

AutoCAD® Map 3D 2010

Geospatial Platform Developer's Guide

The Autodesk logo is displayed in white text on a black rectangular background. The word "Autodesk" is written in a bold, sans-serif font, oriented vertically from bottom to top.

April 2009

© 2009 Autodesk, Inc. All Rights Reserved. Except as otherwise permitted by Autodesk, Inc., this publication, or parts thereof, may not be reproduced in any form, by any method, for any purpose.

Certain materials included in this publication are reprinted with the permission of the copyright holder.

Trademarks

The following are registered trademarks or trademarks of Autodesk, Inc., in the USA and other countries: 3DEC (design/logo), 3December, 3December.com, 3ds Max, ADI, Alias, Alias (swirl design/logo), AliasStudio, AliasWavefront (design/logo), ATC, AUGI, AutoCAD, AutoCAD Learning Assistance, AutoCAD LT, AutoCAD Simulator, AutoCAD SQL Extension, AutoCAD SQL Interface, Autodesk, Autodesk Envision, Autodesk Insight, Autodesk Intent, Autodesk Inventor, Autodesk Map, Autodesk MapGuide, Autodesk Streamline, AutoLISP, AutoSnap, AutoSketch, AutoTrack, Backdraft, Built with ObjectARX (logo), Burn, Buzzsaw, CAiCE, Can You Imagine, Character Studio, Cinestream, Civil 3D, Cleaner, Cleaner Central, ClearScale, Colour Warper, Combustion, Communication Specification, Constructware, Content Explorer, Create>what's>Next> (design/logo), Dancing Baby (image), DesignCenter, Design Doctor, Designer's Toolkit, DesignKids, DesignProf, DesignServer, DesignStudio, DesignStudio (design/logo), Design Web Format, Discreet, DWF, DWG, DWG (logo), DWG Extreme, DWG TrueConvert, DWG TrueView, DXF, Ecotect, Exposure, Extending the Design Team, Face Robot, FBX, Filmbox, Fire, Flame, Flint, FMDesktop, Freewheel, Frost, GDX Driver, Gmax, Green Building Studio, Heads-up Design, Heidi, HumanIK, IDEA Server, i-drop, ImageModeler, iMOUT, Incinerator, Inferno, Inventor, Inventor LT, Kaydara, Kaydara (design/logo), Kynapse, Kynogon, LandXplorer, LocationLogic, Lustre, Matchmover, Maya, Mechanical Desktop, Moonbox, MotionBuilder, Movimento, Mudbox, NavisWorks, ObjectARX, ObjectDBX, Open Reality, Opticore, Opticore Opus, PolarSnap, PortfolioWall, Powered with Autodesk Technology, Productstream, ProjectPoint, ProMaterials, RasterDWG, Reactor, RealDWG, Real-time Roto, REALVIZ, Recognize, Render Queue, Retimer, Reveal, Revit, Showcase, ShowMotion, SketchBook, Smoke, Softimage, Softimage|XSI (design/logo), SteeringWheels, Stitcher, Stone, StudioTools, Topobase, Toxik, TrustedDWG, ViewCube, Visual, Visual Construction, Visual Drainage, Visual Landscape, Visual Survey, Visual Toolbox, Visual LISP, Voice Reality, Volo, Vtour, Wire, Wiretap, WiretapCentral, XSI, and XSI (design/logo).

The following are registered trademarks or trademarks of Autodesk Canada Co. in the USA and/or Canada and other countries: Backburner, Multi-Master Editing, River, and Sparks.

The following are registered trademarks or trademarks of MoldflowCorp. in the USA and/or other countries: Moldflow, MPA, MPA (design/logo), Moldflow Plastics Advisers, MPI, MPI (design/logo), Moldflow Plastics Insight, MPX, MPX (design/logo), Moldflow Plastics Xpert.

All other brand names, product names or trademarks belong to their respective holders.

Disclaimer

THIS PUBLICATION AND THE INFORMATION CONTAINED HEREIN IS MADE AVAILABLE BY AUTODESK, INC. "AS IS." AUTODESK, INC. DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE REGARDING THESE MATERIALS.

Published by:
Autodesk, Inc.
111 McInnis Parkway
San Rafael, CA 94903, USA

Contents

Chapter 1	Overview	1
	Introduction	1
	Relationship to MapGuide	1
	Geospatial Platform API Documentation	2
	Setting Up Visual Studio	3
	Sample Applications	3
Chapter 2	Resources	5
	Introduction	5
	Working With Resources	8
	Differences Between AutoCAD Map 3D and MapGuide	10
	Sample - Using Resources	11
	The XML Files	11
	Adding The Resource	13
	Points To Watch	15
Chapter 3	Feature Service	17
	Overview	17
	Defining Feature Sources	18
	Example: Defining a Vector Feature Source	19
	Example: Defining a Raster Feature Source	21
	Joins	23
	Adding Feature Classes to a Map	26

	Selecting Feature Data	28
	Representation of Geometry	29
	Selecting Using the API	30
	Basic Filters	31
	Examples	31
	Spatial Filters	32
	Creating Geometry Objects From Features	33
	Updating Features	34
	Edit Sets	35
	Creating SDF files	36
Chapter 4	Maps and Layers	41
	Overview	41
	Basic Layer Properties	41
	Layer Groups	42
	Layer Visibility	43
	Example: Actual Visibility	43
	Manipulating Layers	43
	Changing Visibility	44
	Layer Definition	44
	Layers With Joined Feature Sources	51
	Filtering Layers By Geometry	51
	Modifying Layer Style	52
Chapter 5	Geometry	57
	Overview	57
	Geometry Objects	57
	Comparing Geometry Objects	58
	Coordinate Systems	59
	Measuring Distance	60
	Creating a Buffer	61
Chapter 6	Coordinate System API	65
	The Earth Moves Under Our Feet	65
	Coordinate System Types	66
	Coordinate System Categories	68
	Projection Grouping and Characterization	68
	Projection Parameters	73
	Ellipsoid	83
	Datum	83
	Coordinate System	84
	Coordinate Transformation	91
	Quadrant	93
	Secant Projection	93

Caspian Sea Mercator Projection	94
Format Conversion	94
Definition Comparison	96
Index	97

Overview

1

Introduction

The AutoCAD Map 3D Geospatial Platform API is used for managing geospatial data in a map. It shares many classes and methods with Autodesk MapGuide® Enterprise and MapGuide Open Source, so applications written for one product can work in another with minimal modification.

The Geospatial Platform API is not the same as the AutoCAD Map 3D ObjectARX .NET API. For details about this API see the ObjectARX .NET Developer's Guide and the ObjectARX .NET API Reference.

Relationship to MapGuide

Many parts of the Geospatial Platform API are shared between AutoCAD Map 3D and MapGuide Web Server Extensions. There are differences in the products, though, that affect how the API is used.

The MapGuide API is designed to be used in a web server environment, and is available in PHP, Java, and .NET. AutoCAD Map 3D is designed to be used in a desktop environment and exposes only a .NET API.

AutoCAD Map 3D stores some resource information directly in the DWG file, while MapGuide uses an external repository. This is explained in more detail in [Resources](#) (page 5). The MapGuide repository is structured like a file system, with a hierarchy of folders. To ensure as much consistency between the products as possible, AutoCAD Map 3D uses a similar structure for its internal repository.

Some methods in the Geospatial Platform API are not valid in AutoCAD Map 3D, and will throw an exception if called. Generally these will be methods that do not have corresponding functionality in the AutoCAD Map 3D environment.

For example, AutoCAD Map 3D does not have the concept of permissions on resources so any methods dealing with permissions are invalid. These are identified in the API Reference.

All map data in a MapGuide application is written and read through FDO connections. This means that updates occur immediately. AutoCAD Map 3D works with data stored in the DWG file as well as FDO data. Depending on the method chosen, updates in a AutoCAD Map 3D application may be cached and not written directly to the feature source.

Differences in operation between AutoCAD Map 3D and MapGuide mean that in some cases the Geospatial Platform API has a different implementation in the products. AutoCAD Map 3D includes a set of classes that extend the API. For example, the Geospatial Platform API has an `MgLayerBase` class. AutoCAD Map 3D extends this in the `AcMapLayer` class.

All classes that are part of the Geospatial Platform API begin with the prefix `Mg`. Classes that extend the Geospatial Platform API for use in AutoCAD Map 3D begin with the prefix `AcMap`.

AutoCAD Map 3D includes the following enums that are not part of the Geospatial Platform API:

- `EditMode`
- `HighlightMode`

AutoCAD Map 3D also includes event handling, which is not a part of the Geospatial Platform API. For more details see the Geospatial Platform Supplement Reference.

Geospatial Platform API Documentation

The documentation for the Geospatial Platform API is in 3 places:

- AutoCAD Map 3D Geospatial Platform Developer's Guide (this guide)
- AutoCAD Map 3D Geospatial Platform API Reference
- AutoCAD Map 3D .NET Reference Supplement

The Geospatial Platform API reference contains documentation about classes and methods that are common to AutoCAD Map 3D and MapGuide. When a method behaves differently in AutoCAD Map 3D this is noted in the reference.

The Geospatial Platform Supplement Reference contains documentation about classes and methods that are used only within AutoCAD Map 3D. In most cases the classes extend classes in the Geospatial Platform API Reference.

The Geospatial Platform API works with the Feature Data Objects (FDO) API. FDO documentation is available with AutoCAD Map 3D, both in the product installation help folder and in the SDK.

NOTE The API References are available in CHM format. The Developer's Guides are available in PDF and CHM format. To view all CHM documentation as part of one help system, open *sdk.doc.main.chm*.

Setting Up Visual Studio

To use the Geospatial Platform API, follow the instructions in the *AutoCAD Map 3D ObjectARX .NET Developer's Guide*. Add the following reference to the project:

`AcMapApiMgd.dll`

This assembly contains the Geospatial Platform API and adds the namespace `OSgeo.MapGuide`. It is located in the AutoCAD Map 3D installation folder.

Sample Applications

The SDK includes sample applications in the *MapSamples\Platform* folder. See the *Developer Samples Guide* for details. The samples serve as a good introduction to many aspects of the Geospatial Platform API and can be used as a starting point for developing custom applications.

Resources

2

Introduction

In the Geospatial Platform API, *resources* are the files and configuration information necessary to draw layers and maps. There are various types of resources required. For example, a `FeatureSource` describes the location, type, and other details needed for access to GIS feature data. A `LayerDefinition` defines the data and style for a layer.

Resources are stored in a *resource repository*. AutoCAD Map 3D has a single repository named `Library`. This repository is contained within the DWG file.

MapGuide uses multiple repositories. The `Library` repository contains persistent, site-wide data. There is a single `Library` repository for each MapGuide site. There are multiple `Session` repositories, each one containing data from a single MapGuide session. `Session` repositories are unique to an individual session and cannot be shared. All MapGuide repositories are managed on the site server.

Allowable resource types are defined as static members of the class `MgResourceType`. Some resource types apply only to MapGuide.

The following resource types are valid for both Map 3D and MapGuide:

Resource Type	Description
<code>FeatureSource</code>	Contains the required parameters for connecting to a geospatial feature source.
<code>LayerDefinition</code>	Contains the required parameters for displaying and styling a layer. Layers can be drawing layers, vector layers, or grid (raster) layers.

Resource Type	Description
SymbolDefinition	Defines a symbol to be displayed on a map.

The following resource types are only valid for MapGuide:

Resource Type	Description
ApplicationDefinition	Defines a flexible Web layout for the Fusion framework.
DrawingSource	Contains the required parameters for connecting to a DWG file.
Folder	A folder in the resource repository.
LoadProcedure	Contains the required parameters for loading new data into the MapGuide repository.
Map	Contains the run-time definition of a map.
MapDefinition	Defines an initial map state, used as the basis for creating a run-time map.
PrintLayout	Defines the components of a printed map created from MapGuide.
Selection	Contains selection information.
SymbolLibrary	Defines a library of symbols.
WebLayout	Defines the components of a Web layout for a MapGuide Viewer.

Resource repositories are structured like directories, with folders, subfolders, and documents. Each resource is an XML document in the repository, named with a unique resource identifier. A resource identifier is made up of the following parts:

- Repository type—either “Library” or “Session”. Map 3D only uses “Library”.
- Repository name—for library repositories, an empty string. For session repositories, a unique session identifier assigned by the site server.
- Path—the path to the folder containing the resource.
- Name—the resource name, without the extension.

- Resource type—the resource type (the extension). This must match one of the allowable types defined in `MgResourceType`.

For example, the following could be a name for a feature source in either Map 3D or MapGuide:

```
Library://Samples/CityOutline.FeatureSource
```

The following could be a name for a map definition in MapGuide:

```
Session:70ea89fe-0000-1000-8000-005056c00008_en//Map.MapDefinition
```

Resource Service

Resources are managed by the resource service, an `MgResourceService` object.

In AutoCAD Map 3D, use `AcMapServiceFactory` to get the resource service:

```
MgResourceService resourceService =  
    AcMapServiceFactory.GetService(MgServiceType.ResourceService)  
    as MgResourceService;
```

In MapGuide, use `MgSiteConnection` to get the resource service:

```
MgSiteConnection siteConnection = new MgSiteConnection();  
siteConnection.Open(userInfo);  
MgResourceService resourceService =  
    siteConnection.CreateService(MgServiceType.ResourceService)  
    as MgResourceService;
```

Resource Dependencies

Resources may depend on other items:

- Some resources are self sufficient and do not refer to any other resources or files.
- Some resources reference other resources. For example, layer definitions and feature sources are stored as separate resources. A layer definition resource contains a reference to the resources for the feature sources that are used in that layer.
- Some resources use associated *resource data*. For example, this is used to store configuration information for ODBC/WMS/Raster feature sources.

Resource Schemas

Resources are stored in the repository as XML documents. Applications that add or modify resources must ensure that the XML validates against the appropriate schema.

For AutoCAD Map 3D, the schemas are available as part of the SDK, in the *Schema* folder.

For MapGuide, the schemas are installed with the server files, in *InstallDir\Server\Schema*.

The schemas are documented in the XML Schemas module of the *Geospatial Platform Reference*.

Working With Resources

Resources in the repository can be managed by using the *MgByteSource*, *MgByteReader*, and *MgByteSink* objects.

To convert a string representation of the XML data into data for a resource, first create an *MgByteSource*. New *MgByteSource* objects can be initialized with a string or the contents of a file. Get an *MgByteReader* from the byte source and pass that to *MgResourceService.SetResource()*.

For example, if an external file contains the string representation of the XML for a layer definition, the following would read the contents of the file and store it in the repository.

```
MgResourceIdentifier rID_layer = new MgResourceIdentifier(
    @"Library://Data/Zoning.LayerDefinition");
MgByteSource layer_byteSource = new MgByteSource(xmlFilePath);
layer_byteSource.SetMimeType("text/xml");
resourceService.SetResource(rID_layer,
    layer_byteSource.GetReader(), null);
```

There are many ways of modifying the XML in a resource. The best method to use in a given situation depends on the development environment, the developer's familiarity with a given technique, and the desired result. Some possibilities are described in the following sections. Other techniques for working with XML data are equally valid.

xsd.exe

For .NET development, used with AutoCAD Map 3D and the ASP.NET API of MapGuide, it is possible to use the tool *xsd.exe*, which is available with the

SDK that comes with Microsoft Visual Studio. `xsd.exe` reads an XML schema and generates .NET classes for working with XML documents based on the schema. `xsd.exe` can generate both VB.NET and C# classes.

To use `xsd`, a typical command is:

```
xsd.exe LayerDefinition-1.0.0.xsd /c /l:cs /n:OSGeo.Map
Guide.Schema.LayerDefinition
```

For an example of how to use these classes, see the *Developer Samples Guide*.

For more information on `xsd`, see <http://msdn.microsoft.com>.

layerdefinitionfactory.php

MapGuide includes *layerdefinitionfactory.php*, which contains PHP methods for working with the *LayerDefinition* schema. These methods are designed to handle most basic tasks with layer definitions, but do not provide complete support for all possibilities. They create the XML by substituting strings into external templates. It is relatively easy to adapt the techniques from *layerdefinitionfactory.php* for use with a different schema or to translate them into one of the other development languages.

Autodesk MapGuide Studio

One of the simplest ways to create XML templates is to use Autodesk MapGuide Studio with Autodesk MapGuide Enterprise or MapGuide Open Source. Using Studio, create the resources with the desired values, then save them as XML. The resulting file can be edited using a text editor, XML editor, or with techniques like those in *layerdefinitionfactory.php*.

Document Object Model

Finally, all of the development languages support versions of the Document Object Model (DOM) API, which is a general-purpose API for working with and validating XML files. Using the DOM API requires converting data into a string, then converting that string into an XML document. For example, the following takes resource data from the repository and loads it into an XML document:

```
// using System.Xml;

MgByteReader byteReader =
    resourceService.GetResourceContent(resourceId);
XmlDocument doc = new XmlDocument();
doc.LoadXml(byteReader.ToString());
```

Modify the XML document using standard DOM methods, then convert it back to a string and re-save the resource.

```
String xmlString = doc.DocumentElement.OuterXml;
resourceService.SetResource(resourceId,
    new MgByteReader(xmlString, "text/xml"), null);
```

Differences Between AutoCAD Map 3D and MapGuide

The Platform API has been made to be as consistent as possible between AutoCAD Map 3D and MapGuide. However, there are some differences, especially in the way they handle resources.

