effective objects

ProLat™

Coordinate Conversion Toolkit

Includes:

ProLat for .NET pure managed library

User's Manual

Version 5.24

Updated 2020 December 30

License Agreement

Please see the file ProLatLicense.pdf for the license agreement for this software. If you do not agree with the license agreement, please do not use this software product.

Table of Contents

Introduction

ProLat provides a complete managed coordinate conversion library for .Net, Win32/64, and Android environments. The coordinate conversion results are carefully tested against government sources so you are ensured of the most accurate results possible.

- Pure managed code for C# .Net, VB .Net, C++/CLI. Works with Windows 7, 8, RT, and Window Phone 7 and 8
- Native code for Win32 and Android. Works with VB6, Visual Studio, and Android Java.
- Easy to define coordinate systems
- One step translation between any of the supported coordinate systems
- Datum to datum conversions (Over 300 datums including HARN)
- UTM, State Plane, Geocentric (ECEF/XYZ), and many more projections
- Custom coordinate systems and projections with unlimited parameters
- Great Earth distance and direction calculations
- Scale and angle of convergence calculations for all projections
- Well documented with examples in this manual

Getting Started

- Install the software by uncompressing the zip file into an empty folder such as C:\ProLat, or an empty user folder.
- Try the examples located in the folder for the desired platform.
- Add the functions to your program or use an example as a starting point
- Test carefully
- Ensure that the functions can find the supporting files when distributing the software

It is suggested to read the section on "Defining Coordinate Systems" first. Next, jump to the chapters for your specific platform for examples and function documentation.

The different platforms have very similar functionality, but the syntax and function names may vary. The general coordinate conversion information applies to all platforms.

Defining Coordinate Systems

A coordinate system in ProLat is a set of parameters that defines the essentials needed to know about a set of coordinate data points in order to convert them to another coordinate system. Generally it is easy to define a coordinate system, and the beauty is that ProLat can figure out exactly what is needed to convert between any two given coordinate systems.

In ProLat there are two ways to define a coordinate system.

1. **Geodesy coordinate system database** with four parameters: Group, System, Datum, and Units. This method offers easy selection and is self descriptive. It ensures that all elements of a coordinate system are specified. It is used by professionals in many fields.

   An example: `UTM, UTM-15N, WGS84, METERS`

2. **Proj.4 parameter definition** Use this method for existing Proj.4 definitions, or for creating custom coordinate systems. It is flexible, although it may be harder to learn.

   An example: `proj=utm zone=15 datum=WGS84`

The following sections provide more details of defining coordinate systems.

Geodesy Coordinate System Database

ProLat introduces a professional Geodesy database of commonly used coordinate systems. It is likely to contain the desired coordinate system such as UTM, State Plane, etc. It also has a comprehensive selection of datums and units. It may be customized to include new coordinate systems. For info, see the section titled "Customizing the Coordinate System Database."

To specify a coordinate system, four simple strings are needed: 1. **Group** (UTM etc.); 2. **System** within the group (UTM-17N etc.); 3. **Datum** (WGS84 etc.); and 4. **Units** (METERS etc.).

See the Examples folder for sample calculators that show how to load the group and system lists.

Geodesy functions to define a coordinate system

By using the group, system, datum, and units parameters, a complete coordinate system may be defined. ProLat can convert any valid coordinate system to any other coordinate system with the function, CoordSys.Transform().

ProLat for .Net

```
C#:
CoordSys latlong = CoordSys.GetCS("LAT_LONG", "LAT-LONG", "WGS84", "METERS");
CoordSys utm17 = CoordSys.GetCS("UTM", "UTM-17N", "WGS84", "METERS");

VB.Net:
Dim latlong as CoordSys = CoordSys.GetCS("LAT_LONG", "LAT-LONG", "WGS84", "METERS")
Dim utm17 as CoordSys = CoordSys.GetCS("UTM", "UTM-17N", "WGS84", "METERS")

C++/CLI:
CoordSys^ latlong = CoordSys::GetCS("LAT_LONG", "LAT-LONG", "WGS84", "METERS");
CoordSys^ utm17 = CoordSys::GetCS("UTM", "UTM-17N", "WGS84", "METERS");
```

ProLat for Win32

```
C++ / VB6 / etc.
LL = ProLatDefineGeodesy("LAT_LONG", "LAT-LONG", "WGS84", "METERS");
```

ProLat for Android

```
Java
CoordSys latlong = CoordSys.GetCS("LAT_LONG", "LAT-LONG", "WGS84", "METERS");
```

Groups

A complete list of groups can be found with the function CoordSys.**GetGroups**() or .

LAT_LONG	Geodetic Latitude / Longitude
UTM	Universal Transverse Mercator
US_SPC27	US State Plane 1927
US_SPC83	US State Plane 1983
ECEF	XYZ Cartesian ECEF
Argentine_Coordinate_Systems	Argentine Coordinate Systems
AUSTRALIA	Australian AMG Coordinate Systems
Australian_ISG	Australian ISG Coordinate Systems
Australian_MGA_Coordinate_System	Australian MGA Coordinate System
Austrian_Coordinate_Systems	Austrian Coordinate Systems
Bahrainian_Coordinate_Systems	Bahrainian Coordinate Systems
BELGIAN	Belgian Coordinate Systems
BORNEO	Borneo RSO Grids
BRITISH	British Coordinate Systems
Chad_Coordinate_Systems	Chad Coordinate Systems
COLOMBIA	Colombia Coordinate Systems
Egyptian_Coordinate_Systems	Egyptian Coordinate Systems
FRANCE	French Coordinate Systems
Ghanaian_Coordinate_Systems	Ghanaian Coordinate Systems
GK_PULKOVO	Gauss Kruger (Pulkovo 1942) Coordinate Systems
GK3TM	Gauss Kruger 3TM Coordinate Systems
GK6TM	Gauss Kruger 6TM Coordinate Systems
Hungarian_Coordinate_Systems	Hungarian Coordinate Systems
Indian_Coordinate_Systems	Indian Coordinate Systems
Iraq_Coordinate_Systems	Iraq Coordinate Systems
Krovak	Krovak Oblique Conic Conformal
Libyan_Coordinate_Systems	Libyal Coordinate Syatems
Malaysian_RSO_Grids	Malaysian RSO Grids
New_Zealand_Transverse_Mercator	New Zealand TM Coord. Sys.
NEWZEALAND	New Zealand Coordinate Systems
Nigerian_Systems	Nigerian Systems
NZMG	New Zealand Map Grid
Peruvian_Systems	Peruvian Systems
QATAR	Qatar Coordinate Systems
QMTM	Quebec Modified TM Coordinate Systems
QMTM_NAD27	Quebec Modified TM NAD27
ROMANIAN	Romanian Coordinate Systems
Netherlands_Systems	Netherlands Coordinate Systems
Swiss_Coordinate_Systems	Swiss_Coordinate_Systems
BLM	GOM
MICHIGAN_OLD	Michigan Older NAD27 TM

Systems

For each group there is a list of systems in the group. Use the function CoordSys.**GetSystems**(group) to get a list of systems within a group.

