station distance between assemblies cannot be set through the API, and needs to be set through the property page dialog box before this method is called.

The following sample adds a baseline using an existing alignment, profile, and assembly:

```
Set oBaseline = oCorridor.AddBaseline _
   (oAlignment.Name, oProfile.Name, oAssembly.Name)
```

Listing Baseline Regions

The collection of all the regions of a baseline are contained in the `AeccBaseline.BaselineRegions` property.

The AutoCAD Civil 3D API does not include methods for creating new baseline regions or manipulating existing regions.

The following sample displays the start and end station for every baseline region in a baseline:

```
Dim oBaselineRegion As AeccBaselineRegion
For Each oBaselineRegion In oBaseline.BaselineRegions
    Debug.Print "Baseline information -"
    Debug.Print "Start station: " &
oBaselineRegion.StartStation
   Debug.Print "End station: " & oBaselineRegion.EndStation
    Debug.Print
Next
```

Accessing and Modifying Baseline Stations

Assembly cross sections are placed at regular intervals along a baseline. The list of all stations where assemblies are located along a baseline can be retrieved using the `AeccBaseline.GetSortedStations` method while all stations along a baseline region can be retrieved using the `AeccBaselineRegion.GetSortedStations` method.

```
Dim v As Variant
v = oBaselineRegion.GetSortedStations()
Dim i As Integer
Debug.Print "Assembly stations:"
For i = 0 To UBound(v)
    Debug.Print v(i)
Next i
```

New stations can be added to baselines and baseline regions using the `AddStation` method. Existing stations can be deleted using the `DeleteStation` method. `DeleteStation` includes an optional `tolerance` parameter, letting you specify a station within a range. You can list all of the stations added to a baseline region with the `AeccBaselineRegion.GetAdditionalStation` method. `AeccBaselineRegion.ClearAdditionalStations` removes all added stations within a baseline region and leaves only the original stations created at regular intervals.

```
' Add an assembly to the baseline at station 12+34.5
oBaseline.AddStation 1234.5, "Station description"

' Remove the station located within 0.1 units around 5+67.5
oBaseline.DeleteStation 567.5, 0.1
```

Listing Offset Baselines

Within a baseline region, it is possible to have secondary baselines that are offset from the main baseline. The collection of these offset baselines are contained in the `AeccBaselineRegion.OffsetBaselines` property. The collection contains two kinds of baselines derived from the `IAeccBaseBaseline` interface. One is the hardcoded offset baseline (an instances of the `AeccHardcodedOffsetBaseline` class) which is a constant distance from the main baseline for the entire length of the offset baseline. The other is offset baseline (an instance of the `AeccOffsetBaseline` class), which is a variable distance from the main baseline.

NOTE

The AutoCAD Civil 3D API does not include methods for creating new offset baselines or hardcoded offset baselines.

This code examines each offset baseline within a baseline region:

```
Dim oBaseBaseline As IAeccBaseBaseline
For Each oBaseBaseline In oBaselineRegion.OffsetBaselines
   Dim dMainStart As Double ' station on main baseline
   Dim dMainEnd As Double ' station on main baseline
   Dim vOE As Variant

   Select Case oBaseline.Type
   Case aeccCorridorOffsetBaseline
      Dim oOffsetBaseline As AeccOffsetBaseline
      Set oOffsetBaseline = oBaseBaseline

      ' Report that an offset baseline exists.
      dMainStart =
oOffsetBaseline.StartStationOnMainBaseline
      dMainEnd = oOffsetBaseline.EndStationOnMainBaseline
      Debug.Print "Offset baseline, station " & dMainStart
 & _
         " to " & dMainEnd

      ' Report the offset of the baseline at its start and
 end.
      vOE = oOffsetBaseline. _
        GetOffsetElevationFromMainBaselineStation(dMainStart)
      Debug.Print " is offset by: " & _
         vOE(0) & " horizontal and: " & vOE(1) & _
         " vertical at start"
      vOE = oOffsetBaseline. _
        GetOffsetElevationFromMainBaselineStation(dMainEnd)
      Debug.Print " is offset by: " & vOE(0) & _
         " horizontal and: " & vOE(1) & " vertical at end"

   Case aeccCorridorHardcodedOffsetBaseline
      Dim oHardcodedOffsetBaseline As
AeccHardcodedOffsetBaseline
      Set oHardcodedOffsetBaseline = oBaseBaseline

      ' Report that a hardcoded offset baseline exists.
      dMainStart = oHardcodedOffsetBaseline.StartStation
      dMainEnd = oHardcodedOffsetBaseline.EndStation
      Debug.Print "Hardcoded offset baseline, station " _
         & dMainStart & " to " & dMainEnd
      vOE = oHardcodedOffsetBaseline. _
         OffsetElevationFromMainBaseline
      Debug.Print " is offset by: " & vOE(0) & _
         " horizontal and: " & vOE(1) & " vertical"
   End Select
Next
```