- AutoCAD Map 3D does not use ApplicationDefinition, DrawingSource, Folder, LoadProcedure, Map, MapDefinition, PrintLayout, Selection, SymbolLibrary, or WebLayout resources.
- In AutoCAD Map 3D, the resources are stored inside the DWG drawing file or a DWT template drawing, not in an external database. This means that resources and their paths are specific to that drawing. In other words, the same repository path in another drawing may not be defined, or may refer to a different resource. If you need to share resources between drawings, you must add them to each drawing.
- Session repositories are not supported in AutoCAD Map 3D. All resources must be stored in the `Library` repository.
- AutoCAD Map 3D does not use resource headers. For any method that allows them, for example `SetResource()`, enter `null` for that parameter.
- The following methods in `MgResourceService` are not supported in AutoCAD Map 3D:
 - `ApplyResourcePackage()`
 - `ChangeResourceOwner()`
 - `GetRepositoryContent()`
 - `GetRepositoryHeader()`
 - `GetResourceHeader()`
 - `UpdateRepository()`

- The method of specifying the resource data for a file differs.
In MapGuide, the procedure is:
 - 1 In the feature source XML, provide the name of the SDF file. For example:


```
<Value>%MG_DATA_FILE_PATH%
HydrographicPolygons.sdf</Value>
```
 - 2 Use `ResourceService.SetResourceData()` to specify the location of the resource data (the SDF file) on disk.

In AutoCAD Map 3D, the procedure is to specify the absolute location of the resource data in the feature source XML. For example:

```
<Parameter>
  <Name>File</Name>
  <Value>C:\Map ObjectARX SDK\Map Samples\Platform\BuildMap\Data\SDF\Buffered.sdf</Value>
</Parameter>
```

No call to `ResourceService.SetResourceData()` is required.

AutoCAD Map 3D *does* use `ResourceService.SetResourceData()` for streams, but only for the configuration information for ODBC/WMS/Raster feature sources.

Sample - Using Resources

This example shows how to add a resource programmatically, using external files that contain the XML for the resources.

The XML Files

This sample uses data from the BuildMap developer sample. When you run BuildMap, it writes these files to the directory containing `BuildMap.dll`. See the *Developer Samples Guide* for details.

One file is *Zoning.layer*. This specifies all the properties for drawing one layer in a map. Note the `<ResourceId>` element, which refers to the resource for the feature source.

```

<?xml version="1.0" encoding="utf-8"?>
<LayerDefinition
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  version="1.0.0">
  <VectorLayerDefinition>
    <ResourceId>Library://Data/SDF/Zoning.FeatureSource
    </ResourceId>
    <FeatureName>Schema:Zoning</FeatureName>
    <FeatureNameType>FeatureClass</FeatureNameType>
    <Geometry>Geometry</Geometry>
    <VectorScaleRange>
      <AreaTypeStyle>
        <AreaRule>
          <LegendLabel />
          <AreaSymbolization2D>
            <Fill>
              <FillPattern>Solid</FillPattern>
              <ForegroundColor>a03cafda</ForegroundColor>
              <BackgroundColor>FF000000</BackgroundColor>
            </Fill>
            <Stroke>
              <LineStyle>Solid</LineStyle>
              <Thickness>0.0</Thickness>
              <Color>FF000000</Color>
              <Unit>Centimeters</Unit>
            </Stroke>
          </AreaSymbolization2D>
        </AreaRule>
      </AreaTypeStyle>
    </VectorScaleRange>
  </VectorLayerDefinition>
</LayerDefinition>

```

The second file is *Zoning.FeatureSource*, which defines the properties of an SDF feature source. The File parameter points to the location on disk. See [Feature Service](#) (page 17) for more details.

```

<?xml version="1.0" encoding="utf-8"?>
<FeatureSource xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Provider>OSGeo.SDF.3.3</Provider>
  <Parameter>
    <Name>File</Name>
    <Value>C:\Map 3D SDK\Map Samples\Platform\BuildMap\Data\SDF\Zoning.sdf</Value>
  </Parameter>
  <Parameter>
    <Name>ReadOnly</Name>
    <Value>False</Value>
  </Parameter>
</FeatureSource>

```

Adding The Resource

To use the XML files to create the layer and the feature source, use `MgResourceService.SetResource()`. For example:

```

// Add the layer definition.
// It will be stored in the repository using the resource
// identifier.
MgResourceIdentifier rID_layer = new MgResourceIdentifier(
    @"Library://Data/Zoning.LayerDefinition");

// Note: Modify this directory to point to the location of
// the XML files.

String xmlDirectory = @"C:\Map 3D SDK\
    + @"Map Samples\Platform\BuildMap\bin\Debug\";

// Read the XML file, then save its contents in the resource
// repository.

MgByteSource layer_byteSource =
    new MgByteSource(xmlDirectory + "Zoning.layer");
layer_byteSource.SetMimeType("text/xml");
resourceService.SetResource(rID_layer,
    layer_byteSource.GetReader(), null);

// Add the feature source
// Note: The layer definition XML refers to
// 'Library://Data/SDF/Zoning.FeatureSource' so
// we must use that name here.
MgResourceIdentifier rID_feature_source =
    new MgResourceIdentifier(
        @"Library://Data/SDF/Zoning.FeatureSource");
MgByteSource feature_byteSource = new
    MgByteSource(xmlDirectory + "Zoning.FeatureSource");
feature_byteSource.SetMimeType("text/xml");
resourceService.SetResource(rID_feature_source,
    feature_byteSource.GetReader(), null);

```

Then add the layer to the current map. For example:

```

MgLayerBase layer = AcMapLayer.Create(rID_layer,
    resourceService);
layer.SetName("NewLayer");
AcMapMap currentMap = AcMapMap.GetCurrentMap();
currentMap.GetLayers().Add(layer);

```

Points To Watch

- In the feature source XML, the provider name (for example, `OSGeo.SDF.3.3`) must match one of the provider names listed in the output of `MgFeatureService.GetFeatureProviders()`.

- In the layer definition, the `ResourceId` value must match the path of the feature source. In the example above, this path is used in the layer definition:

```
<ResourceId>Library://Data/SDF/Zoning.FeatureSource
</ResourceId>
```

So the corresponding path must be used when the feature source is created.
For example:

```
MgResourceIdentifier rID_feature_source =
    new MgResourceIdentifier(
        @"Library://Data/SDF/Zoning.FeatureSource");
```

- Create new layers using the static method `AcMapLayer.Create()`.
- Create new layer groups using the `AcMapLayerGroup` constructor `AcMapLayerGroup(groupName)`.

Feature Service

3

Overview

A feature source represents a single FDO (Feature Data Objects) connection. FDO is an API for reading and writing geospatial data in a variety of formats. For more details about FDO, see the documentation included with AutoCAD Map 3D and the SDK, or visit *fdo.osgeo.org*.

Feature Service provides a common API for reading and writing feature data from data sources for which an FDO provider exists.

Different feature sources have different capabilities. For example, an Oracle Spatial database will have more capabilities than an ODBC connection. Some feature sources can have multiple schemas.

One schema in a feature source can describe multiple feature classes. For example, a single SDF file could contain a feature class for roads and another feature class for hydrography.

In AutoCAD Map 3D, a single FDO feature class corresponds to a single map layer. There cannot be more than one feature class per layer, and a layer containing feature data cannot contain AcDb entities.

NOTE All Feature Service operations work with features belonging to the current map. To get the current map, call `AcMapMap.GetCurrentMap()`.

Defining Feature Sources

To define a feature source within AutoCAD Map 3D, create a resource identifier for the feature source and store it in the resource repository. Then call `MgResourceService.SetResource()` with three parameters:

- An `MgResourceIdentifier` that contains the path in the resource repository for the resource.
- An `MgByteReader` that provides the XML describing the feature source. This conforms to `FeatureSource.xsd`.
- A null parameter for the resource header. This is not used by AutoCAD Map 3D.

The repository path is of the form

```
Library://resourcePath/resourceName.FeatureSource
```

`resourcePath` is optional, but `FeatureSource` (case sensitive) is mandatory. For example, the following creates an `MgResourceIdentifier`:

```
MgResourceIdentifier parcelId =  
    new MgResourceIdentifier("Library://parcels.FeatureSource");
```

The exact form of the XML describing the feature source varies depending on the FDO provider. See the schema documentation for `FeatureSource.xsd` and the *Connection API* section of *The Essential FDO* for details. Different FDO providers accept different parameters, as described in *The Essential FDO*. For example, the `Autodesk.Raster.3.2` provider accepts a single parameter, `DefaultRasterFileLocation`. This is set in a `Name/Value` parameter in the XML data.

```
<?xml version="1.0" encoding="utf-8"?>  
<FeatureSource  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">  
  <Provider>Autodesk.Raster.3.2</Provider>  
  <Parameter>  
    <Name>DefaultRasterFileLocation</Name>  
    <Value>C:\some\folder\rasterfile.jpg</Value>  
  </Parameter>  
  <ConfigurationDocument>config://rasterfile.jpg</ConfigurationDocument>  
  <LongTransaction />  
</FeatureSource>
```


Some providers require additional configuration. For example, the `Autodesk.Raster.3.2` provider requires georeferencing information to define the location and resolution of the raster data for some raster file formats. This can be in a world file in the same folder as the raster file, or it can be in a separate FDO configuration file. WMS and ODBC providers can also require separate configuration files.

For those providers that do require a configuration file, the `FeatureSource` XML contains a `<ConfigurationDocument>` element that points to resource data. Call `MgResourceService.SetResourceData()` to set the data. The data name must match the name in the `<ConfigurationDocument>` element. The `BuildMap` sample uses `config://` as part of the resource data name, but this is not required.

Example: Defining a Vector Feature Source

Defining a vector feature source requires creating the feature source definition and storing it in the resource repository using the feature source resource identifier. The following example assumes that `SDFpath` is a string containing the file path of an SDF file. It creates a feature source with a resource identifier of `Library://feature.FeatureSource`.

There are many ways to create a feature source definition. One method, used in the sample applications and the following example, uses classes generated using the Visual Studio tool `xsd.exe` on `FeatureSource.xsd`. `FeatureSourceType` and `NameValuePairType` are classes generated this way.

NOTE Resource service requires data to be UTF-8 encoded.

```

// Get the services

MgResourceService rs;
rs = AcMapServiceFactory.GetService(MgServiceType.ResourceService)
    as MgResourceService;
MgFeatureService fs;
fs = AcMapServiceFactory.GetService(MgServiceType.FeatureService)
    as MgFeatureService;
MgResourceIdentifier fsId = new MgResourceIdentifier(
    "Library://feature.FeatureSource");

// Create the feature source definition with a required
// parameter of File and an optional parameter of ReadOnly

string xmlString;
FeatureSourceType fsType = new FeatureSourceType();

fsType.Provider = "OSGeo.SDF.3.2";

NameValuePairType param = new NameValuePairType();
param.Name = "File";
param.Value = SDFpath;
NameValuePairType param2 = new NameValuePairType();
param2.Name = "ReadOnly";
param2.Value = "false";
fsType.Parameter = new NameValuePairType[] { param, param2 };

// Serialize the feature source object model to xml string
using (StringWriter writer = new StringWriter())
{
    XmlSerializer xs = new XmlSerializer(fsType.GetType());
    xs.Serialize(writer, fsType);
    xmlString = writer.ToString();
}

// Convert the Unicode string to UTF8 bytes for Resource Service

byte[] unicodeBytes = Encoding.Unicode.GetBytes(xmlString);
byte[] utf8Bytes = Encoding.Convert(Encoding.Unicode,
    Encoding.UTF8, unicodeBytes);

// Create a byte reader containing the XML feature
// source definition. Store the definition in the repository

```

```
MgByteSource xmlSource = new MgByteSource(utf8Bytes,
    utf8Bytes.Length);
rs.SetResource(fsId, xmlSource.GetReader(), null);
```

Example: Defining a Raster Feature Source

Defining a raster feature source is similar to defining a vector feature source. The parameters vary depending on the provider. Most raster files do not contain coordinate system information, so it must be obtained some other way. This can be done using a world file or through additional configuration for the FDO provider. The following example uses the second method, adding resource data containing the configuration information.

The resource identifier contains the location in the resource repository for the feature source definition.

```
MgResourceIdentifier rasterId = new MgResourceIdentifier(
    "Library://rasterFeature.FeatureSource");
```

```

string xmlString;
FeatureSourceType fsType = new FeatureSourceType();

fsType.Provider = "Autodesk.Raster.3.2";

NameValuePairType param = new NameValuePairType();
param.Name = "DefaultRasterFileLocation";
param.Value = @"C:\some\folder\rasterfile.jpg";
fsType.Parameter = new NameValuePairType[] { param };

fsType.ConfigurationDocument = "config://rasterfile.jpg";
fsType.LongTransaction = "";

// Serialize the feature source object model to an xml string
using (StringWriter writer = new StringWriter())
{
    XmlSerializer xs = new XmlSerializer(fsType.GetType());
    xs.Serialize(writer, fsType);
    xmlString = writer.ToString();
}

byte[] unicodeBytes = Encoding.Unicode.GetBytes(xmlString);
byte[] utf8Bytes = Encoding.Convert(Encoding.Unicode,
    Encoding.UTF8, unicodeBytes);
MgByteSource xmlSource =
    new MgByteSource(utf8Bytes, utf8Bytes.Length);

MgResourceService rs;
rs = AcMapServiceFactory.GetService(MgServiceType.ResourceService)
    as MgResourceService;
rs.SetResource(rasterId, xmlSource.GetReader(), null);

```

The configuration information varies depending on the FDO provider. The WMS, ODBC, and Raster providers may need additional configuration information. Refer to the FDO documentation in the Map SDK and in the AutoCAD Map 3D installation folder for more details.

The following reads the configuration from a file and stores it in the resource repository. Ensure that the data name matches the `<ConfigurationDocument>` element in the feature source definition.

```

MgByteSource source;
source = new MgByteSource(@"C:\some\folder\rasterconfig.xml");
rs.SetResourceData(rasterId, "config://rasterfile.jpg",
    MgResourceDataType.Stream, source.GetReader());

```

Joins

Joins extend a feature source by combining it with data from another feature source, similar to a database join. They are commonly used to combine GIS data with data from a database.

For example, an SDF file containing the following properties:

- parcel geometry
- parcel ID

could be joined with an ODBC database containing the following properties:

- parcel ID
- owner name
- assessment value
- land use designation

The join is defined in the feature source definition. Each join requires 2 feature classes. The primary feature class contains the geometry for display in AutoCAD Map 3D. The secondary feature class does not need geometry, but it must have a property that matches a property in the primary feature class. In the example above, the SDF file contains the primary feature class and the ODBC database contains the secondary feature class.

In most cases the feature classes will be from different feature sources, but it is possible to create a self-referencing join within 1 feature class or a join between 2 feature classes in the same feature source.

To create a joined feature source, begin with the secondary feature source. This does not require anything special. For example, the following is a simple connection to an ODBC database:

```

<FeatureSource
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xsi:noNamespaceSchemaLocation="MapFeatureSource-1.0.0.xsd"
  version="1.0.0">
  <Provider>OSGeo.ODBC.3.2</Provider>
  <Parameter>
    <Name>Password</Name>
    <Value></Value>
  </Parameter>
  <Parameter>
    <Name>GenerateDefaultGeometryProperty</Name>
    <Value>>false</Value>
  </Parameter>
  <Parameter>
    <Name>ConnectionString</Name>
    <Value></Value>
  </Parameter>
  <Parameter>
    <Name>DataSourceName</Name>
    <Value>OWNERS</Value>
  </Parameter>
  <Parameter>
    <Name>UserId</Name>
    <Value></Value>
  </Parameter>
  <ConfigurationDocument></ConfigurationDocument>
  <LongTransaction></LongTransaction>
</FeatureSource>

```

The primary feature source defines the join. It must identify the secondary feature source and which properties are to be used for the join. For example, the following is a feature source definition for an SDF file with a join to an ODBC database:

```

<FeatureSource
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xsi:noNamespaceSchemaLocation="MapFeatureSource-1.0.0.xsd"
  version="1.0.0">
  <Provider>OSGeo.SDF.3.2</Provider>
  <Parameter>
    <Name>ReadOnly</Name>
    <Value>false</Value>
  </Parameter>
  <Parameter>
    <Name>File</Name>
    <Value>C:\some\folder\Parcels.sdf</Value>
  </Parameter>
  <ConfigurationDocument></ConfigurationDocument>
  <LongTransaction></LongTransaction>
  <Extension>
    <Name>Parcels_Joins1</Name>
    <FeatureClass>SHP_Schema:Parcels</FeatureClass>
    <AttributeRelate>
      <AttributeClass>Fdo:Parcels</AttributeClass>
      <ResourceId>Library://ODBC_1</ResourceId>
      <Name>Parcels</Name>
      <AttributeNameDelimiter>|</AttributeNameDelimiter>
      <RelateType>LeftOuter</RelateType>
      <ForceOneToOne>true</ForceOneToOne>
      <RelateProperty>
        <FeatureClassProperty>APN</FeatureClassProperty>
        <AttributeClassProperty>APN</AttributeClassProperty>
      </RelateProperty>
    </AttributeRelate>
  </Extension>
</FeatureSource>

```

The `<Extension>` element defines the join. `<Name>` is the name of the join, used in a layer definition that references the feature source. If a feature source defines more than one join they must have different names. `<FeatureClass>` is a feature class in the primary feature source.