For example the group, UTM, has 60 zones and the State Plane group, US_SPC83, has a list of states and zone systems. Here are a few samples of common groups and systems.

```
Group:   LAT_LONG       (Standard latitude/longitude degrees)
Systems: LAT-LONG

Group:   UTM            (Universal Transverse Mercator)
Systems: UTM-01N
         UTM-02N
         …
         UTM-01S
         UTM-02S
         …

Group:   US_SPC83       (State Plane Coordinate System NAD83)
Systems:
AK83-1    Alaska Zone 1                  KS83-N    Kansas Northern Zone
AK83-10   Alaska Zone 10                 KS83-S    Kansas Southern Zone
AK83-2    Alaska Zone 2                  KY83      Kentucky Single Zone
AK83-3    Alaska Zone 3                  KY83-N    Kentucky Northern Zone
AK83-4    Alaska Zone 4                  KY83-S    Kentucky Southern Zone
AK83-5    Alaska Zone 5                  LA83-N    Louisiana Northern Zone
AK83-6    Alaska Zone 6                  LA83-O    Louisiana Offshore Zone
AK83-7    Alaska Zone 7                  LA83-S    Louisiana Southern Zone
AK83-8    Alaska Zone 8                  MA83-I    Massachusetts Island
AK83-9    Alaska Zone 9                  MA83-M    Massachusetts Mainland
AL83-E    Alabama Eastern Zone           MD83      Maryland
AL83-W    Alabama Western Zone           ME83-E    Maine Eastern Zone
AR83-N    Arkansas Northern Zone         ME83-W    Maine Western Zone
AR83-S    Arkansas Southern Zone         MI83-C    Michigan Central Zone
AZ83-C    Arizona Central Zone           MI83-N    Michigan Northern Zone
AZ83-E    Arizona Eastern Zone           MI83-S    Michigan Southern Zone
AZ83-W    Arizona Western Zone           MN83-C    Minnesota Central Zone
CA83-1    California Zone I              MN83-N    Minnesota Northern Zone
CA83-2    California Zone II             MN83-S    Minnesota South Zone
CA83-3    California Zone III            MO83-C    Missouri Central Zone
CA83-4    California Zone IV             MO83-E    Missouri Eastern Zone
CA83-5    California Zone V              MO83-W    Missouri Western Zone
CA83-6    California Zone VI             MS83-E    Mississippi Eastern
CO83-C    Colorado Central Zone          MS83-W    Mississippi Western Zone
CO83-N    Colorado Northern Zone         MT83      Montana
CO83-S    Colorado Southern Zone         NC83      North Carolina
CT83      Connecticut                    ND83-N    North Dakota Northern Zone
DE83      Delaware                       ND83-S    North Dakota Southern Zone
FL83-E    Florida Eastern Zone           NE83      Nebraska
FL83-N    Florida Northern Zone          NH83      New Hampshire
FL83-W    Florida Western Zone           NJ83      New Jersey
GA83-E    Georgia Eastern Zone           NM83-C    New Mexico Central Zone
GA83-W    Georgia Western Zone           NM83-E    New Mexico Eastern Zone
HI83-1    Hawaii Zone 1                  NM83-W    New Mexico Western Zone
HI83-2    Hawaii Zone 2                  NV83-C    Nevada Central Zone
HI83-3    Hawaii Zone 3                  NV83-E    Nevada Eastern Zone
HI83-4    Hawaii Zone 4                  NV83-W    Nevada Western Zone
HI83-5    Hawaii Zone 5                  NY83-C    New York Central Zone
IA83-E    Indiana Eastern Zone           NY83-E    New York Eastern Zone
IA83-W    Indiana Western Zone           NY83-I    New York Long Island Zone
ID83-C    Idaho Central Zone             NY83-W    New York Western Zone
ID83-E    Idaho Eastern Zone             OH83-N    Ohio Northern Zone
ID83-W    Idaho Western Zone             OH83-S    Ohio Southern Zone
IL83-E    Illinois Eastern Zone          OK83-N    Oklahoma Northern Zone
IL83-W    Illinois Western Zone          OK83-S    Oklahoma Southern Zone
IO83-N    Iowa Northern Zone             OR83-N    Oregon Northern Zone
IO83-S    Iowa Southern Zone             OR83-S    Oregon Southern Zone
```

PA83-N	Pennsylvania Northern Zone
PA83-S	Pennsylvania Southern
PRVI83	Puerto Rico, Virgin Islnds
RI83	Rhode Island
SC83	South Carolina
SD83-N	South Dakota Northern Zone
SD83-S	South Dakota Southern Zone
TN83	Tennessee
TX83-C	Texas Central Zone
TX83-N	Texas Northern Zone
TX83-NC	Texas North Central Zone
TX83-S	Texas Southern Zone
TX83-SC	Texas South Central Zone
UT83-C	Utah Central Zone
UT83-N	Utah Northern Zone
UT83-S	Utah Southern Zone
VA83-N	Virginia Northern Zone
VA83-S	Virginia Southern Zone
VT83	Vermont
WA83-N	Washington Northern Zone
WA83-S	Washington Southern Zone
WI83-C	Wisconsin Central Zone
WI83-N	Wisconsin Northern Zone
WI83-S	Wisconsin Southern Zone
WV83-N	West Virginia Northern
WV83-S	West Virginia Southern
WY83-E	Wyoming Eastern Zone
WY83-EC	Wyoming East Central Zone
WY83-W	Wyoming Western Zone
WY83-WC	Wyoming West Central Zone

For other groups use CoordSys.**GetSystems**(group) to get a list of systems.

Datums

Over 300 of the most commonly used datums are provided. Use the function CoordSys.**GetDatums**() to get a list of available datums. Unlimited additional datums can be created with any ellipsoid and conversion method. ProLat for .Net provides advanced datum shifting capabilities with grid shifts and 3,7 and 10 parameter Molodensky-Badekas methods.

A few common datums include WGS84, NAD27, NAD83, NAD83-ALABAMA-HARN, and many others.

Units

Use the function CoordSys.**GetUnits**() to get a list of available units.

Unit	Description	Unit	Description
BENOIT CHAINS	Benoit's Chain	MILLIMETERS	Millimeter
BENOIT LINKS	Benoit's Link	NAUTICAL MILES	International Nautical Mile
CENTIMETERS	Centimeter		
CHAINS	Chains	PERCH	Perch
CLARKE CHAINS	Clarke's Chains	POLE	Pole
CLARKE FEET	Clarke's Feet	RODS	Rod
DECIMETERS	Decimeter	SEARS CHAINS	Sear's Chain
FEET	Interntnl Feet	SEARS LINKS	Sear's Link
GUNTER CHAIN	Gunter's Chain	SEARS YARD	Sear's Yard
GUNTER LINKS	Gunter's Link	TENTHFEET	1/10 Feet
INCHES	Interntnl Inch	TENTHUSFEET	1/10 USFEET
INDIAN YARD	Indian Yard	U.S. SURVEY MILES	U.S. Survey Mile
INTERNATIONAL MILES	Interntnl Mile	USFEET	US Survey Feet
KILOMETERS	Kilometer	USINCHES	U.S. Survey Inch
LINKS	Links	YARDS	International Yard
METERS	Meter		

Using Proj.4 Compatible Definition Strings

ProLat for .Net

```
C#:
CoordSys latlong = CoordSys.GetCS("+proj=utm +zone=17 +datum=WGS84”);

VB:
Dim latlong As CoordSys = CoordSys.GetCS("+proj=utm +zone=17 +datum=WGS84”)
```

ProLat for Win32

```
C++ / VB6 / etc.
LL = ProLatDefineDef(+proj=utm +zone=17 +datum=WGS84);
```

ProLat for Android

```
Java
CoordSys latlong = CoordSys.GetCS("+proj=utm +zone=17 +datum=WGS84”);
```

Note that in ProLat the plus sign (+) before each parameter is optional.

Defining a coordinate system with a Proj.4 string requires three basic elements.

1. A “proj=” parameter and relevant parameters for that projection. Some projections such as “proj=latlong” and “proj=geocent” have no additional parameters. Most projections, however, have lat_0=, lon_0=, x_0=, y=0 to specify central latitude, central longitude, and an offset. See additional parameter information near the end of this manual.
2. Datum information. A datum contains the ellipsoid, the method used to convert to the WGS84 datum, and the position of zero degrees. An ellipsoid defines the shape of the earth that is used. Coordinates in one datum will land in a different place than the same coordinates in a different datum. It is possible to convert coordinates in one datum to a different datum using either a formula (towgs84= parameters) or with a grid lookup file (nadgrids= parameter). The zero degree position defaults to Greenwich. Some coordinates have a different zero position specified with the “pm=” parameter.
3. The units. For latitude/longitude coordinates, the units default to degrees. For XY projections, the units default to meters. Use the “units=” or “to_meter=” parameter to specify the units.