Assemblies and Subassemblies

An assembly is a pattern for the cross section of a corridor at a particular station. An assembly consists of a connected set of subassemblies, each of which are linked to a centerpoint or to other subassemblies. A subassembly consists of a series of shapes, links, and points. When an assembly is used to define the cross-section of a corridor, a series of applied assemblies (an object of type `AeccAppliedAssembly`) is added to the corridor. Each applied assembly consists of a collection of applied subassemblies, which in turn consist of shapes, links, and points that have been positioned relative to a specific station along the corridor baseline (`AeccCalcualtedShapes`, `AeccCalculatedLinks`, and `AeccCalculatedPoints` respectively). An applied assembly also has direct access to all the calculated shapes, links, and points of its constituent applied subassemblies.

NOTE

The AutoCAD Civil 3D API does not include methods for creating or modifying assemblies.

Object Hierarchy

Listing Applied Assemblies in a Baseline Region

The collection of all applied assemblies used in a baseline region are contained in the `AeccBaselineRegion.AppliedAssemblies` property.

The following sample displays information about the construction of an assembly for every assembly in a baseline region:

```
Dim oAppliedAssembly As AeccAppliedAssembly
For Each oAppliedAssembly In
oBaselineRegion.AppliedAssemblies
    Debug.Print "Applied Assembly"
    Dim lCount As Long
    lCount = oAppliedAssembly.GetShapes().Count
    Debug.Print "  Num Shapes: " & lCount
    Debug.Print
    lCount = oAppliedAssembly.GetLinks().Count
    Debug.Print "  Num Links: " & lCount
    lCount = oAppliedAssembly.GetPoints().Count
    Debug.Print "  Num Points: " & lCount
Next
```

An `AeccAppliedAssembly` object does not contain its baseline station position. Instead, each calculated point contains a method for determining its position with a baseline station, offset, and elevation called `AeccCalculatedPoint.GetStationOffsetElevationToBaseline`. Each calculated shape contains a collection of all links that form the shape, and each calculated link contains a collection of all points that define the link. Finally, each shape, link, and point contain an array of all corridor codes that apply to that element.

This sample retrieves all calculated point in an applied assembly and prints their locations:

```
Dim oPoint As AeccCalculatedPoint
For Each oPoint In oAppliedAssembly.GetPoints()
   Dim vPos As Variant
   vPos = oPoint.GetStationOffsetElevationToBaseline()
   Debug.Print "Position:  Station = " & vPos(0) & _
     " Offset = " & vPos(1) & " Elevation = " & vPos(2)
Next
```

Getting Applied Subassembly Information

An applied subassembly consists of a series of calculated shapes, links, and points, represented by objects of type `AeccCalculatedShape`, `AeccCalculatedLink`, and `AeccCalculatedPoint` respectivly.

```
Dim S, O, E As Double
With oAppliedSubassembly
    S = .OriginStationOffsetElevationToBaseline(0)
    O = .OriginStationOffsetElevationToBaseline(1)
    E = .OriginStationOffsetElevationToBaseline(2)
End With
Debug.Print "Station to baseline   : " & S
Debug.Print "Offset to baseline    : " & O
Debug.Print "Elevation to baseline : " & E
```