`<AttributeRelate>` defines a join to a single feature source. It is possible for the primary feature source to have joins to more than one secondary feature source. This is represented by multiple `<AttributeRelate>` elements. As an example, the owner information could be in one database table and assessment information could be in another.

For each join, `<AttributeClass>` defines the schema and feature class in the secondary feature source. `<ResourceId>` is the resource identifier. `<RelateType>` is the type of join. This release of AutoCAD Map 3D supports left outer joins and inner joins.

The join can have multiple `<RelateProperty>` elements. These define the fields that must match for the join.

Store the feature source definition using `MgResourceService.SetResource()`, as described in [Defining Feature Sources](#) (page 18).

Adding Feature Classes to a Map

To add a feature class to a map, create a layer definition for the new layer and add it to the resource repository. Create an `AcMapLayer` and set its layer definition. Add the layer to the map using

```
AcMapMap.GetCurrentMap().GetLayers().Add().
```

Creating the layer definition is similar to creating the feature source definition. See [Defining Feature Sources](#) (page 18). The layer definition uses the `LayerDefinition.xsd` schema, which includes styling information. See [Layer Definition](#) (page 44) for details.

The `<FeatureName>` element in the layer definition is of the form

```
schemaName:className
```

where *schemaName* and *className* are valid for the feature source. Call `MgFeatureService.GetSchemas()` to get the schema names for a feature source and `MgFeatureService.GetClasses()` to get the class names for a schema.

The following example creates a layer definition for a raster layer and adds the layer to the current map.


```

AcMapMap currentMap = AcMapMap.GetCurrentMap();
string layerDefName = "Library://rasterlayer.LayerDefinition";
MgResourceIdentifier layerId = new
    MgResourceIdentifier(layerDefName);

// Use classes from xsd.exe to build the layer definition

LayerDefinitionType layerDef = new LayerDefinitionType();
GridLayerDefinitionType gridLayerDef = new
    GridLayerDefinitionType();
layerDef.Item = gridLayerDef;
gridLayerDef.ResourceId = rasterId.ToString();
gridLayerDef.FeatureName = "rasters:classname";
gridLayerDef.Geometry = "Image";

GridScaleRangeType[] ranges = new GridScaleRangeType[1];
gridLayerDef.GridScaleRange = ranges;

ranges[0] = new GridScaleRangeType();
ranges[0].ColorStyle = new GridColorStylizationType();

GridColorRuleType[] colorRules = new GridColorRuleType[1];
ranges[0].ColorStyle.ColorRule = colorRules;

colorRules[0] = new GridColorRuleType();
colorRules[0].LegendLabel = "";
colorRules[0].Color = new GridColorType();
colorRules[0].Color.ItemElementName = ItemChoiceType.Band;
colorRules[0].Color.Item = "1";
ranges[0].RebuildFactor = 1;

// Serialize the layer definition to XML

string layerDefString;
using (StringWriter writer = new StringWriter())
{
    XmlSerializer xs = new XmlSerializer(layerDef.GetType());
    xs.Serialize(writer, layerDef);
    layerDefString = writer.ToString();
}

// Convert Unicode to UTF-8

```

```

unicodeBytes = Encoding.Unicode.GetBytes(layerDefString);
utf8Bytes = Encoding.Convert(Encoding.Unicode, Encoding.UTF8,
    unicodeBytes);

// Create a byte reader containing the layer source definition

xmlSource = new MgByteSource(utf8Bytes, utf8Bytes.Length);
rs.SetResource(layerId, xmlSource.GetReader(), null);

// Add the layer to the Map
MgLayerBase layer = new AcMapLayer(layerId, rs);
layer.Name = "newLayer";
currentMap.GetLayers().Add(layer);

```

This adds the new layer to the end of the layer collection, at the top of the draw order.

Layers can be organized into groups. To add a layer to a group call `AcMapLayer.SetGroup()` with the group name.

Selecting Feature Data

Individual features within a feature source are identified by unique feature IDs. The form of the ID depends on the feature source. Call `MgFeatureService.GetClassDefinition()` to get the feature class definition. Call `MgClassDefinition.GetIdentityProperties()` to get a list of all properties that are identity properties.

To get the currently selected features for a map, call `AcMapMap.GetFeatureSelection()`. This returns an `MgSelectionBase` object. A single selection can contain features from multiple layers and classes. Call `MgSelectionBase.GetLayers()` to get all layers containing selected objects, or select an individual layer.

For example, to find selected features on the `Parcels` layer,

```

AcMapMap currentMap = AcMapMap.GetCurrentMap();
MgFeatureService fs =
    AcMapServiceFactory.GetService(MgServiceType.FeatureService)
    as MgFeatureService;

MgLayerCollection layers = currentMap.GetLayers();
MgLayerBase parcelsLayer = layers.GetItem("Parcels");
string fcName = parcelsLayer.GetFeatureClassName();

MgSelectionBase selection = currentMap.GetFeatureSelection();
string selectionFilter = selection.GenerateFilter(parcelsLayer,
    fcName );

MgFeatureQueryOptions queryOpts = new MgFeatureQueryOptions();
queryOpts.SetFilter(selectionFilter);
MgFeatureReader featureReader;

MgResourceIdentifier fsId = new
    MgResourceIdentifier(parcelsLayer.GetFeatureSourceId());
featureReader = fs.SelectFeatures(fsId, fcName, queryOpts);

```

A selection consists of one or more feature IDs. To process the features in a selection, create an `MgFeatureReader` that contains the selected features, then call `MgFeatureReader.ReadNext()` to advance through the feature reader. To create the `MgFeatureReader()` for the selected features in a layer, call `MgSelectionBase.GenerateFilter()`, which creates a filter to select the features. Then create and initialize an `MgFeatureQueryOptions` object. Finally, call `AcMapFeatureService.SelectFeatures()`.

An `MgFeatureReader` iterates through a list of features from a single feature source. To advance to the next feature in the reader, call `MgFeatureReader.ReadNext()`.

Representation of Geometry

AutoCAD Map 3D can represent geometric data in 3 different forms:

- AGF text format, which is an extension of the Open Geospatial Consortium (OGC) Well Known Text (WKT) format. This is used to represent geometry as a character string.
- Binary AGF format. The is used by the FDO technology supporting the Feature Service.

- Internal representation, using `MgGeometry` and classes derived from it.

To convert between AGF text and the internal representation, use an `MgWktReaderWriter` object. Call `MgWktReaderWriter.Read()` to convert AGF text to `MgGeometry`. Call `MgWktReaderWriter.Write()` to convert `MgGeometry` to AGF text.

To convert between binary AGF and the internal representation, use an `MgAgfReaderWriter` object. Call `MgAgfReaderWriter.Read()` to convert binary AGF to `MgGeometry`. Call `MgAgfReaderWriter.Write()` to convert `MgGeometry` to binary AGF.

For example, if you have a WKT representation of the geometry, you could create a geometry object as follows:

```
MgWktReaderWriter wktReaderWriter = new MgWktReaderWriter();  
MgGeometry geometry = wktReaderWriter.Read(wktGeometry);
```

Selecting Using the API

Selections can be created programmatically with the Platform API. This is done by querying data in a feature source, creating a feature reader that contains the features, then converting the feature reader to a selection (`MgSelection` object).

To create a feature reader, apply a selection filter to a feature class in the feature source. A selection filter can be a *basic filter*, a *spatial filter*, or a combination of the two. The filter is stored in an `MgFeatureQueryOptions` object.

Basic filters are used to select features based on the values of feature properties. For example, you could use a basic filter to select all roads that have four or more lanes.

Spatial filters are used to select features based on their geometry. For example, you could use a spatial filter to select all roads that intersect a certain area.

Basic Filters

Basic filters perform logical tests of feature properties. You can construct complex queries by combining expressions. Expressions use the comparison operators below:

Operator	Meaning
=	Equality
<>	Not equal
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
LIKE	Used for string comparisons. The "%" wildcard represents any sequence of 0 or more characters. The "_" wildcard represents any single character. For example, "LIKE Schmitt%" will search for any names beginning with "Schmitt".

The comparison operators can be used with numeric or string properties, except for the `LIKE` operator, which can only be used with string properties.

Combine or modify expressions with the standard boolean operators `AND`, `OR`, and `NOT`.

Examples

These examples assume that the feature class you are querying has an integer property named `year` and a string property named `owner`. To select all features newer than 2001, create a filter like this:

```
MgFeatureQueryOptions queryOptions = new MgFeatureQueryOptions();
queryOptions.SetFilter("year > 2001");
```

To select all features built between 2001 and 2004, create a filter like this:

```
queryOptions.SetFilter("year >= 2001 and year <= 2004");
```

To select all features owned by Davis or Davies, create a filter like this:

```
queryOptions.SetFilter("owner LIKE 'Davi%'");
```

Spatial Filters

With spatial filters, you can do comparisons using geometric properties. For example, you can select all features that are inside an area on the map, or that intersect an area.

There are two ways of using spatial filters:

- Create a separate spatial filter to apply to the feature source, using the `MgFeatureQueryOptions.SetSpatialFilter()` method.
- Include spatial properties in a basic filter created with the `MgFeatureQueryOptions.SetFilter()` method.

The `MgFeatureQueryOptions.SetSpatialFilter()` method requires an `MgGeometry` object to define the geometry and a spatial operation to compare the feature property and the geometry. The spatial operations are defined in class `MgFeatureSpatialOperations`.

To include spatial properties in a basic filter, define the geometry using WKT format. Use the `GEOMFROMTEXT()` function in the basic filter, along with one of the following spatial operations:

- CONTAINS
- COVEREDBY
- CROSSES
- DISJOINT
- EQUALS
- INTERSECTS
- OVERLAPS
- TOUCHES
- WITHIN
- INSIDE

NOTE The spatial operations are not case sensitive, so “CONTAINS” and “contains” produce the same result.

For example, the following selects features that cross an area:

```
queryOptions.SetFilter(  
    "SHPGEOM crosses GEOMFROMTEXT(" + wktGeom + ")" );
```

The same filtering syntax can be used in a layer definition, to create a layer containing only those features that pass the filter. See [Filtering Layers By Geometry](#) (page 51) for details.

NOTE Not all spatial operations can be used on all features. It depends on the capabilities of the FDO provider that supplies the data. This restriction applies to separate spatial filters and spatial properties that are used in a basic filter.

Creating Geometry Objects From Features

You may want to use an existing feature as part of a spatial query. To retrieve the feature’s geometry and convert it into an appropriate format for a query, perform the following steps:

- Create a query that will select the feature.
- Query the feature class containing the feature using
`AcMapLayer.SelectFeatures()` OR `MgFeatureService.SelectFeatures()`.
- Obtain the feature from the query using the `MgFeatureReader.ReadNext()` method.
- Get the geometry data from the feature using the
`MgFeatureReader.GetGeometry()` method. This data is in AGF binary format.
- Convert the AGF data to an `MgGeometry` object using the
`MgAgfReaderWriter.Read()` method.

For example, the following sequence creates an `MgGeometry` object representing the boundaries of zone 1 in the layer named “zones”. It creates an `MgGeometry` object and the WKT representation of that object.

```

MgLayerCollection layers = currentMap.GetLayers();
MgLayerBase layer = layers.GetItem("zones");
string fsId = layer.GetFeatureSourceId();
string className = layer.GetFeatureClassName();
MgFeatureQueryOptions query =
    new MgFeatureQueryOptions();
query.SetFilter("ZONE_ID = 1");
MgResourceIdentifier resId =
    new MgResourceIdentifier(fsId);

MgFeatureReader featureReader =
    fs.SelectFeatures(resId,
        className, query);
if (featureReader.ReadNext())
{
    string geometryName = layer.GetFeatureGeometryName();
    MgByteReader geometryData =
        featureReader.GetGeometry(geometryName);
    MgAgfReaderWriter agfReaderWriter = new MgAgfReaderWriter();
    MgGeometry geometry = agfReaderWriter.Read(geometryData);

    MgWktReaderWriter wktReaderWriter = new MgWktReaderWriter();
    string wkt = wktReaderWriter.Write(geometry);
}

```

The following assumes that another feature class has a geometry property SHPGEOM. It uses the WKT string in a query to find features in the other feature class that intersect the zone:

```

MgFeatureQueryOptions queryOpts = new MgFeatureQueryOptions();
queryOpts.SetFilter("SHPGEOM intersects GEOMFROMTEXT(" +
    wkt + ")");

```

Updating Features

All updates to features in a feature source are performed using an `MgFeatureCommandCollection`. The feature command collection can contain commands to add new features, delete existing features, update existing features, lock features, or unlock features.

To add a new feature, create an `MgInsertFeatures` object. `MgInsertFeatures` has two constructors. One has an `MgPropertyCollection` parameter and is used to insert a single feature. The other has an `MgBatchPropertyCollection` parameter and is used to insert multiple features in a single command.

The `MgPropertyCollection` contains values for the feature properties, including the feature geometry property. Create an empty `MgPropertyCollection`, then add properties using `MgPropertyCollection.Add()`. The different property types, such as `MgGeometryProperty` or `MgInt32Property`, are all derived from the base class `MgProperty`.

Most feature classes will require a geometry property, which is used to display the feature. To create a geometry property, start with an `MgGeometry` object.

Convert the `MgGeometry` to binary AGF format suitable for FDO.

```
MgByteReader geometryStream;  
MgAgfReaderWriter agfReaderWriter = new MgAgfReaderWriter;  
geometryStream = agfReaderWriter.Write(geometry);
```

Create a property collection to contain the feature properties. Add the geometry property. Add any other required properties. The property names are case sensitive and must match the names defined in the feature class.

```
MgPropertyCollection properties = new MgPropertyCollection();  
MgGeometryProperty geomProp;  
geomProp = new MgGeometryProperty("SHPGEOM", geometryStream);  
properties.Add(geomProp);
```

Create a feature command to insert the feature, and add it to a feature command collection.

```
MgFeatureCommandCollection commands =  
    new MgFeatureCommandCollection();  
MgInsertFeatures insertCommand;  
insertCommand = new MgInsertFeatures(className, properties);  
commands.Add(insertCommand);
```

Call `AcMapLayer.UpdateFeatures()` to add the new feature.

```
mapLayer.UpdateFeatures(commands);
```

Deleting and updating features is similar. The constructors for `MgDeleteFeatures` and `MgUpdateFeatures` require different parameters, but the feature commands can be added to the same `MgFeatureCommandCollection`.

Edit Sets

There are two different modes for updates to features:

- Edit set

■ Direct update

NOTE Some FDO providers, including the file-based providers SDF, SHP, and raster, do not support edit set mode.

Updates made using edit set mode are made to the working copy inside the AutoCAD Map 3D application, but are not committed to the feature source. Edits made using direct update are changed in the feature source immediately.

There are two methods for updating features:

■ `AcMapLayer.UpdateFeatures()`

■ `MgFeatureService.UpdateFeatures()`

`AcMapLayer.UpdateFeatures()` works directly with features in the map layer. This is the preferred method when working in edit set mode.

`MgFeatureService.UpdateFeatures()` updates the features in the feature source. This method can be used when the feature source is not being displayed in a layer. For example, an AutoCAD Map 3D application could create an SDF file containing result data without ever having to display the data.

To set the update mode for a layer, call `AcMapLayer.SetEditSetMode()`.

To commit changes made in edit set mode, call `AcMapLayer.SaveFeatureChanges()`. This updates the feature source. To discard the changes, call `AcMapLayer.DiscardFeatureChanges()`.

Creating SDF files

Many geospatial operations require creating new layers, adding features to the layers, and adding the layers to the map. The layers can be temporary or permanent. In AutoCAD Map 3D this is handled using SDF files.

An SDF file contains a single schema that can define multiple feature classes.

To create an SDF file, first define the schema and any feature classes used in the schema. For each feature class, define a list of properties. In most cases, the properties should include an identity property and a geometry property, as well as any other properties.

Use the FDO API to create the schema.