In summary, make sure the coordinate system specifies the projection, datum, and units.

Projections

ProLat includes the following projections for use in a Proj.4 definition. This represents a comprehensive list of projections. If the needed projection is not available, contact Effective Objects.

aea	Albers Equal Area
aeqd	Azimuthal Equidistant
airy	Airy
aitoff	Aitoff
apian	Apian Globular I
august	August Epicycloidal
bacon	Bacon Globular
bipc	Bipolar conic of western hemisphere
boggs	Boggs Eumorphic
bonne	Bonne (Werner lat_1=90)
cass	Cassini
cc	Central Cylindrical
cea	Equal Area Cylindrical
chamb	Chamberlin Trimetric
collg	Collignon
crast	Craster Parabolic
denoy	Denoyer Semi-Elliptical
eck1	Eckert I
eck2	Eckert II
eck3	Eckert III
eck4	Eckert IV
eck5	Eckert V
eck6	Eckert VI
eqc	Equidistant Cylindrical
eqdc	Equidistant Conic
fahey	Fahey
fouc_s	Foucaut Sinusoidal
gall	Gall Stereographic
geocent	Geocentric, ECEF, XYZ
geos	Geostationary Satellite View
gins8	Ginsburg VIII (TsNIIGAiK)
gn_sinu	General Sinusoidal Series
gnom	Gnomonic
goode	Goode Homolosine
gs48	Mod. Stererographics of 48 U.S.
gs50	Mod. Stererographics of 50 U.S.
gstmerc	Gauss-Schreiber Transverse Mercator (aka Gauss-Laborde Reunion)
hammer	Hammer & Eckert-Greifendorff
hatano	Hatano Asymmetrical Equal Area
kav5	Kavraisky V
kav7	Kavraisky VII
krovak	Krovak projection
latlong	Latitude/Longitude (non-projected) (latlon, lonlat, longlat)
lcc	Lambert Conformal Conic
lcca	Lambert Conformal Conic Alternative
leac	Lambert Equal Area
lee_os	Lee Oblated Stereographic
loxim	Loximuthal
lsat	Space oblique for LANDSAT
mbt_s	McBryde-Thomas Flat-Polar Sine (No. 1)
mbt_fps	McBryde-Thomas Flat-Pole Sine (No. 2)
mbtfpp	McBride-Thomas Flat-Polar Parabolic
mbtfpq	McBryde-Thomas Flat-Polar Quartic
mbtfps	McBryde-Thomas Flat-Polar Sinusoidal
merc	Mercator
mil_os	Miller Oblated Stereographic
mill	Miller Cylindrical
moll	Mollweide
murd1	Murdoch I
murd2	Murdoch II
murd3	Murdoch II
natearth	Natural Earth
nell	Nell
nell_h	Nell-Hammer
nicol	Nicolosi Globular
nsper	Near-sided perspective
nzmg	New Zealand Map Grid
ob_tran	General Oblique Transformation
ocea	Oblique Cylindrical Equal Area
oea	Oblated Equal Area
omerc	Oblique Mercator
ortel	Ortelius Oval
ortho	Orthographic
pconic	Perspective Conic
poly	Polyconic (American)
putp1	Putnins P1
putp2	Putnins P2
putp3	Putnins P3
putp3p	Putnins P3’
putp4	Putnins P4
putp4p	Putnins P4’
putp5	Putnins P5
putp5p	Putnins P5’
putp6	Putnins P6
putp6p	Putnins P6’
qua_aut	Quartic Authalic
rhealpix	rHEALPix
robin	Robinson
rouss	Roussilhe Stereographic
rpoly	Rectangular Polyconic

sinu	Sinusoidal
somerc	Swiss Oblique Mercator
stere	Stereographic
sterea	Oblique Stereographic Alternative
tcc	Transverse Central Cylindrical
tcea	Transverse Cylindrical Equal Area
tissot	Tissot
tmerc	Transverse Mercator
tpeqd	Two Point Equidistant
tpers	Tilted perspective
ups	Universal Polar Stereographic
urm5	Urmaev V
urmfps	Urmaev Flat-Polar Sinusoidal
utm	Universal Transverse Mercator
vandg	van der Grinten I
vandg2	van der Grinten II
vandg3	van der Grinten III
vandg4	van der Grinten IV
vitk1	Vitkovsky I
wag1	Wagner I (Kavraisky VI)
wag2	Wagner II
wag3	Wagner III
wag4	Wagner IV
wag5	Wagner V
wag6	Wagner VI
weren	Werenskiold I
wink1	Winkel I
wink2	Winkel II
wintri	Winkel Tripel

Customizing the Coordinate System Database

It is possible to add your own special coordinate system definitions to the standard ProLat definitions. Use the following steps:

1. Look in the geodesy folder for Group.txt, Datums.txt, Units.txt, etc. The ProlatWindows.dll file includes a copy of all these files internally. However, you can tell ProLat to use an external copy by calling the function CoordSys.**AddFileLocationFolder**(foldername). Make a copy of the geodesy folder that you can customize. Then pass the path to this folder to AddFileLocationFolder(). Now ProLat will use the definitions from the folder instead of the internal data.
2. Edit Group.txt. Notice that each line corresponds to a system file with the Name field with .txt extension. For example, the UTM system has a corresponding UTM.txt file. You can add a new system to Group.txt by copying an existing line and changing the strings. If you do this to add a new system, be sure to create a corresponding .txt file. For example, copy AUTRALIA.txt to MySystems.txt and edit the new file with your own definitions.
   You don't have to create a new system in Group.txt. It is ok to edit one of the system files such as AUSTRALIA.txt and edit/add definitions there. It depends on what works best for your application.
3. Add new datums by editing Datums.txt. Add new ellipsoids by editing Ellipsoids.txt.
4. ProLat creates a coordinate definition by combining the sub definitions from the selected system, the Datum/ellipsoid, and the units. This is useful to know, for example, if your custom definition needs a special datum. In this case add the special datum to Datums.txt instead of directly in MySystems.txt.

How Does ProLat Internally Perform Conversions?

Internally, ProLat uses a well defined conversion method to convert between any two coordinate systems. It is not necessary to know following details to use ProLat, but it may help in understanding how it works.

1. Scale units to meters (for most projections).
2. Convert the coordinate from the source projection to latitude/longitude

3. If the destination coordinate system is in a different datum, perform a datum shift.
    a. NAD27 to NAD83/WGS84 uses NADCON grid shift tables
    b. Other datums use a set of towgs84 (pronounced "To WGS84") parameters to convert to WGS84 and then to the destination datum. (This process involves an intermediate conversion to geocentric xyz coordinates.)
4. Project the new latitude/longitude to the destination projection.
5. Scale to the destination units.

What is a Coordinate System?

A coordinate system consists of a set of parameters that defines how a coordinate in space is produced. For example, a GPS unit calculates an XYZ position relative to the center of the Earth. Then, the GPS converts that value to a latitude, longitude, and height based on an estimate of the size and shape of the earth called an ellipsoid. A surveyor may need to convert the latitude longitude coordinate to the State Plane Coordinate system, which uses either a Transverse Mercator projection or a Lambert Conformal Conic projection.

If we analyze the steps in the preceding paragraph, a coordinate system needs to know the ellipsoid parameters and the projection parameters. If the ellipsoid is not the WGS84 ellipsoid, there is one more parameter needed to define how to convert to the WGS84 ellipsoid. The conversion to another ellipsoid is done with a datum grid shift file or with a mathematical translation.

By having the ellipsoid, projection, and WGS84 translation parameters a complete coordinate system is formed. See the examples of coordinate systems in various sections of this manual.