Applied subassemblies also contain a reference to the archetype subassembly (of type `AeccSubassembly`) in the subassembly database.

```
' Get information about the subassembly template.
Dim oSubassembly As AeccSubassembly
Set oSubassembly = oAppliedSubassembly.SubassemblyDbEntity
Debug.Print "Subassembly name: " & oSubassembly.Name
```

Feature Lines

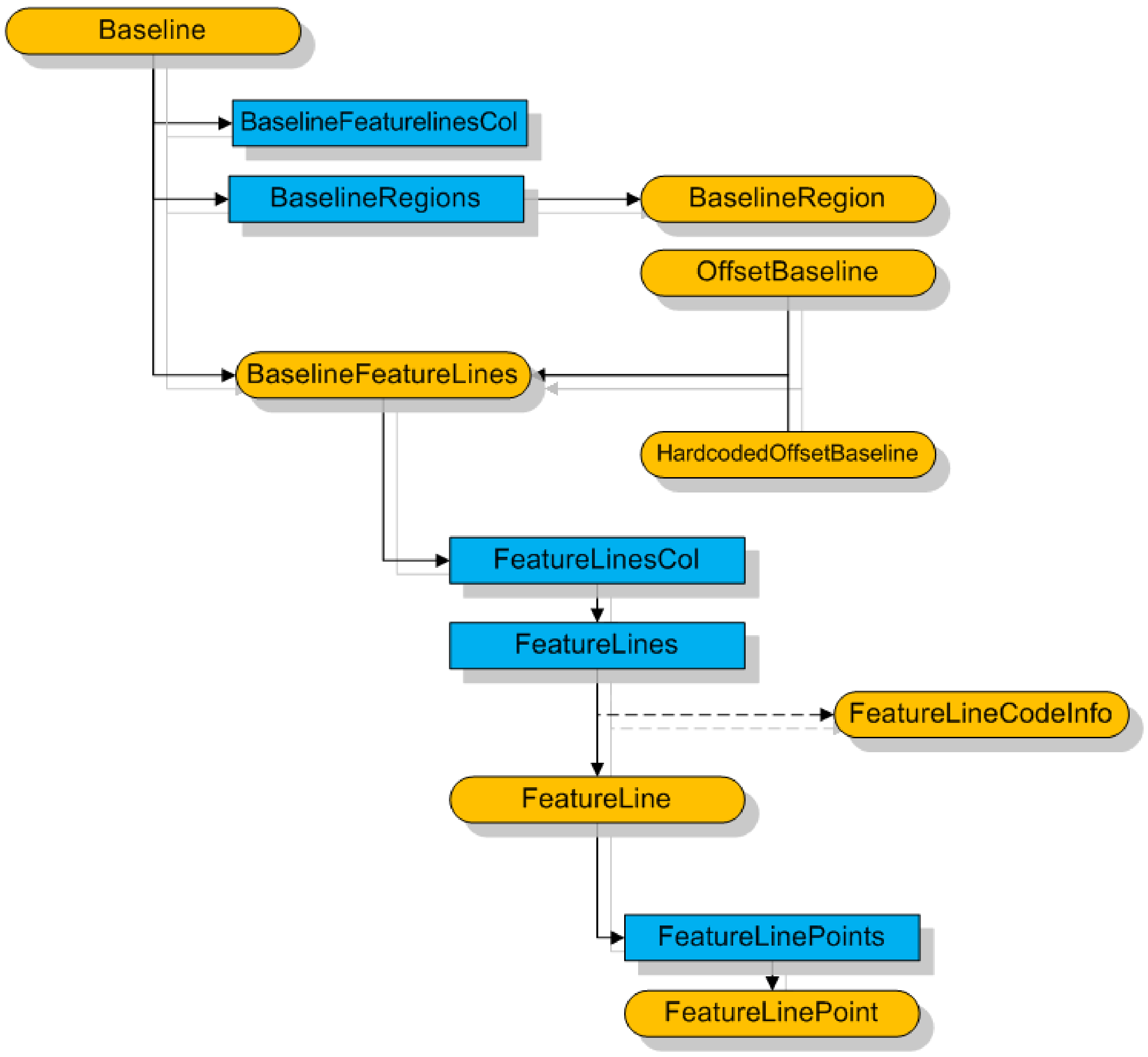
Feature lines are formed by connecting related points in each assembly along the length of a corridor baseline. These lines represent some aspect of the roadway, such as a sidewalk edge or one side of a corridor surface. Points become related by sharing a common *code*, a string property usually describing the corridor feature.