The following creates a simple schema with an autogenerated ID property, a string property, and a geometry property.

```

// Create the feature class definition

FeatureClass classDefinition =
    new FeatureClass("newClass", "class description");

// Add an autogenerated identity property

DataPropertyDefinition idProp =
    new DataPropertyDefinition("id", "property desc");
idProp.DataType = DataType.DataType_Int32;
idProp.IsAutoGenerated = true;
classDefinition.Properties.Add(idProp);
classDefinition.IdentityProperties.Add(idProp);

// Add a name property

DataPropertyDefinition nameProp =
    new DataPropertyDefinition("Name", "property desc");
nameProp.DataType = DataType.DataType_String;
classDefinition.Properties.Add(nameProp);

// Add a geometry property

GeometricPropertyDefinition geomProp =
    new GeometricPropertyDefinition("SHPGEOM", "property desc");
geomProp.GeometryTypes = MgGeometryType.Polygon;
classDefinition.Properties.Add(geomProp);
classDefinition.GeometryProperty = geomProp;

// Create the schema and add the class definition

FeatureSchema schema =
    new FeatureSchema("SHP_Schema", "Schema description");
schema.Classes.Add(classDefinition);

```

Once the schema has been created, use the FDO API to create the SDF file, set the spatial context, and apply the schema. The following is modified from the utility classes in the sample applications.

```

// Set connection properties

IConnectionPropertyDictionary connProperties =
    conn.ConnectionInfo.ConnectionProperties;
connProperties.SetProperty("File", sdfPath);
connProperties.SetProperty("ReadOnly", false.ToString());

// Create data store, ie. SDF file

ICreateDataStore createdSCmd =
    conn.CreateCommand(CommandType.CommandType_CreateDataStore)
    as ICreateDataStore;
createdSCmd.DataStoreProperties.SetProperty("File", sdfPath);
createdSCmd.Execute();

try
{
    // Open the connection
    ConnectionState connState = conn.Open();
    int retryRound = 0;
    while (++retryRound < 5 // Try 5 times
        && (connState == ConnectionState.ConnectionState_Pending
            || connState == ConnectionState.ConnectionState_Busy))
    {
        connState = conn.Open();
    }
    if (retryRound >= 5)
    {
        throw new
            InvalidOperationException("Failed to connect to file "
                + sdfPath);
    }

    // Create spatial context, including coordinate system

    AcMapMap currentMap = AcMapMap.GetCurrentMap();
    string mapSRS = currentMap.GetMapSRS();

    ICreateSpatialContext createSCCmd =
        conn.CreateCommand(CommandType.CommandType_CreateSpatialContext)
        as ICreateSpatialContext;
    createSCCmd.CoordinateSystemWkt = mapSRS;
    createSCCmd.Name = "";
}

```

```

createSCCmd.CoordinateSystem = mapSRS;
createSCCmd.Extent = new byte[] { 0, 0, 0, 0 };
createSCCmd.Description = "Description";
createSCCmd.XYTolerance = 0.0;
createSCCmd.ZTolerance = 0.0;
createSCCmd.Execute();

// Apply schema
if (schema.ElementState !=
    SchemaElementState.SchemaElementState_Unchanged)
{
    IApplySchema applySchemaCmd =
        conn.CreateCommand(CommandType.CommandType_ApplySchema)
        as IApplySchema;
    applySchemaCmd.FeatureSchema = schema;
    applySchemaCmd.Execute();
}
}
catch (System.Exception e)
{
    throw e; // Or add exception handling
}
finally
{
    // Close connection
    if (conn != null)
    {
        conn.Close();
    }
}
}

```


Maps and Layers

4

Overview

A map is composed of layers, where each layer represents data from a feature source.

AutoCAD Map 3D and Autodesk MapGuide differ in some of the ways they handle map layers. Because MapGuide is a web-based mapping product, it must coordinate layer display based on information from the client and the server. AutoCAD Map 3D can handle all layer display directly from the application.

NOTE The Geospatial Platform API deals strictly with layers from FDO feature sources. If a map in AutoCAD Map 3D includes other layers, they must be handled using the appropriate methods from the .NET API.

A layer (`AcMapLayer` object), has a `LayerDefinition` property that points to a layer definition in the resource repository. The layer definition content is XML that conforms to the `LayerDefinition.xsd` schema. Among other elements, it contains a `ResourceId` element that identifies the feature source for the layer.

Basic Layer Properties

A map contains one or more layers (`AcMapLayer` objects) that are rendered to create a composite image. The `AcMapLayer` class, which applies only to AutoCAD Map 3D, is derived from `MgLayerBase`, which is part of the common Geospatial Platform API shared between AutoCAD Map 3D and MapGuide.

Each layer has properties that determine how it displays in the map and map legend. Some of the properties are:

- **Layer name:** A unique identifier
- **Visibility:** whether the layer should be displayed in the map. Note that actual visibility is dependent on more than just the visibility setting for a layer. See [Layer Visibility](#) (page 43) for further details.
- **Selectable:** Whether features in the layer are selectable. This only applies to layers containing feature data. It does not apply to layers containing raster data.

`AcMapMap.GetLayers()` returns an `MgLayerCollection` object that contains all the layers in the map. `MgLayerCollection.GetItem()` returns an individual `MgLayerBase` object, by either index number in the collection or layer name.

Layers in the collection are sorted by drawing order, with the top layers at the beginning of the collection.

Layer Groups

Layers can be optionally grouped into layer groups. Layers in the same group are displayed together in the Display Manager.

The visibility for all layers in a group can be set at the group level. If the group visibility is turned off then none of the layers in the group will be visible, regardless of their individual visibility settings. If the group visibility is turned on, then individual layers within the group can be made visible or not visible separately.

Layer groups can be nested so a group can contain other groups. This provides a finer level of control for handling layer visibility or for grouping layers into legend groups.

`AcMapMap.GetLayerGroups()` returns an `MgLayerGroupCollection` object that contains all the top-level layer groups in the map.

Each layer group in a map must have a unique name, even if it is nested within another group.

Layer Visibility

Whether a layer is visible in a given map depends on three criteria:

- The visibility setting for the layer
- The visibility settings for any groups that contain the layer
- The map view scale and the layer definition for that view scale

In order for a layer to be visible, its layer visibility must be on, the visibility for any group containing the layer must be on, and the layer must have a style setting defined for the current map view scale.

Example: Actual Visibility

For example, assume that there is a layer named Roads that is part of the layer group Transportation. The layer has view style defined for the scale ranges 0–10000 and 10000–24000.

The following table shows some possible settings of the various visibility and view scale settings, and their effect on the actual layer visibility.

Layer Visibility	Group Visibility	View Scale	Actual Visibility
On	On	10000	On
On	On	25000	Off
On	Off	10000	Off
Off	On	10000	Off

Manipulating Layers

Modifying basic layer properties and changing layer visibility settings can be done directly by setting properties like `AcMapLayer.Name`, `AcMapLayer.Selectable`, and `AcMapLayer.Visible`. More complex manipulation requires modifying layer resources in the resource repository.

Changing Visibility

To query the actual layer visibility, call `AcMapLayer.IsVisible()`. There is no method to set actual visibility because it depends on other visibility settings.

To query the visibility setting for a layer, call `AcMapLayer.GetVisible()`. To change the visibility setting for a layer, call `AcMapLayer.SetVisible()`.

To query the visibility setting for a layer group, call `AcMapLayerGroup.GetVisible()`. To change the visibility setting for a layer group, use `AcMapLayerGroup.SetVisible()`.

To change the layer visibility for a given view scale, modify the layer resource and save it back to the repository.

Layer Definition

The feature source and styling for a layer are set using a `LayerDefinition`, stored in the resource repository. The layer definition conforms to the `LayerDefinition.xsd` *LayerDefinition.xsd* schema. Raster layers and vector layers have different styling requirements and capabilities.

For both raster and vector data, a single layer definition can contain multiple scale ranges, which define the styling for particular view scales. For example, a vector layer could have the following scale ranges:

- 0 to 10,000
- 10,000 to 500,000
- 500,000 to infinity

Features in the layer could be styled differently or be hidden completely for the different scale ranges.

For a vector layer, each scale range can contain styling information for the following:

- `AreaTypeStyle`, used for polygons
- `LineTypeStyle`, used for lines and curves
- `PointTypeStyle`, used for points

The layer definition can be any valid XML that conforms to the schema. To help generate the XML, the AutoCAD Map 3D samples use classes generated from `LayerDefinition.xsd` by `xsd.exe`.

For example, to create a vector layer definition that contains 2 scale ranges, do the following:

- Create the two scale ranges. For each, specify the minimum and maximum scale for the range. Define the necessary elements, like fill and stroke settings.
- Create the vector layer definition. Add the scale ranges.
- Set the feature name to the schema/feature class in the feature source.
- Set the resource id to point to the feature source containing the layer data.
- Set the geometry to the geometry property of the feature class.
- Create a layer definition that contains the vector layer definition.
- Store the layer definition in the resource repository.
- Create a new layer that references the layer definition. Add the layer to the layer collection for the current map.

The example code that follows uses the following namespace declarations:

- `using System.IO` (for the `StringWriter` class)
- `using System.Text` (for the `Encoding` class)
- `using System.Xml.Serialization` (for XML serialization routines)
- `using OSGeo.MapGuide.Schema.LayerDefinition` (generated by `xsd.exe`. Code for this class is available with the AutoCAD Map 3D samples.)

The following assumes that the feature source contains only polygons. Feature sources containing line or point data would require styling for those feature types as well.

```

// Create the first scale range

VectorScaleRangeType range = new VectorScaleRangeType();

AreaRule areaRule = new AreaRule();

AreaSymbolizationFillType areaSymFillType =
    new AreaSymbolizationFillType();

FillType fillType = new FillType();
fillType.FillPattern = "Solid";
fillType.ForegroundColor = "9900ffaa";
fillType.BackgroundColor = "FF000000";

StrokeType strokeType = new StrokeType();
strokeType.LineStyle = "Solid";
strokeType.Thickness = "0.0";
strokeType.Color = "FF000000";
strokeType.Unit = LengthUnitType.Centimeters;

areaSymFillType.Fill = fillType;
areaSymFillType.Stroke = strokeType;

areaRule.Item = areaSymFillType;
areaRule.LegendLabel = "";

object[] items = new object[1];
items[0] = new AreaRule[] { areaRule };
range.Items = items;
range.MinScale = 0.0;
range.MaxScale = 10000000.0;
range.MinScaleSpecified = true;
range.MaxScaleSpecified = true;

// Second scale range

VectorScaleRangeType range2 = new VectorScaleRangeType();

AreaRule areaRule2 = new AreaRule();

AreaSymbolizationFillType areaSymFillType2 =
    new AreaSymbolizationFillType();

```

```

FillType fillType2 = new FillType();
fillType2.FillPattern = "Solid";
fillType2.ForegroundColor = "ffffff00";
fillType2.BackgroundColor = "FF000000";

StrokeType strokeType2 = new StrokeType();
strokeType2.LineStyle = "Solid";
strokeType2.Thickness = "0.0";
strokeType2.Color = "FF000000";
strokeType2.Unit = LengthUnitType.Centimeters;

areaSymFillType2.Fill = fillType2;
areaSymFillType2.Stroke = strokeType2;

areaRule2.Item = areaSymFillType2;
areaRule2.LegendLabel = "";

object[] items2 = new object[1];
items2[0] = new AreaRule[] { areaRule2 };
range2.Items = items2;
range2.MinScale = 10000000.0;
range2.MaxScale = Double.MaxValue;
range2.MinScaleSpecified = true;
range2.MaxScaleSpecified = true;

// Now create a vector layer definition and add the scale ranges.

VectorLayerDefinitionType vectorLayerDef =
    new VectorLayerDefinitionType();
vectorLayerDef.VectorScaleRange = new VectorScaleRangeType[2];
vectorLayerDef.VectorScaleRange.SetValue(range, 0);
vectorLayerDef.VectorScaleRange.SetValue(range2, 1);

// Set the resource id to point to the feature source for
// the layer. Set the feature name to a feature class in
// the feature source. Set the geometry property to the
// default geometry property for the feature class.

vectorLayerDef.ResourceId = "Library://Data/SAMPLE.FeatureSource";
vectorLayerDef.FeatureName = "Schema1:FeatureClass1";
vectorLayerDef.Geometry = "Geometry";

// Create a layer definition containing the vector layer

```

```

// definition.

LayerDefinition layerDef = new LayerDefinition();
layerDef.Item = vectorLayerDef;
layerDef.version = "1.0.0";

// Convert to an XML string.

using (StringWriter writer = new StringWriter())
{
    XmlSerializer xs = new XmlSerializer(typeof(LayerDefinition));
    xs.Serialize(writer, layerDef);

    // Convert to UTF-8.

    byte[] unicodeBytes =
        Encoding.Unicode.GetBytes(writer.ToString());
    byte[] utf8Bytes =
        Encoding.Convert(Encoding.Unicode, Encoding.UTF8,
            unicodeBytes);

    MgByteSource source =
        new MgByteSource(utf8Bytes, utf8Bytes.Length);

    // Update the resource in Map.

    MgResourceService rs =
        AcMapServiceFactory.GetService(MgServiceType.ResourceService)
        as MgResourceService;

    // Store the layer definition in the repository.

    MgResourceIdentifier newLayerDefId =
        new MgResourceIdentifier(
            "Library://myNewLayer.LayerDefinition");
    rs.SetResource(newLayerDefId, source.GetReader(), null);

    // Create a new layer that references the layer definition.
    // Add this to the layers in the current map.

    AcMapMap currentMap = AcMapMap.GetCurrentMap();
    MgLayerCollection layers = currentMap.GetLayers();
    AcMapLayer layer;

```

```
layer = new AcMapLayer(newLayerDefId, rs);  
layer.Name = "testlayer";  
layers.Add(layer);  
}
```

This creates the following XML:

```

<?xml version="1.0" encoding="utf-16"?>
<LayerDefinition
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  version="1.0.0">
  <VectorLayerDefinition>
    <ResourceId>Library://Data/SAMPLE.FeatureSource</ResourceId>
    <FeatureName>Schemal:FeatureClass1</FeatureName>
    <FeatureNameType>FeatureClass</FeatureNameType>
    <Geometry>Geometry</Geometry>
    <VectorScaleRange>
      <MinScale>0</MinScale>
      <MaxScale>10000000</MaxScale>
      <AreaTypeStyle>
        <AreaRule>
          <LegendLabel />
          <AreaSymbolization2D>
            <Fill>
              <FillPattern>Solid</FillPattern>
              <ForegroundColor>9900ffaa</ForegroundColor>
              <BackgroundColor>FF000000</BackgroundColor>
            </Fill>
            <Stroke>
              <LineStyle>Solid</LineStyle>
              <Thickness>0.0</Thickness>
              <Color>FF000000</Color>
              <Unit>Centimeters</Unit>
            </Stroke>
          </AreaSymbolization2D>
        </AreaRule>
      </AreaTypeStyle>
    </VectorScaleRange>
    <VectorScaleRange>
      <MinScale>10000000</MinScale>
      <MaxScale>1.7976931348623157E+308</MaxScale>
      <AreaTypeStyle>
        <AreaRule>
          <LegendLabel />
          <AreaSymbolization2D>
            <Fill>
              <FillPattern>Solid</FillPattern>
              <ForegroundColor>ffffff00</ForegroundColor>
              <BackgroundColor>FF000000</BackgroundColor>
            </Fill>
            <Stroke>
              <LineStyle>Solid</LineStyle>
              <Thickness>0.0</Thickness>
              <Color>FF000000</Color>
              <Unit>Centimeters</Unit>
            </Stroke>
          </AreaSymbolization2D>
        </AreaRule>
      </AreaTypeStyle>
    </VectorScaleRange>
  </VectorLayerDefinition>
</LayerDefinition>

```



```

        </Fill>
        <Stroke>
            <LineStyle>Solid</LineStyle>
            <Thickness>0.0</Thickness>
            <Color>FF000000</Color>
            <Unit>Centimeters</Unit>
        </Stroke>
    </AreaSymbolization2D>
</AreaRule>
</AreaTypeStyle>
</VectorScaleRange>
</VectorLayerDefinition>
</LayerDefinition>

```

Layers With Joined Feature Sources

To create a layer definition for a layer that uses a feature source containing a join, create the layer definition as described in [Layer Definition](#) (page 44). Set the `<FeatureName>` element to the join name as defined in the feature source. Set the `<FeatureNameType>` element to `NamedExtension`. For example, the following portion of a layer definition uses the join named “Parcels_Joins1”.

```

<?xml version="1.0" encoding="utf-16"?>
<LayerDefinition
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    version="1.0.0">
    <VectorLayerDefinition>
        <ResourceId>Library://Data/SAMPLE.FeatureSource</ResourceId>
        <FeatureName>Parcels_Joins1</FeatureName>
        <FeatureNameType>NamedExtension</FeatureNameType>
    </VectorLayerDefinition>
</LayerDefinition>

```

See [Joins](#) (page 23) for details about the feature source definition.

Filtering Layers By Geometry

The layer definition for a vector layer can include a `<Filter>` element to select only certain features. The filter is a string that contains a valid FDO selection expression. See [Selecting Using the API](#) (page 30) for a description of filters.

For example, assume the following:

- `geomPoly` is an `MgPolygon` representing the area of interest.

- `geometry` is the default geometry property for a feature class.

The following creates a filter to select only those features that intersect the polygon.

```
MgWktReaderWriter rw = new MgWktReaderWriter();
string wktGeom = rw.Write(geomPoly);
string filter = "geometry intersects GeomFromText(" +
    wktGeom + ")" );
```

To modify the sample in [Layer Definition](#) (page 44) to use the filter, set the `<Filter>` element as follows:

```
vectorLayerDef.Filter = filter;
```

Modifying Layer Style

To modify the styling for a layer, retrieve the layer definition from the repository, make changes to it, store it back to the repository, and refresh the layer.