What is a Datum?

A datum consists of a a set of parameters that define how is defined by a set of constants specifying the coordinate system used for geodetic control, i.e., for calculating the coordinates of points on the Earth.[1] With ProLat, it is not necessary to know the specific parameters of a datum. It is only necessary to determine the datum name used for the available coordinate points and specify that name in the GetCS() method. It is possible to use raw parameters if desired.

The function CoordSys.GetDatums() produces a list of available datum names. ProLat also can use the Proj.4 definitions as NAD27, NAD83, WGS84.

Using a datum name, ProLat looks up the specific parameters that define the datum. A datum has constants for an ellipsoid that defines the shape of Earth considering it closely matches an ellipsoid rather than a perfect sphere. An ellipsoid is usually specified by the semi-major axis, "**a=**", and the reciprocal flattening, "**rf=**". The standard 0 degree of most datums is at Greenwich. A datum that starts elsewhere can use the "**pm**=" parameter. A datum also specifies how to convert to the universal reference datum of WGS84 using the "**nadgrids**=" parameter or the "**towgs84**=" parameter. See documentation for these parameters near the end of the manual.

To summarize, the ellipsoid and the details of how to convert to the WGS84 datum are what defines a datum.

Why is there more than one datum? The use of satellites and other technological improvements in surveying have allowed refinement in the knowledge of the shape of the Earth. Along with these refinements came the process of standardizing the definition of the approximating ellipsoid and establishing an international reference datum. Prior to this, the ellipsoids and datums were established by long line precision surveying and astronomical observation. The processing of the measurements of these surveys led to establishment of ellipsoids which were best fits to local conditions and not the entire Earth and datums which were arbitrary to the surveyor's network. But because this surveying relied upon the use of bubble leveling for alignment of instruments with the horizontal plane (the geoid) they were susceptible to perturbations of the gravity field and thus only useful for local purposes.

Until recently, the reference system for North America has been the North American Datum of 1927 (NAD27), which used Clarke's 1866 ellipsoid and had its origin at Meade's Ranch in Kansas. But because of technical geodetic surveying problems with NAD27 and an interest in standardizing the reference system on an international basis, the North American Datum of 1983 reference system NAD83 has been chosen to replace NAD27. This system is based upon the Geodetic Reference System of 1980 (GRS80) which is geocentric (origin is the center of the Earth's mass) and uses an ellipsoid approximating the entire Earth based on satellite measurements.

The World Geodetic System 1984 (WGS84), the internationally recognized datum was originally based on GRS80, but has had some minor refinements. For practical purposes WGS84 is the same as NAD83 (down to 9 significant digits).

There are several methods for conversion of geographic data between datums, but the most convenient and perhaps common are the Molodensky formula (using the "towgs84=" parameter) and the NADCON (using the "nadgrids=" parameter) used for North American Datum conversions. The Molodensky method is often used for international conversions and is considered to have a conversion accuracy of 5-10m in United States regions. The NADCON method uses of a grid of longitude-latitude corrections from which a correction value can be interpolated for any non-nodal point. The correction grid is determined by minimum curvature gridding of corrections for control points whose location had been accurately determined by both NAD27 and NAD83 surveying methods. Error in conversion with NADCON is generally considered to be less than a meter (0.15m for most of the conus (conterminous U.S.) region) but may suffer in regions of poor control.

ProLat uses the NADCON method to provide conversion between NAD27 and NAD83. Table 1 is a summary of the NADCON grid regions. Conus is the default region for ProLat. To use the other regions, see the function reference for applying the region parameter.

Table 1: NADCON correction regions

	Extent

Region	nadgrids= Parameter	East	West	South	North
Conterminous U.S.	conus.ncn	131° W	63° W	20° N	50° N
Alaska	alaska.ncn	166° W	128° W	46° N	77° N
Hawaii	hawaii.ncn	161° W	154° W	18° N	23° N
Puerto Rico and Virgin Islands	prvi.ncn	68° W	64° W	17° N	19° N
St. George Is., AK	stgeorge.ncn	171° W	169° W	56° N	57° N
St. Lawrence Is., AK	stlrnc.ncn	172° W	168° W	62° N	64° N
St. Paul Is., AK	stpaul.ncn	171° W	169° W	57° N	58° N

Other Datum Conversion Methods

Some datums use a 3, 7, or 10 parameter equation to convert to the WGS84 datum. The parameters are defined with "towgs84" (pronounced "To WGS84") in a Proj.4 definition. See more information about towgs84 in later chapters.

What is a Projection?

In ProLat, a projection defines how latitude longitudes are converted to an XY grid system, usually in meters. There are so many projections because there is no perfect way of flattening an elliptical earth onto a flat grid. Some projections are better for regions that extend primarily in the North South direction such as Transverse Mercator. Other projections work better for regions that extend primarily in the East West directions such as Lambert Conformal Conic.

There special projections that aren't really projections such as Latitude Longitude where there is no projection needed, and Geocentric, which is a 3 dimension XYZ projection.

A projection is most commonly thought of as an easting/northing xy mapping onto a flat surface. It is much like peeling an orange in a way that makes it lie flat. An xy projection is useful because it allows distance calculations between points.

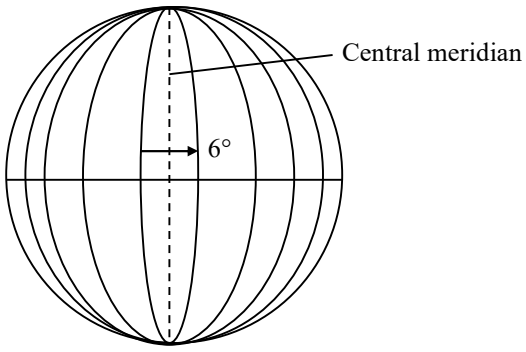
There are several common characteristics of xy projections.

- There is not a single xy project that works perfectly for the whole world at once, for the same reason an orange can't be peeled into one piece that lies perfectly flat without gaps.
- A projection works well in a limited region. That is why there are so many projections.
- Some projections work well in vertical strips, or in horizontal strips, or in diagonal strips, or at the poles. A cartographer chooses the best projection for the job.

ProLat provides support for a long list of projections. Some like the UTM, and State Plane projections are easily accessed through standard functions. Others may be accessed through a custom coordinate system. The mathematical details are not covered in this manual. The user would need to get advanced information from other reference material.

The Universal Transverse Mercator (UTM) Coordinate System

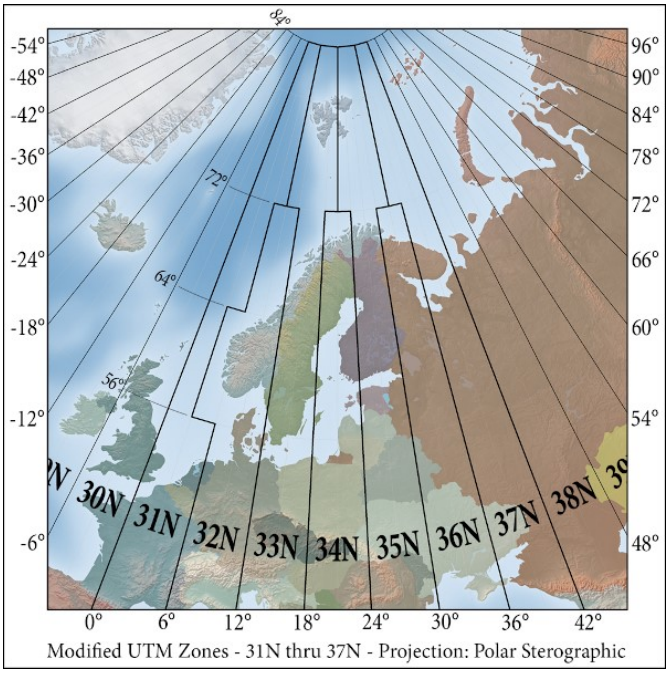
UTM is popular for several reasons. It was adopted by the U.S. Army in 1947 for designating rectangular coordinates on large scale maps. It uses relatively few parameters and covers the whole Earth except the poles. However, its limitation is that each zone covers a narrow vertical strip which may not be suitable for some applications.