Each baseline has two sets of feature lines, one for lines that are positioned along the main baseline and one for lines that are positioned along any of the offset baselines.

NOTE

You can create feature lines from polylines using the `IAeccLandFeatureLine::AddFromPolyline()` method.

Object Hierarchy



Feature Line Object Model

Listing Feature Lines Along a Baseline

The set of all feature lines along a main baseline are held in the `AeccBaseline.MainBaselineFeatureLines` property, an object of type `AeccBaselineFeatureLines`. This object contains information about all the feature lines, such as a list of all codes used. Its `AeccBaselineFeatureLines.FeatureLinesCol` property is a collection of collections of feature lines. Each feature line (an object of type `AeccFeatureLine`) contains the code string used to create the feature line and a collection of all feature line points.

This sample lists all the feature line collections and feature lines along the main baseline. It also lists the code and every point location for each feature line.

```
Dim oBaselineFeatureLines As AeccBaselineFeatureLines
Set oBaselineFeatureLines =
oBaseline.MainBaselineFeatureLines

Dim oFeatureLinesCol As AeccFeatureLinesCol
Set oFeatureLinesCol = oBaselineFeatureLines.FeatureLinesCol
Debug.Print "# line collections:" & oFeatureLinesCol.Count

Dim oFeatureLines As AeccFeatureLines
For Each oFeatureLines In oFeatureLinesCol
   Debug.Print "Feature Line collection"
   Debug.Print "# lines in collection: " &
oFeatureLines.Count
   Dim oFeatureLine As AeccFeatureLine
   For Each oFeatureLine In oFeatureLines
      Debug.Print
      Debug.Print "Feature Line code: " &
oFeatureLine.CodeName

      ' Print out all point locations of the
      ' feature line.
      Dim oFeatureLinePoint As AeccFeatureLinePoint
      For Each oFeatureLinePoint In
oFeatureLine.FeatureLinePoints
         Dim X As Double
         Dim Y As Double
         Dim Z As Double
         X = oFeatureLinePoint.XYZ(0)
         Y = oFeatureLinePoint.XYZ(1)
         Z = oFeatureLinePoint.XYZ(2)
         Debug.Print "Point: " & X & ", " & Y & ", " & Z
      Next ' Points in a feature line
   Next ' Feature lines
Next ' Collections of feature lines
```

Listing Feature Lines Along Offset Baselines

As there can be many offset baselines in a single main baseline, the list of all feature lines along all offset baselines contains an extra layer. The `AeccBaseline.OffsetBaselineFeatureLinesCol` property contains a collection of `AeccBaselineFeatureLines` objects. These `AeccBaselineFeatureLines` objects not only contain the feature lines just as for the main baseline, but also contain properties identifying which offset baseline each group of feature lines belong to.

This sample shows how to modify the previous sample for feature lines along offset baselines:

```
Dim oBFeatureLinesCol As AeccBaselineFeatureLinesCol
Set oBFeatureLinesCol =
oBaseline.OffsetBaselineFeatureLinesCol

Dim oBaselineFeatureLines As AeccBaselineFeatureLines
' Loop through the groups of collections, one group for
each
' offset baseline.
For Each oBaselineFeatureLines In oBFeatureLinesCol

   Dim oFeatureLinesCol As AeccFeatureLinesCol
   Set oFeatureLinesCol =
oBaselineFeatureLines.FeatureLinesCol
   Debug.Print "# line collections:" &
oFeatureLinesCol.Count

   ' [...]
   ' This section is the same as the previous topic.

Next ' Groups of collections of feature lines
```

Each offset baseline and hardcoded offset baseline also has direct access to the feature lines related to itself. The `AeccBaselineFeatureLines` collection is accessed through the `RelatedOffsetBaselineFeatureLines` property in both types of offset baselines.