To retrieve the layer definition, get its resource identifier from the map layer. Get the resource content and deserialize it using an `XmlSerializer` object.

```

AcMapMap currentMap = AcMapMap.GetCurrentMap();
MgLayerCollection layers = currentMap.GetLayers();
MgResourceService rs =
    AcMapServiceFactory.GetService(MgServiceType.ResourceService)
    as MgResourceService;

// Get the layer definition. Convert it to an XML string
// stored in a byte reader.

AcMapLayer layer1 = layers.GetItem(layerName) as AcMapLayer;
MgResourceIdentifier layerDefId = layer1.GetLayerDefinition();
MgByteReader layerReader = rs.GetResourceContent(layerDefId);

// Deserialize it to create a LayerDefinition object.

XmlSerializer layerDefSerializer = new
    XmlSerializer(typeof(LayerDefinition));
LayerDefinition xmlLayerDef =
    layerDefSerializer.Deserialize(new
        StringReader(layerReader.ToString()))
    as LayerDefinition;

```

Updating the `LayerDefinition` object is similar to creating a new one. Find the correct elements and set their values. For example, the following changes the foreground color for the area rule in the first scale range, then it updates the layer definition in the resource repository and refreshes the layer.

```

// Should be a VectorLayerDefinitionType

if (xmlLayerDef.Item is VectorLayerDefinitionType)
{
    VectorLayerDefinitionType xmlVectorLayerDef =
        xmlLayerDef.Item as VectorLayerDefinitionType;

    // Check the rules for the first scale range.
    // Only update rules if they are the correct type.

    object[] items = xmlVectorLayerDef.VectorScaleRange[0].Items;
    foreach (object item in items)
    {
        Type type = item.GetType();

        if (type.Name == typeof(AreaRule[]).Name)
        {
            AreaRule[] areaRules = item as AreaRule[];

            // OK, got the correct area rule. Set the foreground color.

            if (areaRules.Length > 0)
            {
                areaRules[0].Item.Fill.ForegroundColor = newColor;
            }
        }
    }

    // Now convert it back to an XML string and update the
    // resource.

    using (StringWriter writer = new StringWriter())
    {
        // Get the xml string
        layerDefSerializer.Serialize(writer, xmlLayerDef);

        // Create the resource
        byte[] unicodeBytes =
            Encoding.Unicode.GetBytes(writer.ToString());
        byte[] utf8Bytes =
            Encoding.Convert(Encoding.Unicode,
                Encoding.UTF8, unicodeBytes);
    }
}

```

```
MgByteSource source =  
    new MgByteSource(utf8Bytes, utf8Bytes.Length);  
  
// Update the resource in Map  
rs.SetResource(layer1.LayerDefinition,  
    source.GetReader(), null);  
  
layer1.ForceRefresh();  
}  
}
```


Geometry

5

Overview

The AutoCAD Map 3D Geospatial Platform API includes a complete set of classes for working with geometry. This includes the following:

- Describing simple and complex geometric objects
- Performing spatial comparisons between objects
- Creating new objects based on the intersection, difference, or union of existing objects
- Converting between different coordinate systems
- Creating buffers around objects

Geometry Objects

`MgGeometry` is the base class for all the geometry types. The simple geometry types are:

- `MgPoint` — a single point
- `MgLineString` — a series of connected line segments
- `MgCurveString` — a series of connected curve segments
- `MgPolygon` — a polygon with sides formed from line segments
- `MgCurvePolygon` — a polygon with sides formed from curve segments

The curve segments are circular arcs, defined by a start point, an end point, and a control point.

Complex types are formed by aggregating simple types. The complex types are:

- `MgMultiPoint` — a group of points
- `MgMultiLineString` — a group of line strings
- `MgMultiCurveString` — a group of curve strings
- `MgMultiPolygon` — a group of polygons
- `MgMultiCurvePolygon` — a group of curve polygons
- `MgMultiGeometry` — a group of simple geometry objects of any type

Comparing Geometry Objects

The `MgGeometry` class contains methods for comparing different geometry objects. These are similar to the spatial filters described in [Selecting Using the API](#) (page 30). Methods to test spatial relationships include:

- `Contains()`
- `Crosses()`
- `Disjoint()`
- `Equals()`
- `Intersects()`
- `Overlaps()`
- `Touches()`
- `Within()`

For example, if you have an `MgLineString` object `line` and an `MgPolygon` object `polygon`, you can test if the line crosses the polygon with a call to

```
line.Crosses(polygon)
```


Methods to create new geometry objects from the point set of two other geometries include:

- `Difference()`
- `Intersection()`
- `SymmetricDifference()`
- `Union()`

Complete details are in the Geometry module of the Geospatial Platform API reference, under Spatial Relationships.

Coordinate Systems

A single map will often combine data from different sources, and the different sources may use different coordinate systems. The map has its own coordinate system, and any feature sources used in the map may have different coordinate systems. It is important for display and analysis that all locations are transformed to the same coordinate system.

NOTE A coordinate system can also be called a spatial reference system (SRS) or a coordinate reference system (CRS). This guide uses the abbreviation SRS.

AutoCAD Map 3D supports three different types of coordinate system:

- Arbitrary X-Y
- Geographic, or latitude/longitude
- Projected

An `MgCoordinateSystem` object represents a coordinate system.

NOTE You cannot transform between arbitrary X-Y coordinates and either geographic or projected coordinates.

To create an `MgCoordinateSystem` object from an `AcMapMap` object,

- Get the WKT representation of the map coordinate system, using `AcMapMap.GetMapSRS()`.
- Create an `MgCoordinateSystem` object, using `MgCoordinateSystemFactory.Create()`.

To create an `MgCoordinateSystem` object from a map layer,

- Get the feature source for the layer.
- Get the active spatial context for the feature source.
- Convert the spatial context to a WKT.
- Create an `MgCoordinateSystem` object from the WKT.

To transform geometry from one coordinate system to another, create an `MgCoordinateSystemTransform` object using the two coordinate systems. Apply this transform to the `MgGeometry` object.

For example, if you have geometry representing a feature on a layer that uses one coordinate system, and you want to compare it to a feature on another layer that uses a different coordinate system, perform the following steps:

```
string featureSource1 = layer1.GetFeatureSourceId();
MgSpatialContextsReader contexts1 =
    featureService.GetSpatialContexts(
        featureSource1, true);
contexts1.ReadNext();
string srs1 = contexts1.GetCoordinateSystemWkt();

string featureSource2 = layer2.GetFeatureSourceId();
MgSpatialContextsReader contexts2 =
    featureService.GetSpatialContexts(
        featureSource2, true);
contexts2.ReadNext();
string srs2 = contexts2.GetCoordinateSystemWkt();

MgCoordinateSystemFactory coordsysFactory =
    new MgCoordinateSystemFactory();
MgCoordinateSystemTransform xform =
    coordsysFactory.GetTransform(srs1, srs2);
MgGeometry geometry1xform = geometry1.Transform(xform);
```

Measuring Distance

Measuring distance in geographic or projected coordinate systems requires great circle calculations. Both `MgGeometry.Buffer()` and `MgGeometry.Distance()` accept a measurement parameter that defines the

great circle to be used. If the measurement parameter is null, the calculation is done using a linear algorithm.

Create the measurement parameter, an `MgCoordinateSystemMeasure` object, from the `MgCoordinateSystem` object.

Distance is calculated in the units of the SRS. `MgCoordinateSystem` includes two methods, `ConvertCoordinateSystemUnitsToMeters()` and `ConvertMetersToCoordinateSystemUnits()`, to convert to and from linear distances.

For example, to calculate the distance between two `MgGeometry` objects `a` and `b`, using the coordinate system `srs`, perform the following steps:

```
MgCoordinateSystemMeasure measure = srs.GetMeasure();
double distInMapUnits = a.Distance(b, measure);
double distInMeters = srs.ConvertCoordinateSystemUnitsToMeters(
    distInMapUnits);
```

Another way to calculate the distance is to use `MgCoordinateSystemMeasure.GetDistance()`, as in the following:

```
distInMapUnits = measure.GetDistance(a, b);
```

Creating a Buffer

To create a buffer around a feature, use the `MgGeometry.Buffer()` method. This returns an `MgGeometry` object that you can use for further analysis. For example, you could display the buffer by creating a feature in a temporary feature source and adding a new layer to the map. You could also use the buffer geometry as part of a spatial filter. For example, you might want to find all the features within the buffer zone that match certain criteria, or you might want to find all roads that cross the buffer zone.

To create a buffer, get the geometry of the feature to be buffered. If the feature is being processed in an `MgFeatureReader` as part of a selection, this requires getting the geometry data from the feature reader and converting it to an `MgGeometry` object. For example:

```
MgByteReader geometryData =
    featureReader.GetGeometry(geometryName);
MgGeometry featureGeometry = agfReaderWriter.Read(geometryData);
```

If the buffer is to be calculated using coordinate system units, create an `MgCoordinateSystemMeasure` object from the coordinate system for the map. For example:

```

string mapWktSrs = currentMap.GetMapSRS();
MgCoordinateSystemFactory coordSysFactory =
    new MgCoordinateSystemFactory();
MgCoordinateSystem srs = coordSysFactory.Create(mapWktSrs);
MgCoordinateSystemMeasure srsMeasure = srs.GetMeasure();

```

Use the coordinate system measure to determine the buffer size in the coordinate system, and create the buffer object from the geometry to be buffered.

```

double srsDist =
    srs.ConvertMetersToCoordinateSystemUnits(bufferDist);
MgGeometry bufferGeometry =
    featureGeometry.Buffer(srsDist, srsMeasure);

```

To display the buffer in the map, perform the following steps:

- Create a feature source for the buffer. This may require creating a new file for the feature source. See [Creating SDF files](#) (page 36).
- Create a layer that references the feature source. Add it to the map and make it visible.
- Create a new feature using the buffer geometry and insert it into the feature source.

For example, the following code assumes that the layer `resLayer` has been created using a feature source with a geometry property of `Geometry`. It adds a new feature using the buffer geometry.

```

MgPropertyCollection properties = new MgPropertyCollection();
properties.Add(new MgGeometryProperty("Geometry",
    agfReaderWriter.Write(bufferGeometry)));

MgFeatureCommandCollection commands =
    new MgFeatureCommandCollection();
commands.Add(new MgInsertFeatures(resFeatureClassName,
    properties));

resLayer.UpdateFeatures(commands);

```

To use the buffer as part of a query, create a spatial filter using the buffer geometry, and use this in a call to `AcMapLayer.SelectFeatures()`. For example, the following code uses a basic filter and a spatial filter to select parcels inside the buffer area that are of type “MFG”. You can use the `MgFeatureReader` to

perform tasks like generating a report of the parcels, or creating a new layer that puts point markers on each parcel.

```
MgFeatureQueryOptions queryOptions = new MgFeatureQueryOptions();
queryOptions.SetFilter("PARCELTYPE = 'MFG'");
queryOptions.SetSpatialFilter('SHPGEOM', bufferGeometry,
    MgFeatureSpatialOperations.Inside);

MgResourceIdentifier featureResId = new MgResourceIdentifier(
    "Library://Data/Parcels.FeatureSource");
MgFeatureReader featureReader =
    layer.SelectFeatures(queryOptions);
```


Coordinate System API

6

The Earth Moves Under Our Feet

We use coordinate systems to track the moving earth. The action of plate tectonics constantly changes the geographic/cartographic coordinates of physical landmarks. In addition, our ability to measure improves over time. The effect of these forces is the creation of new coordinate system, ellipsoid, and datum definitions to reflect the new reality. As well, transformation functions must be defined that map coordinates in dated coordinate systems to coordinates in more recently defined coordinate systems.

The `OSGeo.MapGuide` namespace contains a set of classes whose base name is `MgCoordinateSystem`. These classes provide access to the CS-Map coordinate system functionality in both the Map 3D and MapGuide environments. The CS-Map software was formerly the property of Mentor Software Inc, a Colorado corporation. Autodesk acquired most of the intellectual property of Mentor Software on September 24, 2007. As part of the transaction, Mentor Software was granted a license to use all of the acquired intellectual property as necessary for the purpose of supporting existing Mentor Software clients. Autodesk donated the CS-Map software to the OSGeo Foundation in 2008. Autodesk personnel actively participate in the maintenance of this software.

The coordinate system factory is used to get the coordinate system categories and the coordinate system catalog. The coordinate system catalog is used to get the various dictionaries. The coordinate system dictionary contains definitions for earth-based and Euclidean coordinate systems. The ellipsoid dictionary contains definitions for the ellipsoid models of the earth's surface. The datum dictionary contains definitions for the transformations that track the changes in the coordinate address of physical landmarks over time. The following code instantiates the coordinate system factory, catalog and dictionaries.

```
MgCoordinateSystemFactory coordSysFactory = new MgCoordinateSystem
Factory();
MgCoordinateSystemCatalog csCatalog = coordSysFactory.GetCatalog();
MgCoordinateSystemDictionary csDict = csCatalog.GetCoordinateSys
temDictionary();
MgCoordinateSystemDatumDictionary datumDict = csCatalog.GetDatum
Dictionary();
MgCoordinateSystemEllipsoidDictionary ellipsoidDict = csCatalog.Ge
tEllipsoidDictionary();
```

Each coordinate system is identified by a code. Use an `MgCoordinateSystemFormatConverter` object to map this code to an EPSG code or to one of the various flavors of WKT (OGC, Oracle, ESRI, GeoTiff, GeoTools). The code for instantiating this class follows:

```
MgCoordinateSystemFormatConverter formatConverter = csCatalog.Get
FormatConverter();
```

Coordinate systems, ellipsoids, and datums may be functionally equivalent. Use an `MgCoordinateSystemMathComparator` object to determine this. The code for instantiating this class follows.

```
MgCoordinateSystemMathComparator mathComparator = csCatalog.Get
MathComparator();
```

Each cartographic coordinate system has projection information associated with it. This information is accessed through the `MgCoordinateSystemProjectionInformation` class. The code for instantiating this class follows:

```
MgCoordinateSystemProjectionInformation projectionInformation =
csCatalog.GetProjectionInformation();
```

Each coordinate system has a unit of measure. Information about this unit is retrieved using the `MgCoordinateSystemUnitInformation` class. Code for instantiating this class follows.

```
MgCoordinateSystemUnitInformation unitInformation = csCatalog.Ge
tUnitInformation();
```

Coordinate System Types

CS-Map classifies coordinate systems into 3 types: projected, geographic, and arbitrary.

Projected and geographic coordinate system definitions reference the earth in the assignment of values to origin and extents. An arbitrary coordinate system has an abstract spatial context; its origin and extents are defined arbitrarily.

The following code extracts all of the coordinate system names from the coordinate system dictionary, uses those names to get all of the coordinate system definitions from the dictionary, determines the type (projected, geographic, or arbitrary) of the coordinate system from its definition, and stores the name of the coordinate system in a list according to its type.

```
List<string> arbitraryCs = new List<string>();
List<string> geographicCs = new List<string>();
List<string> projectedCs = new List<string>();
// csDict defined above
MgCoordinateSystemEnum csDictEnum = csDict.GetEnum();
// the following line gets all of the names in the dictionary
// uses those names to get the coordinate
int csCount = csDict.GetSize();
MgStringCollection csNames = csDictEnum.NextName(csCount);
string csName = null;
MgCoordinateSystem cs = null;
int csType = 0;
for (int i = 0; i < csCount; i++)
{
    csName = csNames.GetItem(i);
    cs = csDict.GetCoordinateSystem(csName);
    csType = cs.GetType();
    if (csType == MgCoordinateSystemType.Arbitrary)
    {
        arbitraryCs.Add(csName);
    }
    else if (csType == MgCoordinateSystemType.Geographic)
    {
        geographicCs.Add(csName);
    }
    else if (csType == MgCoordinateSystemType.Projecte
    {
        projectedCs.Add(csName);
    }
}
```

Coordinate System Categories

The coordinate systems are grouped into categories. The “LL” category contains all of the geographic coordinate systems. The projected coordinate systems are categorized according to country or region, for example, “Afghanistan,” “Africa,” “USA, Idaho,” “UTM, International Ellipsoid (No datum),” “UTM, WGS84 Datum,” and “World/Continental.” The code for getting the categories and then the coordinate systems belonging to that category follows.

```
MgStringCollection categories = coordSysFactory.EnumerateCategories();
for (int i = 0; i < categories.GetCount(); i++)
{
    string category = categories.GetItem(i);
    MgBatchPropertyCollection batchProperties = coordSysFactory.EnumerateCoordinateSystems(category);
    for (int j = 0; j < batchProperties.Count; j++)
    {
        MgPropertyCollection properties = batchProperties[j];
        MgStringProperty codeProp = properties.GetItem("Code") as MgStringProperty;
        string code = codeProp.GetValue();
    }
}
```

Projection Grouping and Characterization

The projected coordinate systems are grouped together and the groups are assigned a code. This topic describes these projection groupings in terms of properties such as the method used to map the earth’s surface onto a flat plane. The topic contains two elements: a description of the properties and the mapping of the projection groups to the properties.

The `MgCoordinateSystem` class has a method called `GetProjectionCode` which returns an integer. The meaning of the integer value can be determined from the set of integer constants contained in the `MgCoordinateSystemProjectionCode` class. These integer constants map to a set of coordinate system categories that are used to group the coordinate systems. The following code shows the retrieval of the projection code and its comparison to one of the projection code constants.

```

// csDict is defined above
// "CANADA-ALBERS" is in the Alber group of projections
MgCoordinateSystem cs = csDict.GetCoordinateSystem("CANADA-AL
BERS");
int projectionCode = cs.GetProjectionCode();
if (cs.GetProjectionCode() == MgCoordinateSystemProjectionCode.Al
ber)
{
    // do something
}

```

The mapping of the projection code values to properties is programmatically extracted from the CS Map file CSdataPJ.c. In CSdataPJ.c there is a set of macro definitions of the form “#define cs_PRJFLGS_<GroupName> <OR’d set of bit flags>”. The bit flag definitions are contained in cs_map.h and are of the form “#define cs_PRJ_FLG_<FlagName> <bitDefinition>”.