UTM is the ellipsoidal Transverse Mercator with 60 predefined zones. Each zone is 6° wide. A central meridian is defined as running down the center of each zone. There are also vertical divisions roughly 8° high, but these are not used as parameters in the transformation. The zone system provides a convenient grid system across the globe except for the poles.

- Its 60 zones are each 6° wide in longitude
- The longitude center of each zone is the central meridian
- The central meridian of each zone always has the x position of 500,000 meters
- X positions increase positively to the east
- Zones are numbered 1 to 60
- Zone 1 covers 180° to 174° W. The central meridian is at 177° W.
- Zone 60 is 174° to 180° E.
- UTM covers latitudes 84°N to 80°S. The corresponding projection for the polar regions is the Universal Polar Stereographic (UPS).
- Vertical divisions are 8° high. Maps that use the UTM grid use letters for the vertical zone. It is not necessary to specify the vertical zone for the transformation.
- For the northern hemisphere, Y starts at 0 meters at the equator

- For the southern hemisphere, Y starts at 10,000,000 at the equator
- For both southern and northern hemispheres, Y increases positively to the north
- Exceptions occur around Norway.



Exception Regions

Lat1	Lat2	Lon1	Lon2	Zone
56	64	0	3	31N
72	84	0	9	31N
56	64	3	12	32N
72	84	9	21	33N
72	84	21	33	35N
72	84	33	42	37N

Distance measurements work within a zone, but not between zones. To use UTM conversions with ProLat for .Net use the CoordSys.GetCS() function with the "UTM" group to create a coordinate system definition. Use the GetCS() function to create another coordinate system to convert to or from UTM. Then use the CoordSys.Transform() function.

Normally a zone is selected by the user. However, to calculate a UTM zone from a lat/lon coordinate, the following pseudocode can be adapted to most languages.

```
// Calc UTM zone, with lon positive east.
zone = (int) ((180 + lon) / 6) + 1;
if (lat >= 56 && lat < 64 && lon >= 3 && lon < 12)
  zone = 32;
else if (lat >= 72 && lat < 84) {
  if (lon >= 0 && lon < 9)
    zone = 31;
  if (lon >= 9 && lon < 21)
    zone = 33;
  if (lon >= 21 && lon < 33)
    zone = 35;
  if (lon >= 33 && lon < 42)
    zone = 37;
}
```

State Plane Coordinate System (SPCS)

To use State Plane Coordinate Systems with ProLat for .Net is as easy as using CoordSys.GetCS with a group name of “US_SPC83” or “US_SPC27” and selecting the right system. For example, the following line selects

SP

The State Plane coordinate system was established by the U.S. Coast and Geodetic Survey in the 1930's. One to five zones (due to its size, Alaska has 10 zones) were set up in each state, using a Lambert Conformal or a Transverse Mercator projection (depending on the dominant orientation of the state, N-S or E-W). The Oblique Mercator projection is used for the Alaskan Panhandle due to its more angular orientation.€€. The specific projection and the size of the zone was selected to fit the geometry of the state, and to keep distortions at or below one part in 10,000. The low distortion makes the SPCS useful at the state and county levels. Zone boundaries are typically political boundaries such as county or city lines.

There are two sets of State Plane definitions. One based on the NAD27 datum and one based on the NAD83 datum. There are also other differences between these two sets. ProLat provides coordinate conversion for State Plane in NAD27 and NAD83.

To use State Plane coordinates with ProLat for .Net, use the CoordSys.GetCS() function with the group, “US_SPC27” or “US_SPC83”. Also define another coordinate system with GetCS(). Then, use CoordSys.Transform to convert between these systems. Use CoordSys.GetSystem(“US_SPC83”) to get a list of zones within the group. See the function reference and examples for details.

Example:

C#:

```
try {
    CoordSys LatLon = CoordSys.GetCS("LAT_LONG", "LAT-LONG", "WGS84", "METERS");
    ‘ Alabama Western zone
    CoordSys SPC = CoordSys.GetCS("US_SPC83", " AL83-W", "NAD83", "METERS");

    // Place the longitude and latitude values in arrays
    double[] Lon_x = new double[] { -87.6 };  // Note West longitudes are negative
    double[] Lat_y = new double[] {  30.1 };
    double[] Hgt_z = new double[] {   0.0 };

    CoordSys.Transform(LatLon, SPC, Lon_x, Lat_y, Hgt_z, 1);

    // Display converted values ...
}
catch(CCConfigException ex)
{
    // Any configuration problems show up here with a message in e.Message
    MessageBox.Show(ex.Message, “ProLat message”, MessageBoxButtons.OK, 0);
}
```

The Geocentric ECEF XYZ Coordinate System

Geocentric earth centered earth fixed coordinates are the raw coordinates used by GPS units. They are 3-dimensional with the zero point of the xyz grid being at the center of the earth. The units are in meters.

Many high-end GPS units store their coordinates as Geocentric to maintain the highest accuracy.

To use Geocentric coordinates, use the CoordSys.GetCS("ECEF", "ECEF", "WGS84", "METERS") to define a coordinate system definition. Also define another coordinate system with GetCS(). Then, use CoordSys.Transform to convert between these systems. See the function reference and examples for details.

Note that Geocentric ECEF XYZ coordinates should not be confused with geocentric latitude longitude coordinates. Geocentric latitude longitude coordinates are rarely used and look similar to standard lat / lon coordinates. However, the latitude angle is relative to the earth's center instead of being relative to the tangential plane of the earth's surface at that location as is used with normal latitude longitude coordinates.

Geocentric ECEF XYZ plot with z extending straight out of the page.

Redistributable Files

ProLat needs the files to be in the same directory where the file ProLatNet.dll or prolat.dll is located. Alternatively, the function CoordSys.SearchPathAdd() or ProLatSetFilePath may be used to define the directory where support files are located.

It is not necessary to keep all of the supporting files with the application if it is known they will not be needed. Some are fairly large.

Here is a list of ProLat for .Net redistributable files:

ProLatNet.dll, prolat.dll
: The main DLL library files. It should be located in the same directory as the executable, or in a directory that is on the PATH environment variable.

  The following files should be located in the same directory as ProLatNet.dll, or use CoordSys.SearchPathAdd() to specify their location.

Geodesy*.txt
: The files in this folder constitute the Geodesy database. They normally should reside in a sub-folder named Geodesy in the folder with the application.

conus
: NAD27 to NAD83 conversion for the Conterminous U.S.

ntv1_can.dat
: NAD27 to NAD83 conversion for Canada. Note: the improved ntv2_can.gsb is available for free end-user download from the NRCan web site at http://www.geod.nrcan.gc.ca/index_e/products_e/software_e/ntv2_e.html

alaska
: NAD27 to NAD83 conversion for Alaska.

hawaii
: NAD27 to NAD83 conversion for Hawaii.

prvi
: NAD27 to NAD83 conversion for Puerto Rico and Virgin Islands.

stgeorge
: NAD27 to NAD83 conversion for St. George Is. AK.

srlrnc
: NAD27 to NAD83 conversion for St. Lawrence Is. AK.

stpaul
: NAD27 to NAD83 conversion for St. Paul Is. AK.

xxharn.hrn
: HARN/HPGN to NAD83 grid shift files for all NADCON regions.

Programming in Windows .Net Environments

This section covers Visual Studio .Net programming for C#, VB .Net, and C++/CLI.

Steps for Visual Studio and Windows 8

The Examples folder has examples for C#, VB, and C++ in Visual Studio 2010. ProLat also works with Windows 8 and Visual Studio 2012.