Corridor Surfaces

Corridor surfaces can represent the base upon which the corridor is constructed, the top of the finished roadway, or other aspects of the corridor. Such surfaces are represented by the `AeccSurface` class and by the unrelated `AeccCorridorSurface` class. `AeccCorridorSurface` objects contain corridor-specific information about the surfaces, such as which feature line, point, and link codes were used to create it.

Listing Corridor Surfaces

The collection of all corridor surfaces for each corridor is held in the the `AeccCorridor.CorridorSurfaces` property. Each corridor surface contains the boundary of the surface and a list of all point, link, and feature line codes used in the construction of the surface. Corridor surfaces also contain read-only references to the surface style and section style used in drawing the surface.

NOTE

The AutoCAD Civil 3D API does not include methods for creating new corridor surfaces or modifying existing corridor surfaces.

This sample lists all the corridor surfaces within a corridor and specifies which point codes were used:

```
Dim oCorridorSurface As AeccCorridorSurface
For Each oCorridorSurface In oCorridor.CorridorSurfaces
    Debug.Print "Surface name: "; oCorridorSurface.Name

    ' Get the point codes that were used to construct
    ' this surface.
    Dim sCodes() As String
    Dim sCodeList As String
    Dim i as Integer
    sCodes = oCorridorSurface.PointCodes
    For i = 0 To UBound(sCodes)
        sCodeList = sCodeList & " " & sCodes(i)
    Next i
    Debug.Print "Point codes: " & sCodeList
Next
```

Listing Surface Boundaries

Two different objects are used to define the limits of a corridor surface: boundaries and masks. A boundary is a polygon representing the outer edge of a surface or the inside edge of a hole in a surface. A mask is a polygon representing the part of the surface that can be displayed. The collection of all the boundaries of a surface are stored in the `AeccCorridorSurface.Boundaries` property and the collection of all masks are stored in the `AeccCorridorSurface.Masks` property.

Boundaries (of type `AeccCorridorSurfaceBoundary`) and masks (of type `AeccCorridorSurfaceMask`) are both derived from the same base interface (`IAeccCorridorSurfaceBaseMask`) and both have the similar methods and properties. The array of points making up the border polygon is retrieved by calling the `GetPolygonPoints` method. If the border was originally defined by selecting segments of feature lines, the collection of all such feature line components are contained in the `FeatureLineComponents` property.

NOTE

The AutoCAD Civil 3D API does not include methods for creating or modifying corridor boundaries or masks.

This sample loops through all the boundaries of a corridor surface and displays information about each:

```
Dim oCSBoundary As AeccCorridorSurfaceBoundary

For Each oCSBoundary In oCorridorSurface.Boundaries
   ' Get the type of boundary.
   Dim sBoundaryTitle As String
  If (oCSBoundary.Type = aeccCorridorSurfaceInsideBoundary)
 Then
      sBoundaryTitle = "  Inner Boundary: "
   Else
      sBoundaryTitle = "  Outer Boundary: "
```

```
    End If
    Debug.Print sBoundaryTitle & oCSBoundary.Name

    ' Get the points of the boundary polygon.
    Dim vPoints As Variant
    vPoints = oCSBoundary.GetPolygonPoints()
    Debug.Print " " & UBound(vPoints) & " points"
    ' Print the location of the first point. Usually
 corridors
    ' have a large number of boundary points, so we will
 not
    ' bother printing all of them.
    X = vPoints(1)(0)
    Y = vPoints(1)(1)
    Z = vPoints(1)(2)
    Debug.Print "Point 1: "; X; ", "; Y; ", "; Z

    ' Display information about each feature
    ' line component in this surface boundary.
    Debug.Print
    Debug.Print "Feature line components"
    Debug.Print " Count: ";
    Debug.Print oCSBoundary.FeatureLineComponents.Count

    Dim oFeatureLineComponent As AeccFeatureLineComponent
    For Each oFLineComponent In
 oCSBoundary.FeatureLineComponents
       Debug.Print "Code:" &
 oFLineComponent.FeatureLine.CodeName
       Debug.Print "Start station:" &
 oFLineComponent.StartStation
       Debug.Print "End station:" &
 oFLineComponent.EndStation
       Debug.Print: Debug.Print
    Next ' Feature line components
 Next ' Corridor surface boundaries
```