Each projection code has a flag indicating the projection method. The four possibilities are flat plane (azimuthal), conical, cylindrical, and other. Other is defined as being neither flat plane, conical or cylindrical. The presentation of the projection code properties is divided into four tables according to the projection method used for the coordinate systems belonging to group represented by the projection code.

Properties

In the following table the property names are the same as the flag names extracted from the code. The intent is to provide entry points into the CS Map code body as well as to describe the properties.

Property	Description
SPHERE	True indicates that the spherical form of the projection is supported.
ELLIPS	True indicates that the ellipsoidal form of the projection is supported.
SCALK	True indicates that the scale factor along a parallel is determined using an analytical formula; otherwise it is determined empirically.
SCALH	True indicates that the scale factor along a meridian is determined using an analytical formula; otherwise it is determined empirically.
CNVRG	True indicates that the convergence angle is determined using an analytical formula; otherwise it is determined empirically. As you go East from the Central Meridian of a projection, you add the convergence angle to geodetic North to calculate the value of the grid North.

Property	Description
	As you go West from the Central Meridian, you subtract the convergence angle from geodetic North to calculate the value of the grid North.
CNFRM	True indicates that angles are preserved locally.
EAREA	True indicates that area is preserved.
EDIST	True indicates that distance from a point or line is preserved.
AZMTH	True indicates that the scale factors are measured along radial lines from an origin and along lines normal to the radial lines.
OBLQ	True indicates that the projection surface is tangent to a great circle that does not include the North and South poles.
TRNSV	True indicates that the cylindrical projection surface is tangent to a meridian and its anti-podal meridian and touches the North and South poles.
PSEUDO	True indicates that the projection does not conform to the standard application of a cylindrical or conic projection.
INTR	True indicates that the projection is interrupted. It is divided into several sections.
CYLND	True indicates the projection is cylindrical.
CONIC	True indicates that the projection is conic.
FLAT	True indicates that the projection uses a flat plane tangential to the globe at one point.
OTHER	True indicates that the projection uses a method that is not cylindrical, conic, or flat plane.
SCLRED	True indicates that the projection requires a Scale Reduction Factor parameter and is a secant projection.
ORGLAT	True indicates that an origin latitude is not supported.
ORGLNG	True indicates that an origin longitude is not supported.

The following table maps properties of projection groups whose projection method is flat plane (azimuthal).

Name	SPHERE	EL-LIPS	SCALK	SCALH	EAREA	AZMTH	EDIST	CN-FRM	SCLRED
AZMEA	x	x	x	x	x	x			
AZMED	x	x	x	x		x	x		
MSTRO	x	x	x					x	
NZLND		x						x	
OSTRO	x	x	x			x		x	x
PSTRO	x	x	x			x		x	x
PSTROSL	x	x	x			x		x	

The following table maps properties of projection groups whose projection method is conical.

SP=SPHERE; EL=ELLIPS; K=SCALK; H=SCALH; PS=PSEUDO; CV=CNVRG;													
Name	SP	EL	K	H	EAREA	PS	CN-FRM	ORGLAT	ORGLNG	CV	OB-LQ	SCLRED	EDIST
ALBER	x	x	x	x	x								
BONNE	x	x			x	x							
BPCNC	x						x	x	x				
KROVAK	x	x	x				x				x	x	x
KRVK95	x	x	x				x	x	x		x	x	x
LM1SP	x	x	x				x				x		x
LM2SP	x	x	x				x				x		
LMBLG	x	x	x				x				x		
LMTAN		x					x						x

SP=SPHERE; EL=ELLIPS; K=SCALK; H=SCALH; PS=PSEUDO; CV=CNVRG;													
Name	SP	EL	K	H	EAREA	PS	CN-FRM	ORGLAT	ORGLNG	CV	OB-LQ	SCLRED	EDIST
MNDOTL	x	x	x				x			x			
PLYCN	x	x	x	x					x				x
WCCSL	x	x	x				x			x			

The following table maps properties of projection groups whose projection method is cylindrical.

SP=SPHERE; EL=ELLIPS; K=SCALK; H=SCALH; TR=TRNV; LN=ORGLNG; CV=CNVRG; LT=ORGLAT; PS=PSEUDO;															
Name	SP	EL	K	H	TR	LN	CV	EDIST	CN-FRM	OB-LQ	SCLRED	LT	PS	EAREA	IN-TR
CSINI	x	x	x	x	x	x									
EDCYL	x		x	x			x	x							
GAUSSK	x	x	x		x	x	x		x						
HOM1XY	x	x	x			x			x	x	x	x			
MILLR	x		x	x		x	x					x			
MNDOTT	x	x	x		x	x	x		x						
MRCAT	x	x	x			x	x		x			x			
MRCATK	x	x	x			x	x		x		x	x			
OBQCYL	x	x							x	x	x				
OSTN02	x	x	x		x	x	x		x			x			
OSTN97	x	x	x		x	x	x		x			x			
ROBIN	x											x	x		
RSKEW	x	x	x			x			x	x	x	x			

SP=SPHERE; EL=ELLIPS; K=SCALK; H=SCALH; TR=TRNV; LN=ORGLNG; CV=CNVRG;
LT=ORGLAT; PS=PSEUDO;

Name	SP	EL	K	H	TR	LN	CV	EDIST	CN-FRM	OB-LQ	SCLRED	LT	PS	EAREA	IN-TR
RSKEWC	x	x	x			x			x	x	x	x			
RSKEWO	x	x	x			x			x	x	x	x			
SINUS	x	x	x	x			x					x	x	x	x
SOTRM	x	x	x		x	x	x		x		x				
SWISS	x	x							x	x					
TRMER	x	x	x		x	x	x		x		x				
TRMRKRG	x	x	x		x	x	x		x		x				
TRMRS	x	x	x		x	x	x		x		x				
UTM	x	x	x		x	x	x		x			x			
WCCST	x	x	x		x	x	x		x		x				

The following table maps properties of projection groups whose projection method is neither flat plane, conical, or cylindrical.

Name	SPHERE	ORGLAT	AZMTH	PSEUDO
VDGRN	x	x		
WINKL	x	x	x	x

Projection Parameters

A projected coordinate system may have 0 or more parameters. The projected coordinate systems in the groupings presented in the previous topic have the same properties and also the same number and kind of parameters. This topic contains two elements: a description of the parameters and the mapping of the projection groups to the parameters. The projection groups are subdivided

according to the projection method used by the group: flat plane, conical, cylindrical, or other (not flat plane, conical, or cylindrical).

The `MgCoordinateSystem` class has a method called `GetProjectionCode` which returns an integer code specifying the projection group to which this coordinate system belongs. These codes are defined in `MgCoordinateSystemProjectionCode`. Use the `MgCoordinateSystemProjectionCode` object to get projection information about the projection group. Use that information to get projection information particular to the `MgCoordinateSystem` instance. Use the `MgCoordinateSystemUnitInformation` class to get information about the projection unit. The following code shows this sequence of calls.


```

//csDict is defined above
// "CANADA-ALBERS" is a coordinate system in the Alber group of
projections
MgCoordinateSystem cs = csDict.GetCoordinateSystem("CANADA-AL
BERS");
// the unit codes identify the various linear and angular units
such as meter, foot, degree, and radian
// the unit codes are integer constants defined in MgCoordinateSys
temUnitCode
int unitCode = cs.GetUnitCode();
int projectionCode = cs.GetProjectionCode();
// unit type is either angular, linear, or unknown
// the unit types are integer constants defined in MgCoordinateSys
temUnitType
// there are 2 ways to get the unit type
int projInfoUnitType = projectionInformation.GetUnitType(projec
tionCode);
int csUnitType = unitInformation.GetUnitType(unitCode);
// an example of a unit abbreviation is M for meter
string unitAbbreviation = unitInformation.GetAbbreviation(unit
Code);
// an example of a unit tag string is METER
string unitTagString = unitInformation.GetTagString(unitCode);
// unit scale is relative to the meter
// if the unit is FOOT, the unit scale is 0.304800609601219
double unitScale = unitInformation.GetLinearUnitScale(unitCode);
bool usingOffset = projectionInformation.IsUsingOffset(projection
Code);
if (usingOffset)
{
    // false Easting
    doubl offsetX = cs.GetOffsetX();
    // false Northing
    doubleoffsetY = cs.GetOffsetY();
}
bool usingOriginLatitude = projectionInformation.IsUsingOriginLat
itude(projectionCode);
if (usingOriginLatitude)
{
    double originLatitude = cs.GetOriginLatitude();
}
bool usingOriginLongitude = projectionInformation.IsUsingOriginLon
gitude(projectionCode);

```

```

if (usingOriginLongitude)
{
    double originLongitude = cs.GetOriginLongitude();}
// see the quadrant topic
bool usingQuadrant = projectionInformation.IsUsingQuadrant(projectionCode);
if (usingQuadrant)
{
    int quadrant = cs.GetQuadrant();
}
// See the Secant Projection topic for a discussion of scale reduction
bool usingScaleReduction = projectionInformation.IsUsingScaleReduction(projectionCode);
if (usingScaleReduction)
{
    double scaleReduction = cs.GetScaleReduction();
}
int parameterCount = projectionInformation.GetParameterCount(projectionCode);
for (int j = 1; j <= parameterCount; j++)
{
    bool usingParameter = projectionInformation.IsUsingParameter(projectionCode, j);
    if (!usingParameter) continue;
    //Parameter types are defined in MgCoordinateSystemProjectionParameterType
    int parameterType = projectionInformation.GetParameterType(projectionCode, j);
    //the logical types are defined in MgCoordinateSystemProjectionLogicalType
    int parameterLogicalType = projectionInformation.GetParameterLogicalType(projectionCode, j)
    // the format types are defined in MgCoordinateSystemProjectionFormatType
    int parameterFormatType = projectionInformation.GetParameterFormatType(projectionCode, j);
    double parameterDefault = projectionInformation.GetParameterDefault(projectionCode, j);
    double parameterMin = projectionInformation.GetParameterMin(projectionCode, j);
    double parameterMax = projectionInformation.GetParameterMax(projectionCode, j);
}

```

```
double projectionParameter = cs.GetProjectionParameter(j);
}
```

The mapping of the projection code values to parameters is programmatically extracted from the CS Map file CSdataPJ.c. In CSdataPJ.c there is an array of structures called cs_PrjprmMap. Each structure maps a projection code to an array of parameter codes. The projection and parameter codes are defined in cs_map.h.

The following table describes the parameters.

Parameter	Description
0	Zero parameters.
CNTMER	Central Meridian, also known as Origin Longitude. Used in projection groups Cassini, GaussK, Miller, Mndott, Modp, c Mrcat, MrcatK, Plycn, Sotrm, Tm, Trmeraf, Trmrkr, Trmrs, and Wccst.
NSTDPLL, SSTDPLL	Northern Standard Parallel, also known as Standard Parallel 1, and Southern Standard Parallel, also known as Standard Parallel 2. Used in secant conic projections. Used in projection groups Alber, Lm2sp, Lmbgl, Lmbtaf, Mndotl, and Wccsl.
STDPLL	Standard Parallel. Used in projection groups Edcyl, Mrcat, Neacyl, and Winkl. See the discussion of the Caspian Sea projection.
GCPLNG, GCPLAT, GCAZM	Great Circle Point Longitude, Great Circle Point Latitude, and Great Circle Azimuth. These are used for the oblique Mercator projections. The central geodesic can be defined by a point and an azimuth or by 2 points. In practice it is always defined by a point and an azimuth. Used in projection groups Hom1uv, Hom1xy, Rskew, Rskewc and, with the exception of GCAZM, Rskewo.
YAXISAZ	Y Axis Azimuth. This is a non-standard parameter used in some azimuthal projections to define the direction of the Y axis. Used in projection groups Azede, Azmea, Azmed, and Sstro.
P1LNG, P1LAT, P2LNG, P2LAT, ADP1P2,	First Pole Longitude, First Pole Latitude, Second Pole Longitude, Second Pole Latitude, Angular Distance between the poles, Angular distance to the first standard parallel, Angular distance to the second standard parallel. These are used for the Bi-polar conformal conic projection, which was invented for the purpose of making a map of North and South America.

Parameter	Description
ADSP1, ADSP2	
CM- PLXAN, CMPLXBN	Complex Parameter A(n), Complex Parameter B(n). There are several projections which are based on an expansion series of complex numbers. CSMap supports up to 12 terms, that is, 12 pairs of real and imaginary coefficients. Used in projection group Mstero.
UTMZN	UTM Zone. For the UTM version of the transverse Mercator, you can specify a zone number instead of a central meridian. Used in the UTM projection group.
HSNS	North/South Hemisphere. Use this number to indicate the hemisphere of a UTM zone. A positive number indicates the Northern Hemisphere, and a negative number indicates the Southern Hemisphere. 1.0 and -1.0 are used. Used in the UTM projection group.
GHGT	Average Geoid Height. Used in the Wisconsin State county coordinate systems to indicate the average geoid height (in meters) in the county. This value is used to adjust the geoid height of the underlying ellipsoid. Used in the projection groups Wccsl and Wccst.
AELEV	Average Elevation. Used in the Wisconsin and Minnesota State county coordinate systems to indicate the average orthometric height (elevation in system units) of the terrain in the county. This value is used to adjust the geoid height in the underlying ellipsoid. Used in projection groups Azede, Mndotl, Mndott, Wccsl, and Wccst.
POLELNG, POLELAT, OSTDPLL	Oblique Pole Longitude, Oblique Pole Latitude, Oblique Cone Standard Parallel. Used for the Krovak Oblique Conformal Conic projection. CS-Map's parameterization is different from most other packages in that it takes the latitude of the oblique pole on the ellipsoid as a parameter rather than the co-latitude of the pole on the gaussian sphere. Used in projection groups Krovak and Krvk95.
STDCIR	Standard Circle Latitude. Polar aspects of azimuthal projections, particularly the Stereographic variation, uses the concept of a standard circle, which is analogous to the standard parallel in Mercator projections. Used in the Pstros1 projection group.

Parameter	Description
NRMPLL	Normal Parallel. The Oblique Cylindrical projection, also known as Rosamund, as used in Hungary, takes this additional parameter, which is used to compute the radius of the Gaussian sphere. Used in the Obqcyl projection group.
SKWAZM	Rectified Skew Orthomorphic, Skew. The Rectified Skew Orthomorphic, also known as RSO, projection differs from the Hotine Oblique Mercator in that the azimuth parameter is the azimuth of the central geodesic at the projection origin (rather than the projection center). Thus, the RSO takes one of these parameters rather than that described above for the Hotine Oblique Mercator. Used in the Rskewo projection group.

The following table maps the parameters of projection groups whose projection method is flat plane.

Name	YAXISAZ	CM-PLXAN	CM-PLXBN	0	STDCIR
AZMEA	x				
AZMED	x				
MSTRO		12	12		
NZLND				x	
OSTRO				x	
PSTRO				x	
PSTROSL					x

The following table maps parameters of projection groups whose projection method is conical.

N=NSTDPOLL; S=SSTDPLL; P1G=P1LNG; P1T=P1LAT; P2G=P2LNG; P2T=P2LAT; P1P2=ADP1P2; SP1=ADSP1; SP2=ADSP2; PG=POLELNG; PT=POLELAT; OP=OSTDPOLL; AE=AELEV; CM=CENTMER; GH=GHGT;

Name	N	S	0	P1G	P1T	P2G	P2T	P1P2	SP1	SP2	PG	PT	OP	AE	CM	GH
ALBER	x	x														

N=NSTDPOLL; S=SSTDPLL; P1G=P1LNG; P1T=P1LAT; P2G=P2LNG; P2T=P2LAT;
P1P2=ADP1P2; SP1=ADSP1; SP2=ADSP2; PG=POLELNG; PT=POLELAT; OP=OSTDPOLL;
AE=AELEV; CM=CENTMER; GH=GHGT;

Name	N	S	O	P1G	P1T	P2G	P2T	P1P2	SP1	SP2	PG	PT	OP	AE	CM	GH
BONNE			x													
BPCNC				x	x	x	x	x	x	x						
KROVAK											x	x	x			
KRVK95											x	x	x			
LM1SP			x													
LM2SP	x	x														
LMBLG	x	x														
LMTAN			x													
MNDOTL	x	x												x		
PLYCN															x	
WCCSL	x	x												x		x

The following table maps parameters of projection groups whose projection method is cylindrical.