For both VS2010 and VS2012, here are the steps to use ProLat for .Net:

1. Add a reference to ProLatPortable.dll. This is done by right clicking on the Visual Studio project and select Add Reference. Navigate to the ProLatPortable\bin\Release folder and select ProLatPortable.dll.

2. At the top of a source code file add a line to import the namespace:
   C#: add `using ProLatNet;`
   VB: add `Imports ProLatNet`
   C++: add `using namespace ProLatNet;`

   Call the ProLat functions in the class CoordSys as shown in the examples. In most cases, these two steps are all that is needed.

Windows Phone 7 and 8

ProLat for .Net is works seamlessly in Windows Phone 7 and 8. For Windows Phone 7, the Phone 7 SDK is required from Microsoft. It provides a phone emulator.

A ProLat example for Windows Phone 7 is provide in Examples\ProLatWPA for VS2010.

For Windows Phone 8, it is necessary to use Windows 8 with Visual Studio 2012 Express for Windows Phone 8 or a higher version of Visual Studio. The ProLatPortable.dll file works with both Windows Phone 7 and 8.

.Net Examples

Examples for C# .Net and VB .Net are included in the "examples" folder. These are created with Visual Studio 2010. See below for basic examples.

Using ProLat for .Net functions

Here is an example that converts latitude / longitude coordinates to UTM coordinates.

C#:

```
Using ProLatNet;

try {
    // Get a coordinate system for standard latitude/longitudes
    CoordSys LatLon = CoordSys.GetCS("LAT_LONG", "LAT-LONG", "WGS84", "METERS");

    // Get a coordinate system for UTM zone 17 North
    CoordSys UTM17 = CoordSys.GetCS("UTM", "UTM-17N", "WGS84", "METERS");

    // Place the longitude and latitude values in arrays
    double[] Lon_x = new double[] { -82.0 };  // Note West longitudes are negative
    double[] Lat_y = new double[] {  36.0 };
    double[] Hgt_z = new double[] {   0.0 };


    // Peform the conversion.  In this case there is only one point
    CoordSys.Transform(LatLon, UTM17, Lon_x, Lat_y, Hgt_z, 1);

    // Display converted values ...
}
catch(CCConfigException ex)
{
    // Any configuration problems show up here with a message in e.Message
    MessageBox.Show(ex.Message, "ProLat message", MessageBoxButtons.OK, 0);
}
```

Let's step through this example to see the essential parts. Most of the capabilities are provided by the CoordSys class.

1. Use the CoordSys.GetCS() method to get a coordinate system for the source coordinates. In this case it is latitude/longitude degrees. See the Groups and Systems sections below to find the values to use.
2. This example converts to UTM so it also gets a coordinate system for UTM zone 17.
3. Next the source latitude and longitude is placed in arrays. ProLat for .Net requires arrays because they are much more efficient when processing lots of data. So we start out that way from the first examples.
4. Finally, CoordSys.Transform converts the coordinates. Transform will convert between any defined coordinate systems. To convert the other direction, just swap the first two parameters. The conversion is in-place within the arrays because when processing large arrays it is more efficient.
5. Error handling is provided with the catch mechanism. All ProLat errors will use CSConfigException to report a message. An application also can check CoordSys.GetErrNo() to get a numerical code for the last error, although the message is

more descriptive. Sometimes a conversion may fail without throwing an exception, in which case the converted values will be very large to indicate the problem.

Alternately Proj.4 definitions may be used as in the following example.

C#:

```
// Get a coordinate system for latitude/longitudes using a Proj.4 definition
CoordSys LatLon = CoordSys.GetCS("+proj=longlat +datum=WGS84");

// Get a coordinate system for UTM zone 17 North using a Proj.4 definition
CoordSys UTM17 = CoordSys.GetCS("+proj=utm +zone=17 +datum=WGS84");
```

VB:

```
Dim LatLon As CoordSys = CoordSys.GetCS("+proj=longlat +datum=WGS84")
Dim UTM17 As CoordSys = CoordSys.GetCS("+proj=utm +zone=17 +datum=WGS84")
```

C++:

```
CoordSys^ LatLong = CoordSys::GetCS("LAT_LONG", "LAT-LONG", "WGS84", "METERS");
CoordSys^ UTM17 = CoordSys::GetCS("UTM", "UTM-17N", "WGS84", "METERS");
```

.Net Class and Function Reference

The following classes are provided in ProLat for .Net:

- CoordSys: This is the primary class that offers coordinate conversion and projection capabilities.
- DMS: This class provides parsing of common text coordinate formats. For example it can convert W 92°30'21.5" N 43°30'21.5" into decimal format of -92.5059722 43.5059722, etc.

Class CoordSys

See the Examples section near the beginning of this manual for a quick view of how the CoordSys class is used to perform coordinate conversion.

There is no CoordSys constructor. Instead, use CoordSys.GetCS to get an instance of a CoordSys object that represents a fully qualified coordinate system.

The following methods are called statically on the class (not on an instance):

GetCS() - Creates a coordinate system definition returned as an instance of CoordSys.

Transform() - The main function to convert coordinate points.

GetGroups() - Returns a list of groups in the Geodesy coordinate systems database.
GetSystems() - Returns a list of systems in a selected group.
GetDatums() - Returns a list of available datums in the Geodesy database.

GetUnits() - Returns a list of available units in the Geodesy coordinate systems database.

GetProjNames() - Returns a list of projections available for Proj.4 compatible definitions.
GetProjDescriptions() - Returns a list of descriptions of projections.
GetProjParameters() - Returns an abbreviated list of parameters needed by projections.

AddFileLocationAssembly() - Informs ProLat to look in a new assembly for supporting files.
AddFileLocationFolder() - Adds a directory where support files are located.

EllipsoidForward() - Calculate the location when given the distance and direction.
EllipsoidInverse() - Calculate the shortest distance and direction between two locations.

GetErrNo() - Get's an error code.

The following items are available with an instance of CoordSys:

ScaleConvergence() - Find the scale error and direction to the real location for a projection.

The following member variables are available to provide the numerical values of a projection's parameters. These values are undocumented, but made available. Use these with care because each projection sets the variables differently.

a - Ellipsoid semi-major axis in meters
b - Ellipsoid semi-minor axis in meters
rf - Ellipsoid reciprocal flattening
k0 - Scale factor
lon_0 - Central meridian
lat_0 - Central parallel
x_0 - False easting
y_0 - False northing
to_meter - Unit scaling

Class DMS - Text Coordinate Parsing Class

GetDMS() - Extract coordinate pairs from a string into arrays
GetDMSSingle() - Get a single component such as latitude only or longitude only.
GetLat() - Get a single latitude value.
GetLon() - Get a single longitude value.

Testing and Verification

Testing is very important for coordinate conversion applications!

Coordinate translations require a high degree of care to verify that the results are as intended. There are many different parameters which increase the chances of a one parameter being

configured wrong. Effective Objects highly recommends using a verification procedure at the design time of your program and each time you use a different set of configuration parameters.

A verification procedure is straightforward. Find two or more known geographic coordinates with their translated values. These are known as control points. Run these coordinates through the software to make sure they transform properly to the known values.

CoordSys Method Reference

GetCS

.Net
static CoordSys **GetCS** (string **Group**, string **System**, string **Datum**, string **Units**)
static CoordSys **GetCS** (string **ProjDef**)

Creates a coordinate system object that may be used with Transform to convert coordinates.

Input Parameters:

- **Group** The name of a Geodesy group. (Use GetGroups() to get a list of groups.)
- **System** The name of a system within the group. (Use GetSystems() to get a list.)
- **Datum** The name of a datum. (Use GetDatums() to get a list.)
- **Units** The name of the units for this coordinate system. (Use GetUnits() to get a list.)
- **ProjDef** A Proj.4 compatible definition string. This supports all Proj.4 parameters except +init. See the appendix for more information about these parameters.