Computing Cut and Fill

One important use of corridor surfaces is to compare them against an existing ground surface to determine the amounts of cut and fill required to shape the terrain to match the proposed corridor. While `AeccCorridorSurface` objects cannot be used with `AeccSurface` objects directly to compute such statistics, each `AeccCorridorSurface` object also has a companion `AeccSurface` object of the same name.

This sample code demonstrates the creation of a volume surface from the difference between the existing ground and the datum surface of a corridor to determine cut, fill, and volume statistics:

```
' Get the collection of all surfaces in the drawing.
Dim oSurfaces As AeccSurfaces
Set oSurfaces = oRoadwayDocument.Surfaces

' Assign the setup information for the volume
' surface to be created.
Dim oTinVolumeCreationData As New AeccTinVolumeCreationData
oTinVolumeCreationData.Name = VOLUME_SURFACE_NAME
Dim sLayerName as String
sLayerName = oRoadwayDocument.Layers.Item(0).Name
oTinVolumeCreationData.BaseLayer = sLayerName
oTinVolumeCreationData.Layer = sLayerName
Set oTinVolumeCreationData.BaseSurface =
oSurfaces.Item("EG")
' Get the surface with the same name as the corridor
surface.
Set oTinVolumeCreationData.ComparisonSurface =
oSurfaces.Item(oCorridorSurface.Name)
oTinVolumeCreationData.Style =
oSurfaces.Item("EG").StyleName
oTinVolumeCreationData.Description = "Volume surface of
corridor"

' Create a volume surface that represents the
' difference between the two surfaces.
Dim oTinVolumeSurface As AeccTinVolumeSurface
Set oTinVolumeSurface =
oSurfaces.AddTinVolumeSurface(oTinVolumeCreationData)

' Get information about the volume surface and
' display it in a messagebox.
Dim dNetVol As Double
Dim dCutVol As Double
Dim dFillVol As Double
dNetVol = oTinVolumeSurface.Statistics.NetVolume
```

```
dCutVol = oTinVolumeSurface.Statistics.CutVolume
dFillVol = oTinVolumeSurface.Statistics.FillVolume
MsgBox "Net Volume = " & dNetVol & " cu.m" & _
  vbNewLine & "Cut = " & dCutVol & " cu.m" & _
  vbNewLine & "Fill = " & dFillVol & " cu.m", _
  vbOKOnly, _
  "Differences between """ & _
  oTinVolumeCreationData.BaseSurface.Name & _
  """ and """ & _
  oTinVolumeCreationData.ComparisonSurface.Name & _
  """"
```

Styles

These style objects control the visual appearance of applied assemblies.

Assembly Style

The collection of all assembly style objects are found in the `AeccRoadwayDocument.AssemblyStyles` property. The assembly style object contains properties for adjusting the marker types for the assembly attachment points, each of the standard `AeccMarkerType` property. While you can create new styles and edit existing styles, you cannot assign a style to an existing assembly using the AutoCAD Civil 3D API.

```
' Create a new assembly style and change it so that the
' place where the assembly attaches to the main baseline
' is marked with a red X.
Dim oAssemblyStyle As AeccAssemblyStyle
Set oAssemblyStyle =
oRoadwayDocument.AssemblyStyles.Add("Style1")
With oAssemblyStyle.MarkerStyleAtMainBaseline
     .CustomMarkerStyle = aeccCustomMarkerX
     .MarkerDisplayStylePlan.Color = 10 ' red
     .MarkerDisplayStylePlan.Visible = True
End With
```

Link Style

The collection of all link style objects are found in the `AeccRoadwayDocument.LinkStyles` property. This style object contains properties for adjusting the visual display of assembly and subassembly links.