CM=CENTMER; GCG=GCPLNG; GCT=GCPLAT; NP=NRMLPLL; SK=SKWAZM;

Name	CM	STDPLL	GCG	GCT	GCAZM	AELEV	NP	O	SK	UT-MZN	HSNS	GHGT
CSINI	x											
EDCYL		x										
GAUSSK	x											
HOM1XY			x	x	x							
MILLR	x											

CM=CENTMER; GCG=GCPLNG; GCT=GCPLAT; NP=NRMLPLL; SK=SKWAZM;												
Name	CM	STDPLL	GCG	GCT	GCAZM	AELEV	NP	0	SK	UT-MZN	HSNS	GHGT
MNDOTT	x					x						
MRCAT	x	x										
MRCATK	x											
OBQCYL								x				
OSTN02									x			
OSTN97									x			
ROBIN									x			
RSKEW			x	x	x							
RSKEWC			x	x	x							
RSKEWO			x	x					x			
SINUS									x			
SOTRM	x											
SWISS									x			
TRMER	x											
TRMRKRG	x											
TRMRS	x											
UTM										x	x	
WCCST	x					x						x

The following table maps parameters of projection groups whose projection method is neither flat plane, conical, or cylindrical.

Name	0	STDPLL
VDGRN	x	
WINKL		x

The parameters are classified using the categories defined in `MgCoordinateSystemLogicalType`. The category to which a parameter belongs is retrieved using the `GetParameterLogicalType` method on the `MgCoordinateSystemProjectionInformation` object. The following table presents the classification.

Category	Parameters
Angular Distance	Adp1p2 Adsp1 Adsp2
Azimuth	Gcazm Nrthrot Skwazm Yaxisaz
Complex Coefficient	Cmplxan Cmplxbn
Elevation	Aelev
Geoid Height	Ghgt
Hemisphere Selection	Hsns
Latitude	Gcp1lat Gcp2lat Gcplat Nparall Nrmlpll Nstdpll Ostdp1l P1lat P2lat Polelat Sparall Sstdpll Stdcir Stdpll
Longitude	Cntmer Eastll Estdmer Gcp1lng Gcp2lng Gcplng P1lng P2lng Polelng Westll
UTM Zone Number	Utmzn

Ellipsoid

Key ellipsoid values are the polar radius and the equatorial radius. The majority of the coordinate systems in the dictionary use one of the following 8 ellipsoid definitions. The radii have been rounded.

Ellipsoid	Num Coord Sys	Equatorial R	Polar R	(ER - PR)
Bessel	251	6377396	6356078	21318
Clrk66	441	6378206	6356584	21622
Clrk80	257	6378248	6356514	21734
GRS1980	1693	6378137	6356753	21384
Intl	870	6378388	6356911	21477
Krasov	573	6378245	6356863	21382
WGS72	248	6378135	6356751	21384
WGS84	359	6378137	6356752	21385

The following code retrieves information about the ellipsoids.

```
MgCoordinateSystemEnum ellipsoidEnum = ellipsoidDict.GetEnum();
MgStringCollection ellipsoidNames = ellipsoidEnum.NextName(ellipsoidDict.GetSize());
for (int i = 0; i < ellipsoidNames.GetCount(); i++)
{
    ellipsoidName = ellipsoidNames.GetItem(i);
    MgCoordinateSystemEllipsoid ellipsoid = ellipsoidDict.GetEllipsoid(ellipsoidName);
    double polarRadius = ellipsoid.GetPolarRadius();
    double equatorialRadius = ellipsoid.GetEquatorialRadius();
}
```

Datum

The key datum values are the ellipsoid name and one or more sets of geodetic transformation values. The geodetic transformation values are obtained from

the `MgCoordinateSystemGeodeticTransformation` class, specifically the return values of methods `GetBursaWolfeTransformBwScale`, `GetBursaWolfeTransformRotationX`, `GetBursaWolfeTransformRotationY`, `GetBursaWolfeTransformRotationZ`, `GetBursaWolfeTransformationMethod`, `GetOffsetX`, `GetOffsetY`, and `GetOffsetZ`.

The following table maps the higher runner ellipsoid datum combinations to the number of coordinate systems using that combination. The `GetDatum` and `GetEllipsoid` methods on the `MgCoordinateSystem` class is used to generate the data in this table.

Ellipsoid	Datum	Number of coordinate systems
GRS1980	NAD83	738
GRS1980	HPGN	509
KRASOV	PULKOVO	358
CLRK66	NAD27	321
WGS84	WGS84	265
INTNL	nameless	142
WGS72	WGS72	123
WGS72	WGS72-TBE	122

The follow code gets the datum names.

```
MgCoordinateSystemEnum datumEnum = datumDict.GetEnum();
MgStringCollection datumNames = datumEnum.NextName(datumDict.GetSize());
for(int i = 0; i < datumNames.GetCount(); i++)
{
    string datumName = datumNames.GetItem(i);
}
```

Coordinate System

If the coordinate system is a geographic one, the key values are the datum name, ellipsoid name, the origin longitude and latitude, the scale reduction,

the unit scale, the minimum longitude and latitude, and the maximum longitude and latitude. If the coordinate system is a projection, the key values include those of the associated geographic coordinate system and the projection parameters, the X and Y offsets, the map scale, the zero X and Y, the minimum X and Y, the maximum X and Y, and the quadrant.

The `MgCoordinateSystem` class has methods for doing the following:

- get and set properties including the associated ellipsoid and datum definitions
- if a projection, convert planar coordinates to and from longitude and latitude. The planar and geographic coordinates are within the region covered by the coordinate system, and the conversion is governed by the parameters of the geographic coordinate system associated with the cartographic coordinate system.
- convert system units to and from meters
- if a projection, get the convergence angle for a coordinate
- if a projection, get scale values for a coordinate
- return a coordinate given a start coordinate, an azimuth, and a distance
- if a projection, measure euclidean (linear) distance between two coordinates
- measure great circle distance between two geographic coordinates
- measure the azimuth between 2 coordinates. The value is positive if the direction of the vector is headed east and negative if the direction is headed west.

Methods Pertinent to a Cartographic Coordinate System

The following code creates the definition instance for the “CANADA-ALBERS” coordinate system. This coordinate system is secant conical projection. This coordinate system definition is used for all of the cartographic coordinate system example method calls.

```
MgCoordinateSystem cs = csDict.GetCoordinateSystem("CANADA-ALBERS");
```

The following code gets properties of the coordinate system, which are considered when determining whether the coordinate system is the “same” as another coordinate system.

```

string datum = cs.Datum; // WGS84
string ellipsoid = cs.Ellipsoid; // WGS84
string projection = cs.Projection; // AE
int projectionParamCount = cs.GetProjectionParameterCount(); // 2
// the first one is Northern Standard Parallel
// the second one is Southern Standard Parallel
// the cone's surface is beneath the ellipsoid's surface between
the 50 and 60 degrees North
double projectionParameter = 0; // 50 and 60
for (int i = 1; i <= projectionParamCount; i++)
{
    projectionParameter = cs.GetProjectionParameter(i);
}
string units = cs.Units; // METER
double originLongitude = cs.GetOriginLongitude(); // -100 (100
degrees West)
double originLatitude = cs.GetOriginLatitude(); //55 (55 degrees
North)
double offsetX = cs.GetOffsetX(); // 0 (false Easting)
double offsetY = cs.GetOffsetY(); // 0 (false Northing)
double scaleReduction = cs.GetScaleReduction(); // 1
double mapScale = cs.GetMapScale(); // 1
double unitScale = cs.UnitScale; // 1
double zeroX = cs.GetZeroX(); // 0
double zeroY = cs.GetZeroY(); // 0
double lonMin = cs.GetLonMin(); // -145
double latMin = cs.GetLatMin(); // 40
double lonMax = cs.GetLonMax(); // -55
double latMax = cs.GetLatMax(); // 70
double minX = cs.GetMinX(); // -3524946.41524901
double minY = cs.GetMinY(); // -1632615.063261
double maxX = cs.GetMaxX(); // 3524946.41524901
double maxY = cs.GetMaxY(); // 1698580.12154214
// 1 means that X increases to the east and Y increases to the
north.
short quadrant = cs.GetQuadrant();

```

The following code gets other properties of the coordinate system.

```

string code = cs.GetCode(); // "CANADA-ALBERS"
bool isLegalCode = cs.IsLegalCode(code); // true
string group = cs.GetGroup(); // WORLD
bool isLegalGroup = cs.IsLegalGroup(group); // true
string description = cs.GetDescription(); // "Albers Equal Area
for Canada, Meter"
bool isLegalDescription = cs.IsLegalDescription(description); //
true
short age = cs.GetAge(); // -1
string countryOrState = cs.GetCountryOrState(); // ""
bool isLegalCountryOrState = cs.IsLegalCountryOrState(countryOr
State); // true
string location = cs.GetLocation(); // ""
bool isLegalLocation = cs.IsLegalLocation(location); // true
// Datum Description is "World Geodetic System of 1984"
string datumDescription = cs.GetDatumDescription();
// Ellipsoid Description is "World Geodetic System of 1984, GEM
10C"
string ellipsoidDescription = cs.GetEllipsoidDescription();
string projection = cs.GetProjection(); // AE
// Projection Description is "Albers Equal Area Conic Projection"
string projectionDescription = cs.GetProjectionDescription();
string source = cs.GetSource(); // Autodesk
bool isLegalSource = cs.IsLegalSource(source); // true
bool isEncrypted = cs.IsEncrypted(); // true
bool isGeodetic = cs.IsGeodetic(); // true
bool isProtected = cs.IsProtected(); // true
bool isValid = cs.IsValid(); // true
bool isValidXY = cs.IsValidXY(-116426.504475028, -
194031.395349903); // true

```

The following code converts a planar coordinate in system units to a geographic coordinate in degrees and then converts the geographic coordinate back to its planar equivalent.

```

// toLonLat is (-101.75, 53.25)
MgCoordinate toLonLat = cs.ConvertToLonLat(-116426.504475028, -
194031.395349903);
// fromLonLat is (-116426.504475028, -194031.395349903)
MgCoordinate fromLonLat = cs.ConvertFromLonLat(toLonLat);

```

The following code gets the convergence angles for 2 coordinates. The origin longitude is -100. One coordinate is West of the origin longitude. Its convergence angle is negative. The other coordinate is East of the origin

longitude. Its convergence angle is positive. In both cases the convergence angle is subtracted from the coordinate's longitude to get its true north.

```
// convergenceMinXMinY is -1.42809311378578
double convergenceMinXMinY = cs.GetConvergence(-101.75, 53.25)
// convergenceMaxXMaxY is 1.42809311380822
double convergenceMaxXMaxY = cs.GetConvergence(-98.25, 56.75);
```

The following code gets the scale for the same 2 coordinates that were used in the GetConvergence example.

```
// minXMinYScale is 0.996816990601911
double minXMinYScale = cs.GetScale(-101.75, 53.25);
// maxXMaxYScale is 0.996527143513552
double maxXMaxYScale = cs.GetScale(-98.25, 56.75);
```

The following code gets the longitudinal scale for the same 2 coordinates that were used in the GetConvergence example.

```
// minXMinYScaleH is 1.00319317329871
double minXMinYScaleH = cs.GetScaleH(-101.75, 53.25);
// maxXMaxYScaleH is 1.00348495924978
double maxXMaxYScaleH = cs.GetScaleH(-98.25, 56.75);
```

The following code gets the latitudinal scale for the same 2 coordinates that were used in the GetConvergence example.

```
// minXMinYScaleK is 0.996816990601911
double minXMinYScaleK = cs.GetScaleK(-101.75, 53.25);
// maxXMaxYScaleK is 0.996527143513552
double maxXMaxYScaleK = cs.GetScaleK(-98.25, 56.75);
```

The following code gets the azimuth of 2 planar coordinates twice. The order of the coordinate arguments in the first call results in a positive azimuth. The order of the coordinate arguments in the second call results in a negative azimuth.

```
// positiveAzimuth is 28.4694490801922
double positiveAzimuth = cs.GetAzimuth(-116426.504475028, -
194031.395349903, 106680.575844089, 196898.95225439);
// negativeAzimuth is -148.66188542923
double negativeAzimuth = cs.GetAzimuth(106680.575844089,
196898.95225439, -116426.504475028, -194031.395349903);
```

The following code gets the euclidean (linear) distance between the same 2 planar coordinates as used in the GetAzimuth example.

```
// euclideanDistance is 450114.769771593
double euclideanDistance = cs.MeasureEuclideanDistance(-
116426.504475028, -194031.395349903, 106680.575844089,
196898.95225439);
```

The following code uses a coordinate, an azimuth, and a distance to get a second coordinate. It does this twice by going from one coordinate to the second and then from the second coordinate back to the first. The code uses the same 2 coordinates as the GetAzimuth and MeasureEuclideanDistance examples. The first call to GetCoordinate derives the 2nd coordinate from the 1st using the azimuth from the 1st to the 2nd and the distance between them. The second call to GetCoordinate derives the 1st coordinate from 2nd using the azimuth from 2nd to the 1st and the distance between them.

```
// 2ndCoord is (106680.575844089, 196898.95225439)
MgCoordinate 2ndCoord = cs.GetCoordinate(-116426.504475028, -
194031.395349903, 28.4694490801922, 450114.769771593);
// 1stCoord is -116426.504475028, -194031.395349903
MgCoordinate 1stCoord = cs.GetCoordinate(106680.575844089,
196898.95225439, -148.66188542923, 450114.769771593);
```

Methods Pertinent to a Geographic Coordinate System

The following code creates the definition instance for the “LL84” coordinate system. This coordinate system definition is used for all of the geographic coordinate system example method calls.

```
MgCoordinateSystem cs = csDict.GetCoordinateSystem("LL84");
```

The following code gets properties of the coordinate system, which are considered when determining whether the coordinate system is the “same” as another coordinate system.

```
string datum = cs.Datum; // WGS84
string ellipsoid = cs.Ellipsoid; // WGS84
string units = cs.Units; // DEGREE
double originLongitude = cs.GetOriginLongitude(); // 0
double originLatitude = cs.GetOriginLatitude(); // 0
double unitScale = cs.UnitScale; // 111319.490793274
double lonMin = cs.GetLonMin(); // -180
double latMin = cs.GetLatMin(); // -90
double lonMax = cs.GetLonMax(); // 180
double latMax = cs.GetLatMax(); // 90
```

The following code gets other properties of the coordinate system.

```

string code = cs.GetCode(); // "LL84"
bool isLegalCode = cs.IsLegalCode(code); // true
string group = cs.GetGroup(); // LL
bool isLegalGroup = cs.IsLegalGroup(group); // true
string description = cs.GetDescription(); // "WGS84 datum, Latitude-Longitude; Degrees"
bool isLegalDescription = cs.IsLegalDescription(description); // true
short age = cs.GetAge(); // -1
string countryOrState = cs.GetCountryOrState(); // ""
bool isLegalCountryOrState = cs.IsLegalCountryOrState(countryOrState); // true
string location = cs.GetLocation(); // ""
bool isLegalLocation = cs.IsLegalLocation(location); // true
// Datum Description is "World Geodetic System of 1984"
string datumDescription = cs.GetDatumDescription();
// Ellipsoid Description is "World Geodetic System of 1984, GEM 10C"
string ellipsoidDescription = cs.GetEllipsoidDescription();
string projection = cs.GetProjection(); // LL
// Projection Description is "Projection Description Null Projection, produces/processes Latitude & Longitude"
string projectionDescription = cs.GetProjectionDescription();
string source = cs.GetSource(); // "Mentor Software"
bool isLegalSource = cs.IsLegalSource(source); // true
bool isEncrypted = cs.IsEncrypted(); // true
bool isGeodetic = cs.IsGeodetic(); // true
bool isProtected = cs.IsProtected(); // true
bool isValid = cs.IsValid(); // true
bool isValidXY = cs.IsValidXY(180, 90); // true

```

The following code gets the azimuth of 2 geographic coordinates twice. The order of the coordinate arguments in the first call results in a positive azimuth. The order of the coordinate arguments in the second call results in a negative azimuth.

```

// positiveAzimuth is 169.839538518442
double positiveAzimuth = cs.GetAzimuth(170, 80, -170, -80);
// negativeAzimuth is -10.1604614815584
double negativeAzimuth = cs.GetAzimuth(-170, -80, 170, 80);

```

The following code gets the great circle distance between the same 2 geographic coordinates as used in the GetAzimuth example.


```
// greatCircleDistance is 159.941320107699
double greatCircleDistance = cs.MeasureGreatCircleDistance(-170,
-80, 170, 80);
```

The following code uses a coordinate, an azimuth, and a distance to get a second coordinate. It does this twice by going from one coordinate to the second and then from the second coordinate back to the first. The code uses the same 2 coordinates as the GetAzimuth and MeasureEuclideanDistance examples. The first call to GetCoordinate derives the 2nd coordinate from the 1st using the azimuth from the 1st to the 2nd and the distance between them. The second call to GetCoordinate derives the 1st coordinate from 2nd using the azimuth from 2nd to the 1st and the distance between them.

```
// 2ndCoord is (170, 79.9999999999998)
MgCoordinate 2ndCoord = cs.GetCoordinate(-170, -80, -
10.1604614815584, 159.941320107699);
// 1stCoord is -170, -79.9999999999998
MgCoordinate 1stCoord = cs.GetCoordinate(170, 80, 169.839538518442,
159.941320107699);
```

Coordinate Transformation

Coordinate transformation is converting a coordinate value in one coordinate system into its equivalent in another coordinate system. The first consideration is whether the source coordinate has an equivalent in the target coordinate system. A UTM10 coordinate cannot be transformed into a UTM29 coordinate. The key determinants in the transformation are differences between the ellipsoid and datum definitions of the source and target coordinate systems.