Outputs:

Returns an instance of CoordSys which can be used with the Transform method.

Errors:

Throws an exception named CSConfigException if an error is detected. Use the exception message for more information. Note that some errors such as typos in values are not detected, so it is important to check a few known reference points.

GetCS creates a coordinate system object of type CoordSys. It attempts to check parameters and the existence of the proper supporting files such as grid shift files. The resulting instance can be passed to Transform to convert coordinates to a different coordinate system.

Examples with Geodesy definitions:

```
C#:
CoordSys latlong = CoordSys.GetCS("LAT_LONG", "LAT-LONG", "WGS84", "METERS”);
CoordSys utm17 = CoordSys.GetCS("UTM", "UTM-17N", "WGS84", "METERS”);
```

```
VB.Net:
Dim latlong as CoordSys = CoordSys.GetCS("LAT_LONG", "LAT-LONG", "WGS84", "METERS”)
Dim utm17 as CoordSys = CoordSys.GetCS("UTM", "UTM-17N", "WGS84", "METERS”)


C++/CLI:
CoordSys^ latlong = CoordSys::GetCS("LAT_LONG", "LAT-LONG", "WGS84", "METERS");
CoordSys^ utm17 = CoordSys::GetCS("UTM", "UTM-17N", "WGS84", "METERS");
```

Examples with Proj.4 definitions:

```
C#:
CoordSys utm17 = CoordSys.GetCS("+proj=utm +zone=17 +datum=WGS84”);

VB:
Dim utm17 As CoordSys = CoordSys.GetCS("+proj=utm +zone=17 +datum=WGS84”)

C++/CLI:
CoordSys^ utm17 = CoordSys::GetCS("+proj=utm +zone=17 +datum=WGS84”);
```

Transform

.Net
Static void **Transform** (CoordSys ***Source,*** CoordSys ***Destination***,
double[] ***LonX,*** double[] ***LatY***, double[] ***HgtZ***,
int ***PointCount***)
Win32/64
int **ProLatTransform** (int ***FromHandle,*** int ***ToHandle***,
double *__LonX,__ double *__LatY__ , double *__HgtZ__,
int ***iCount***)

Transforms coordinates between coordinate systems.

Inputs:

Source: A CoordSys instance returned from GetCS. It represents the source coordinate system

Destination: A CoordSys instance returned from GetCS. It represents the destination coordinate system.

LonX: A managed double array that acts as an input and an output. The conversion is done in-place and LonX will return with the results. Note that West directions should have a negative value.

LatY: A managed double array that acts as an input and an output. The conversion is done in-place and LatY will return with the results. Note that South directions should have a negative value.

HgtZ: A managed double array that acts as an input and an output. The conversion is done in-place and HgtZ will return with the results. If no Z height values are present in the source coordinates, the HgtZ array should be filled with zeros.

PointCount : The number of coordinate points to convert.

Note that if a conversion is not successful, Transform may throw an exception for critical errors or may return a huge value for non-critical errors. Error values will be greater than 1e100. Use GetErrNo() to check for an error code.

There are several possible reasons for an error. Double check the parameters for the coordinate systems. The two most likely problems encountered are: 1. Coordinates out of range for valid conversion with the defined parameters; and 2. ProLat was unable to find supporting files needed to shift datums, etc. Also, note that ProLat functions are not able to check for all valid parameters because of the unlimited possible combinations.

It is highly recommended to test conversion parameters using control points. This involves getting several coordinate points with their known converted values. Test using these control points with ProLat functions to verify the desired conversion is produced.

.Net Examples:

VB:

```
Try
  Dim latlong as CoordSys = CoordSys.GetCS("LAT_LONG", "LAT-LONG", "WGS84", "METERS")
  Dim utm17 as CoordSys = CoordSys.GetCS("UTM", "UTM-17N", "WGS84", "METERS")

  Dim LonX(1), LatY(1), Z(1) As Double
  LonX(0) = -82.0
  LatY(0) = 43.1
  Z(0) = 0.0

  CoordSys.Transform(latlong, utm17, LonX, LatY, Z, 1)

  ' Display values here
Catch ex As CSConfigException
  MsgBox(ex.Message, MsgBoxStyle.OkOnly, "ProLat Message")
End Try
```

C#:

```
Try
{
    CoordSys ll = CoordSys.GetCS("proj=longlat datum=WGS84");
    CoordSys utm10 = CoordSys.GetCS("proj=utm zone=10 datum=WGS84");
    double[] x = { -122.0 };  // Downtown Seattle
    double[] y = { 47.0 };
    double[] z = { 0 };

    CoordSys.Transform(ll, utm10, x, y, z, 1);
}
Catch (CSConfigException e)
{
    MessageBox.Show(e.Message, "ProLat message");
}
```

C++/CLI:

```
try
{
    CoordSys^ A = CoordSys::GetCS("LAT_LONG", "LAT-LONG", "WGS84", "METERS");
    CoordSys^ B = CoordSys::GetCS("UTM", "UTM-17N", "WGS84", "METERS");

    array<double>^ lon_x = gcnew array<double>(1);
    array<double>^ lat_y = gcnew array<double>(1);
    array<double>^ z = gcnew array<double>(1);
    lon_x[0] = -82.5;
    lat_y[0] = 43.0;
    z[0] = 0;

    CoordSys::Transform(A, B, lon_x, lat_y, z, 1);
}
catch(CSConfigException^ e)
{
    MessageBox::Show(e->Message, "ProLat message");
}
```

Win32 Examples:

See the examples folder for complete examples.

VB6 / Excel / Access:

```
iReturn = ProLatTransform(hFrom, hTo, dfX(0), dfY(0), dfZ(0), iDetected)
```

C++:

```
iReturn = ProLatTransform(hFrom, hTo, X, Y, Z, iDetected);
```

Android Examples:

See the examples folder for complete examples.

```
csA.transform(csB, 2,
```

C++:

```
iReturn = ProLatTransform(hFrom, hTo, X, Y, Z, iDetected);
```

CoordSys.AddFileLocationAssembly

.Net
void **CoordSys.AddFileLocationAssembly** (Assembly ***assembly***)

Adds an assembly reference so that ProLat can look for supporting files such as grid shift files and custom definitions.

ProLat includes coordinate definitions and datum grid files within its assembly. If the application needs files that are not already there (like NTV2_0.GSB, which is fairly large) the application will need to include the file in its own assembly and then tell ProLat to look in the assembly with this function.

Here are the steps for adding a file to a project and then telling ProLat about the project assembly.

a. Right-click the project and select Add -> New folder. Name the new folder "Geodesy" or "ProLatFiles". The folder name does not matter because ProLat will search all the files within an assembly to find the one it needs.

b. Right-click the folder that was just added and select Add -> Existing item… Navigate to the file needed and select the file(s) to add. (It is possible to highlight the first file and then Shift-click on the last file to select them all.)

c. In the Visual Studio folder where the files were just added, select all of the files. In the properties window at the bottom right, there is a property named, "Build Action". Change the Build Action to "Embedded Resource". If this is not done, the file will not be available.

d. In the application start-up code, call `CoordSys.AddFileLocationAssembly( Assembly.GetExecutingAssembly() )`

Example:

VB:

```
Imports System.Reflection    (put at top of file)

CoordSys.AddFileLocationAssembly( Assembly.GetExecutingAssembly() )
```

C#:

```
using System.Reflection;  (put at top of file)

CoordSys.AddFileLocationAssembly( Assembly.GetExecutingAssembly() );
```

C++/CLI:

```
CoordSys::AddFileLocationAssembly( Assembly.GetExecutingAssembly() );
```

CoordSys.AddFileLocationFolder

void **CoordSys.AddFileLocationFolder** (String ***folder***)

Adds a folder for ProLat to look for supporting files such as grid shift files. Call this function once when your app starts. It is a static class function so one call applies to all instances of CoordSys. Several folders can be added. Avoid adding the same folder more than once.