NOTE

Link style objects are not used directly with link objects, but are instead used with roadway style sets.

```
' Create a new link style and color it green.
Dim oLinkStyle As AeccRoadwayLinkStyle
Set oLinkStyle = oRoadwayDocument.LinkStyles.Add("Style2")
With oLinkStyle
    .LinkDisplayStylePlan.color = 80
    .LinkDisplayStylePlan.Visible = True
End With
```

Shape Style

The collection of all shape style objects are found in the `AeccRoadwayDocument.ShapeStyles` property. This style object contains properties for adjusting the visual display of assembly and subassembly shapes, including the outline and the inside area.

NOTE

Shape style objects are not used directly with shape objects, but are instead used with roadway style sets.

```
' Create a new shape style and change it so that it has
' an orange border and a yellow hatch fill.
Dim oShapeStyle As AeccRoadwayShapeStyle
Set oShapeStyle = oRoadwayDocument.ShapeStyles.Add("Style3")
With oShapeStyle
    .AreaFillDisplayStylePlan.color = 50 ' yellow
    .AreaFillDisplayStylePlan.Visible = True
    .AreaFillHatchDisplayStylePlan.HatchType =
aeccHatchPreDefined
```

```
    .AreaFillHatchDisplayStylePlan.Pattern = "LINE"
    .BorderDisplayStylePlan.color = 30 ' orange
    .BorderDisplayStylePlan.Visible = True
End With
```

Roadway Style Sets

The visual display of applied assemblies is defined by roadway style sets, which are a set of shape styles and link styles assigned to shapes and links that use specified code strings. The collection of all style sets are found in the `AeccRoadwayDocument.StyleSets` property. A style set is itself a collection of `AeccRoadwayStyleSetItem` objects. Each style set item has a `AeccRoadwayStyleSetItem.CodeStyle` property that can reference either an existing shape style object or link shape object. New style set items are added to a style set though the `AeccRoadwayStyleSet.Add` method which takes parameters describing the kind of style object, the code string, and the style object itself. The particular style set in use is selected through the `AeccRoadwayStyleSet.InitAsCurrent` method.

```
' Create a new style set using our previously created
styles.
Dim oStyleSet As AeccRoadwayStyleSet
Set oStyleSet = oRoadwayDocument.StyleSets.Add("Style Set
 01")
Call oStyleSet.Add( _
   aeccLinkType, _
   "TOP", _
   g_oRoadwayDocument.LinkStyles.Item("Style2"))
Call oStyleSet.Add( _
   aeccShapeType, _
   "BASE", _
   oRoadwayDocument.ShapeStyles.Item("Style3"))

' Assign our new style set as the style set in current use.
oStyleSet.InitAsCurrent
```

Sample Program

CorridorSample.dvb

<installation-directory>\Sample\Civil 3D API\Vba\Corridor\CorridorSample.dvb

The `CreateCorridorExample` subroutine demonstrates the creation of a simple corridor using the `AeccCorridors.Add` method. Before calling this subroutine, be sure the current document contains at least one assembly. A suitable drawing is the file *Corridor-2b.dwg* located in the *<installation-directory>\Help\AutoCAD Civil 3D Tutorials\Drawings* directory.

The `GetCorridorInformationExample` subroutine extracts information of all existing corridors, baselines, feature lines, surfaces, assemblies, and subassemblies within the current document and displays the data in an instance of Word. A suitable drawing is the file *Corridor-4b.dwg* located in the *<installation-directory>\Help\Civil Tutorials\Drawings* directory.