The key ellipsoid values are polar and equatorial radii. The values for the ellipsoids presented in this document vary between 6377396 and 6378388 meters for the equatorial radius and between 6356078 and 6356752 meters for the polar radius.

The key datum values are those obtainable from the `MgCoordinateSystemGeodeticTransformation` object contained in the `MgCoordinateSystemDatum` object. Of particular interest is the transformation method. The types of the methods generally available are defined in the `MgCoordinateSystemGeodeticTransformationMethod` class.

The following code illustrates transformations between 2 geographic coordinate systems, between a geographic coordinate system and a cartographic coordinate system, and between 2 cartographic coordinate systems. The codes for the two geographic coordinate systems are “LL72” and “LL84”. The codes for the two cartographic coordinate systems are “WGS72.UTM-10N” and

“UTM84-10N”. The “LL72” and “WGS72.UTM-10N” coordinate systems use the “WGS72” ellipsoid definition and the “WGS72” datum definition. The “LL84” and “UTM84-10N”. coordinate systems use the “WGS84” ellipsoid definition and the “WGS84” datum definition. As can be seen in the table showing the equatorial and polar radii of the “WGS72” and “WGS84” ellipsoid definitions, the differences between the two sets of values are negligible. The geodetic transformation method of the “WGS72” datum is “WGS72”. The geodetic transformation method of the “WGS84” datum is “WGS84”.

```
MgCoordinateSystem LL72Cs = csDict.GetCoordinateSystem("LL72");
MgCoordinateSystem LL84Cs = csDict.GetCoordinateSystem("LL84");
MgCoordinateSystem utm10n72Cs = csDict.GetCoordinateSystem("WGS72.UTM-10N");
MgCoordinateSystem utm10n84Cs = csDict.GetCoordinateSystem("UTM84-10N");
// geographic to geographic
MgCoordinateSystemTransform LL72ToLL84 = coordSysFactory.GetTransform(LL72Cs, LL84Cs);
// LL72ToLL84Coord is (-124.749846111111, 39.7500328381061)
MgCoordinate LL72ToLL84Coord = LL72ToLL84.Transform(-124.75, 39.75);
// geographic to projected
MgCoordinateSystemTransform LL72ToUtm10n84 = coordSysFactory.GetTransform(LL72Cs, utm10n84Cs);
// the transform result LL72ToUtm10n84Coord is in meters (350086.148700075, 4401477.98122743)
MgCoordinate LL72ToUtm10n84Coord = LL72ToUtm10n84.Transform(-124.75, 39.75);
// convert the result to degrees (-124.749846111114, 39.7500328381018)
MgCoordinate utm10n84LonLatCoord = utm10n84Cs.ConvertToLonLat(350086.148700075, 4401477.98122743);
// projected to projected
MgCoordinateSystemTransform utm10n72ToUtm10n84 = coordSysFactory.GetTransform(utm10n72Cs, utm10n84Cs);
// convert (-124.75, 39.75) to "WGS72.UTM-10N" system units (meters)
// which is (350072.941601698, 4401473.42938798)
MgCoordinate utm10n72XYCoord = utm10n72Cs.ConvertFromLonLat(-124.75, 39.75);
// the result is (350086.148699852, 4401477.98122696)
MgCoordinate utm10n72ToUtm10n84Coord = utm10n72ToUtm10n84.Transform(350072.941601698, 4401473.42938798);
```

Quadrant

Quadrant is CSMaP's way of handling the rare case where the X-axis does not increase to the East and the Y-axis does not increase to the North. The following table maps the quadrant values to the directions in which the X and Y values increase.

Quad-rant	X increases to the	Y increases to the
0, 1	East	North
2	West	North
3	West	South
4	East	South
-1	North	East
-2	North	West
-3	South	West
-4	South	East

Secant Projection

The purpose of a secant projection is to distribute the scale distortion across the geographic region covered by the coordinate system.

The **scale reduction factor** is one of the means of defining a secant projection. For example, the scale reduction factor used in UTM coordinate systems indicates that the projection surface is shrunk to the degree necessary to produce a grid scale factor of 0.9996 at the central meridian which would otherwise have a grid scale factor of unity (1.0). A factor of 0.9996 indicates a secant projection whereas a scale factor of 1.0 would indicate a tangential projection.

See the projection group property tables in the [Properties](#) (page 69) section for the list of azimuthal, cylindrical, and conical projection groups that have scale reduction factors.

See the definition of scale reduction factor at http://www.onwordpress.delmar.cengage.com/resources/inside_GeoMedia/Module3c.aspx (browsed Jan 2009). Use `MgCoordinateSystemProjectionInformation` method `IsUsingScaleReduction` to determine whether a coordinate system has a scale reduction factor the `GetScaleReduction` method on the `MgCoordinateSystem` instance to get the scale reduction value.

The presence of the two projection parameters, Northern Standard Parallel and Southern Standard Parallel, indicates a secant projection. Projections in the Alber, Lm2sp, Lmblg, Lmbrtaf, Mndotl, and Wccsl projection groups have these parameters.

Caspian Sea Mercator Projection

The Caspian Sea is a secant Mercator projection. The projection surface is “inside” the earth between 42N and 42S. The projection surface is “outside” the earth at all other latitudes.

The standard parallel specification of 42N specifies that at that latitude, the grid scale distortion is zero. At latitudes north of 42N, and latitudes south of 42S, the grid scale factor will be greater than 1.0. At latitudes between 42N and 42S, the grid scale factor will be less than one. The Caspian Sea ranges from (a guess) 37N to 47N. That is the intended region for the coordinate system.

Format Conversion

The underlying CS-Map software references the coordinate systems in the dictionary using its own code values. Use the `MgCoordinateSystemFormatConverter` class to find the equivalent EPSG, Oracle, ESRI, OGC, GeoTiff, or GeoTools definitions.

The following code converts a CS-Map coordinate system code to an EPSG coordinate system code and then converts the EPSG code back to a CS-Map code. The first argument is an integer constant identifying the type of the code being converted. The integer constants defining the two types: CS-Map and EPSG are defined in `MgCoordinateSystemCodeFormat`. The second argument is the code to convert. The third argument is an `MgCoordinateSystemCodeFormat` constant identifying the type of the result of the conversion.

```
// the result epsgCode is "2001"
string epsgCode = formatConverter.CodeToCode(MgCoordinateSystem
CodeFormat.Mentor, "Antigua43.BWIGrid", MgCoordinateSystemCode
Format.Epsg);
// the reverse conversion
string csMapCode = formatConverter.CodeToCode(MgCoordinateSystem
CodeFormat.Epsg, "2001", MgCoordinateSystemCodeFormat.Mentor);
```

The following code converts a CS-Map coordinate system code to an Oracle WKT definition. The first argument is an `MgCoordinateSystemCodeFormat` constant identifying the type of the code to convert. The second argument is the code value. The third argument is an integer constant identifying the WKT type of the result of the conversion. The WKT types are defined in `MgCoordinateSystemWktFlavor`.

```
// the result of the conversion is "GEO
GCS["LL84", DATUM["WGS84", SPHER
OID["WGS84", 6378137.000, 298.25722293]], PRIMEM["Green
wich", 0], UNIT["Degree", 0.01745329251994]]"
string oracleWktFromMentorCode = formatConverter.CodeToWkt(MgCo
ordinateSystemCodeFormat.Mentor, "LL84", MgCoordinateSystemWktFla
vor.Oracle);
```

The following code converts a WKT string to a coordinate system code. The operation reverses the preceding conversion.

```
string wkt = "GEOGCS["LL84", DATUM["WGS84", SPHER
OID["WGS84", 6378137.000, 298.25722293]], PRIMEM["Green
wich", 0], UNIT["Degree", 0.01745329251994]]"
string codeFromFormatConverter = formatConverter.WktToCode(MgCo
ordinateSystemWktFlavor.Oracle, wkt, MgCoordinateSystemCode
Format.Mentor);
```

This class has two other methods of interest. The `DefinitionToCode` takes an `MgCoordinateSystem` instance as its first argument and an `MgCoordinateSystemCodeFormat` constant as its second argument. The `MgCoordinateSystem` class a method for getting the CS-Map code but not the EPSG code. The `DefinitionToWkt` takes an `MgCoordinateSystem` instance as its first argument and an `MgCoordinateSystemWktFlavor` constant as its second argument. The `MgCoordinateSystem` does not have a method for getting its WKT.

Definition Comparison

The `MgCoordinateSystemMathComparator` offers methods for comparing coordinate system, ellipsoid, and datum definitions for mathematical equivalence. The key values specified in the discussion of the coordinate system, ellipsoid and datum definitions are those that are considered in the calculation of mathematical equivalence.

The following code compares two coordinate systems. This comparison implicitly includes ellipsoid and datum comparisons.

```
MgCoordinateSystem can83Cs = csDict.GetCoordinateSystem("CAN83-11");
MgCoordinateSystem canQCs = csDict.GetCoordinateSystem("CANQ-M11");
bool projCsSame = mathComparator.SameCoordinateSystem(can83Cs,
canQCs);
```

The following code compares two ellipsoid definitions.

```
MgCoordinateSystemEllipsoid clrk80Ellipsoid = ellipsoidDict.GetEllipsoid("CLRK80");
MgCoordinateSystemEllipsoid clrkRgsEllipsoid = ellipsoidDict.GetEllipsoid("CLRK-RGS");
bool ellipsoidsSame = mathComparator.SameEllipsoid(clrk80Ellipsoid,
clrkRgsEllipsoid);
```

The following code compares two datum definitions.

```
MgCoordinateSystemDatum wgs84Datum = datumDict.GetDatum("WGS84");
MgCoordinateSystemDatum dgn95Datum = datumDict.GetDatum("DGN95");
bool datumsSame = mathComparator.SameDatum(wgs84Datum, dgn95Datum);
```

Index

= 31
> 31
>= 31

A

AcMapApiMgd assembly 3
AcMapFeatureService.SelectFeatures() 29
AcMapLayer objects 41
AcMapLayer.DiscardFeatureChanges() 36
AcMapLayer.SaveFeatureChanges() 36
AcMapLayer.SelectFeatures() 33
AcMapLayer.SetEditSetMode() 36
AcMapLayer.UpdateFeatures() 36
AcMapMap.GetCurrentMap() 17
AcMapMap.GetFeatureSelection() 28
Add() method of MgLayerCollection 26
Add() method of
 MgPropertyCollection 35
adding features 34
AGF geometry format 29
ApplicationDefinition resource type 6,
 10
arbitrary X-Y coordinates 59
Autodesk MapGuide Studio 9

B

basic selection filters 30–31
buffer, creating 61
buffers 60

C

ConfigurationDocument element, in
 feature source definition 19
CONTAINS 32
coordinate reference system 59
coordinate systems 59
 transforming between 60
COVEREDBY 32

creating geometry from feature 33
creating SDF files 36
CROSSES 32
CRS 59

D

deleting features 34
direct update 35
DiscardFeatureChanges() method of
 AcMapLayer 36
DISJOINT 32
distance, measuring 60
Document Object Model 9
DOM 9
drawing order 42
DrawingSource resource type 6, 10

E

edit sets 35
encoding, UTF-8 for Resource Service 19
EQUALS 32

F

FDO 17
 raster provider 19
feature classes 17
 creating layers from 26
Feature Data Objects 17
Feature Service 17
feature source 17
 XML representation 18
feature source schema 17
FeatureName element in LayerDefinition
 schema 26
features
 adding 34
 deleting 34
 getting geometry from 33

- selecting 28
- updating 34–35
- FeatureSource resource type 5
- FeatureSource.xsd 18
- Folder resource type 6, 10

G

- GenerateFilter() method of
 - MgSelectionBase 29
- geographic coordinates 59
- geometry
 - comparing spatial relationships 58
- geometry formats 29
- geometry types 57
- GEOMFROMTEXT() 32
- GetClassDefinition() method of
 - MgFeatureService 28
- GetClasses() method of
 - MgFeatureService 26
- GetFeatureSelection() method of
 - AcMapMap 28
- GetGeometry() method of
 - MgFeatureReader 33
- GetIdentityProperties() method of
 - MgClassDefinition 28
- GetLayers() method of
 - MgSelectionBase 28
- GetSchemas() method of
 - MgFeatureService 26
- great circle calculation, in measuring distance 60

I

- INSIDE 32
- INTERSECTS 32

L

- latitude/longitude coordinates 59
- layer groups 42
- layer name 42
- layer properties 41
- layer visibility 43

- layer visibility, and layer groups 42
- LayerDefinition resource type 5
- LayerDefinition.xsd schema 26
- layerdefinitionfactory.php 9
- layers
 - creating from feature classes 26
 - creating new with SDF file 36
- Library repository 5
- LIKE 31
- LoadProcedure resource type 6, 10

M

- Map 3D
 - and MapGuide, differences 10
- map layer, and feature source 17
- Map resource type 6, 10
- MapDefinition resource type 6, 10
- MapGuide
 - and Map 3D, differences 10
- MapGuide Studio 9
- measuring distance 60
- MgAgfReaderWriter 30
- MgAgfReaderWriter.Read() 30, 33
- MgAgfReaderWriter.Write() 30
- MgByteReader 8
- MgByteSink 8
- MgByteSource 8
- MgClassDefinition.GetIdentityProperties() 28
- MgCoordinateSystem 59
- MgCoordinateSystemTransform 60
- MgCurvePolygon 57
- MgCurveString 57
- MgFeatureCommandCollection 34
- MgFeatureQueryOptions 29–30
- MgFeatureQueryOptions.SetFilter() 32
- MgFeatureQueryOptions.SetSpatialFilter() 32
- MgFeatureReader 29
- MgFeatureReader.GetGeometry() 33
- MgFeatureReader.ReadNext() 29, 33
- MgFeatureService.GetClassDefinition() 28
- MgFeatureService.GetClasses() 26
- MgFeatureService.GetSchemas() 26
- MgFeatureService.SelectFeatures() 33
- MgFeatureService.UpdateFeatures() 36

- MgGeometry 29, 57
 - creating from feature 33
- MgInsertFeatures 34
- MgLayerCollection
 - GetItem() 42
- MgLayerCollection object 42
- MgLayerCollection.Add() 26
- MgLineString 57
- MgMap
 - GetLayers() 42
- MgMultiCurvePolygon 58
- MgMultiCurveString 58
- MgMultiGeometry 58
- MgMultiLineString 58
- MgMultiPoint 58
- MgMultiPolygon 58
- MgPoint 57
- MgPolygon 57
- MgPropertyCollection.Add() 35
- MgResourceService.SetResource() 8, 18
- MgResourceType 5
- MgSelection 30
- MgSelectionBase 28
- MgSelectionBase.GenerateFilter() 29
- MgSelectionBase.GetLayers() 28
- MgWktReaderWriter 30
- MgWktReaderWriter.Read() 30
- MgWktReaderWriter.Write() 30

O

OVERLAPS 32

P

path, in repository 18
 PrintLayout resource type 6, 10
 projected coordinates 59
 properties, of layers 41

R

raster provider 19
 Read() method of
 MgAgfReaderWriter 30, 33

Read() method of
 MgWktReaderWriter 30
 ReadNext() method of
 MgFeatureReader 29, 33
 repository 5
 repository name 6
 repository path 6, 18
 repository type 6
 resource data 7
 resource dependencies 7
 resource headers, in Map 3D 10
 resource name 6
 resource paths, in DWG file 10
 resource schemas 8
 resource service 7
 Resource Service 5
 UTF-8 encoding 19
 resource type 7
 resource types 5
 resources 5

S

SaveFeatureChanges() method of
 AcMapLayer 36
 schema
 feature source 17
 schemas, resource 8
 SDF files
 creating 36
 SelectFeatures() method of
 AcMapFeatureService 29
 SelectFeatures() method of
 AcMapLayer 33
 SelectFeatures() method of
 MgFeatureService 33
 selecting
 with the Platform API 30
 selecting features 28
 selection filters 30
 basic 31
 spatial 32
 Selection resource type 6, 10
 Session repositories 5
 Session repositories, in Map 3D 10

- SetEditSetMode() method of
 - AcMapLayer 36
- SetFilter() method of
 - MgFeatureQueryOptions 32
- SetResource() method, in
 - MgResourceService 18
- SetSpatialFilter() method of
 - MgFeatureQueryOptions 32
- spatial filters 32
- spatial operators 32
- spatial reference system 59
- spatial relationships, between geometry
 - objects 58
- spatial selection filters 30
- SRS 59
- Studio 9
- SymbolDefinition resource type 6
- SymbolLibrary resource type 6, 10

T

- templates, for XML resources 9
- TOUCHES 32
- transforming coordinate systems 60

U

- update modes 35

- UpdateFeatures() method of
 - AcMapLayer 36
- UpdateFeatures() method of
 - MgFeatureService 36
- updating features 34
- UTF-8 encoding, for Resource Service 19

V

- visibility
 - rules 43

W

- WebLayout resource type 6, 10
- Well Known Text 29
- WITHIN 32
- WKT (Well Known Text) 29
- Write() method of
 - MgAgfReaderWriter 30
- Write() method of
 - MgWktReaderWriter 30

X

- XML templates 9
- XML, for feature source 18
- xsd.exe 8