NOTE

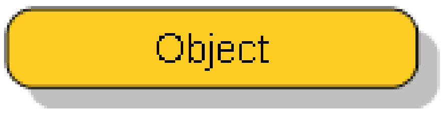
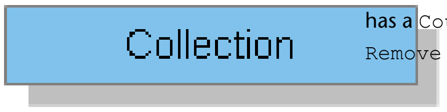
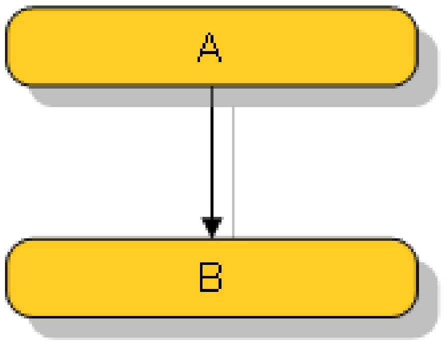
Microsoft Word must be running before starting this program.

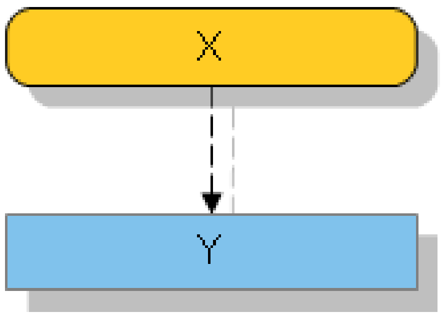
Object Hierarchy

AutoCAD Civil 3D

These images contain the hierarchy of all major objects. This is useful for determining which class instances are required to create an object of a particular type, or which objects can be accessed from an existing instance.

Legend

Graphic	Meaning
	a classname of “AeccOb-
	ı classname of “AeccCollec- ct is a list of other objects which can be enumerated. It also usually has a `Count` property and `Add`, `Item`, and `Remove` methods.
	tain object B from a property or object A.

Graphic	Meaning
	t X you can obtain a collection a subset of a larger collection.

Creating Client Applications

Overview

You can create stand-alone applications that use AutoCAD Civil 3D libraries to perform tasks. Sample programs written in C++, C#, and Visual Basic.NET are included in the *Samples* directory.

Samples

All of the following are located in the **<installation-directory>\Sample\Civil 3D API\COM** directory.

C++ Using COM

.\VC++\COM C++\ProjectStats.vcproj

Directly launches AutoCAD Civil 3D and creates a dialog box that displays some information about the current drawing or adds sample lines into the alignments of any selected sites.

Managed C++

.\VC++\Managed C++\C3DManagedExample.vcproj

Using COM interops, launches AutoCAD Civil 3D and creates a dialog box that displays some information about the current drawing or adds sample lines into the alignments of any selected sites.

C++ Using CustomDraw

.\VC++\CustomDraw\Sample\C3DCustomDraw.vcproj

Demonstrates accessing the CustomDraw API. This project overrides how triangles in TIN surfaces are drawn so that they're numbered. It requires the Autodesk ObjectARX libraries.

C++ Using Custom Events

.\VC++\CustomEvent\Sample\C3DCustomEvent.vcproj

Demonstrates using custom events. This project recieves notification just before and just after a corridor is rebuilt. It requires the Autodesk ObjectARX libraries.

C++ Using Custom UI

.\VC++\CustomEvent\Sample\C3DCustomUI.vcproj

Demonstrates UI customization. This project adds a button to the Properties Property sheet that opens a custom dialog for TIN surfaces. It requires the Autodesk ObjectARX libraries.

C++ Client Sample

.\VC++\VcClient\VcClientSamp.vcproj

Creates a dialog box that lets you launch AutoCAD Civil 3D and determine simple information about the current drawing.

C#

.\CSharp\CSharpClient\CSharpClientSample.csproj

Creates a dialog box that lets you launch AutoCAD Civil 3D and determine simple information about the current drawing.

Visual Basic .NET

.\VB_NET\VbDotNetClient\VBDotNetClientSample.vbproj

Creates a dialog box that lets you launch AutoCAD Civil 3D and determine simple information about the current drawing.

Index