Not available with Windows Phone. Use AddFileLocationAssembly() instead.

Example:

VB:

```
CoordSys.AddFileLocationFolder("C:\geodesy")
```

C#:

```
CoordSys.AddFileLocationFolder("C:\geodesy" );
```

C++/CLI:

```
CoordSys::AddFileLocationFolder("C:\geodesy");
```

CoordSys.GetErrNo

int **CoordSys.GetErrNo**()

Returns an error code. 0 indicates no error, and a non-zero error indicates an error was detected. Most ProLat functions will throw a CSConfigException if a serious error is detected. Check the exception message to get more details.

In some cases Transform() can have error that do not throw an exception. It is possible to use GetErrNo() to check for detected errors after calling Transform(). In this case the converted coordinates that caused an error will be set to a huge value. For some errors, Transform will continue processing the whole array.

When a non-zero error is detected, check the returned data values from the previous call to a ProLat function. Sometimes a conversion may detect an error and set the value to a very large number (greater than 1E100).

Possible error codes:
-14 Latitude or longitude exceeded limits
-15 Invalid x or y

Example:

VB:

```
CoordSys.Transform(latlong, utm, x, y, z, 1000)
If CoordSys.GetErrNo() <> 0 Then
   ' Check for high values
End If
```

C#:

```
CoordSys.Transform(latlong, utm, x, y, z, 1000);
if (CoordSys.GetErrNo() != 0)
{
  // Check for high values
}
```

CoordSys.GetGroups and related functions

List<string> **CoordSys.GetGroups**()
List<string> **CoordSys.GetGroups**(List<string> Descriptions)
List<string> **CoordSys.GetGroups**(List<string> Descriptions, List<string> SuggestedDatum)

List<string> **CoordSys.GetSystems**(string Group)
List<string> **CoordSys.GetSystems**(string Group, List<string> Descriptions)

List<string> **CoordSys.GetDatums**()
List<string> **CoordSys.GetDatums** (List<string> Descriptions)

List<string> **CoordSys.GetUnits**()
List<string> **CoordSys.GetUnits** (List<string> Descriptions)

These functions return the lists of available groups, systems, datums, and units available for use with the function CoordSys.GetCS(group, system, datum, units).

For examples, see the ProLat Examples folder and look for the ProLatCalc examples.

CoordSys.GetProjNames and related functions

List<string> **CoordSys.GetProjNames**()
List<string> **CoordSys.GetProjNames** (List<string> Descriptions)
List<string> **CoordSys.GetProjNames** (List<string> Descriptions, List<string> Params)

These functions return the lists of available projections for use with the function CoordSys.GetCS(string definitionProj4). The returned names may be used with the "+proj=" parameter. The param strings are fairly cryptic at the moment. "Sphere" indicates it can take just a radius parameter, and "Ellips" indicates it needs the semi-major axis, "+a=", and the reciprocal flattening, "+rf". See the list of projections and parameters at the end of this manual for more details.

For Win32

The following functions get lists of available geodesy items.

```
int ProLatGetGroups(char *Groups, char *Descs, char *Datums);
int ProLatGetSystems(const char *Group, char *Systems, char *Descs, char *Datums, char *Units)
int ProLatGetDatums(char *Datums, char *Descs, char *Methods, char *Ellipsoids)
int ProLatGetUnits(char *Units, char *Descs, char *Suffixes, char *FmMeters, char *ToMeters);
```

These functions fill the strings with a list of available items separated by a | character. Use a split function to separate them into individual items. The strings should be pre-allocated with at least 10,000 characters of space. See the ProLat Calculator example.

EllipsoidInverse

static void **EllipsoidInverse** (CoordSys ***CS,*** double[] ***LatFrom,*** double[] ***LonFrom***,
double[] ***LatTo***, double[] ***LonTo***, double[] ***Geodesic***,
double[] ***AzForward***, double[] ***AzBack***, int ***PointCount***)

static void **EllipsoidInverse2** (double **a**, double ***rf,*** double[] ***LatFrom,*** double[] ***LonFrom***,
double[] ***LatTo***, double[] ***LonTo***, double[] ***Geodesic***,
double[] ***AzForward***, double[] ***AzBack***, int ***PointCount***)

Calculates the geodesic (shortest distance), the Forward Azimuth (angle from North), and the Back Azimuth accurate to the ellipsoid.

Inputs:

CS: A CoordSys instance returned from GetCS. It represents the coordinate system with an xy projection.

LatFrom: A managed array of double float values with the starting latitude values.

LonFrom: A managed array of double float values with the starting longitude values.

LatTo: A managed array of double float values with the destination latitude values.

LonTo: A managed array of double float values with the destination longitude values.

a: The ellipsoid semi-major axis. This alternate form allows calculations without needing a full coordinate system definition.

rf: The ellipsoid reciprocal flattening.

PointCount : The number of coordinate points to convert.

Outputs:

Geodesic: A managed array of double to receive the results of the geodesic calculation.

AzForward: A managed array of double float to receive the forward azimuth values.

AzBack: A managed array of double float to receive the back azimuth values.

This function uses the T. Vincenty modified Rainsford's method which is accurate to the ellipsoid. The To and From positions should not be at a geographic pole.

Example:

```
C#:
CoordSys utm17 = CoordSys.GetCS("+proj=utm +zone=17 +datum=WGS84”);
double[] LonFrom = { -93 };
double[] LatFrom = { 37 };
double[] LonTo = { -93.1 };
double[] LatTo = { 37.1 };
double[] Geodesic = new double[1];
double[] AzForward = new double[1];
double[] AzBack = new double[1];
CoordSys.EllipsoidInverse(utm17, LonFrom, LatFrom, LonTo, LatTo,
                          Geodesic, AzForward, AzBack, 1);
```

EllipsoidForward

static void **EllipsoidForward** (CoordSys ***CS,*** double[] ***LatFrom,*** double[] ***LonFrom***,
double[] ***Geodesic***, double[] ***Azimuth***,
double[] ***LatTo***, double[] ***LonTo***, int ***PointCount***)

static void **EllipsoidForward2** (double **a**, double ***rf,*** double[] ***LatFrom,*** double[] ***LonFrom***,
double[] ***Geodesic***, double[] ***Azimuth***,
double[] ***LatTo***, double[] ***LonTo***, int ***PointCount***)

Calculates a point from a starting point with geodesic distance, and azimuth (angle from North), accurate to the ellipsoid.

Inputs:

CS: A CoordSys instance returned from GetCS. It represents the coordinate system with an xy projection.

LatFrom: A managed array of double float values with the starting latitude values.

LonFrom: A managed array of double float values with the starting longitude values.

Geodesic: A managed array of double with the geodesic distance.

Azimuth: A managed array of double float with the angle in degrees from North.

a: The ellipsoid semi-major axis. This alternate form allows calculations without needing a full coordinate system definition.

rf: The ellipsoid reciprocal flattening.

PointCount : The number of coordinate points to convert.

Outputs:

LatTo: A managed array of double float values with the destination latitude values.

LonTo: A managed array of double float values with the destination longitude values.

This function uses the T. Vincenty modified Rainsford's method which is accurate to the ellipsoid. The To and From positions should not be at a geographic pole.

Example:

```
C#:
CoordSys utm17 = CoordSys.GetCS("+proj=utm +zone=17 +datum=WGS84");
double[] LonFrom = { -93.0 };
double[] LatFrom = { 37.0 };
double[] Geodesic = { 20000.0 };
double[] Azimuth = { 47.0 };
double[] LonTo = new double[1];
double[] LatTo = new double[1];
CoordSys.EllipsoidForward(utm17, LonFrom, LatFrom, Geodesic, Azimuth,
      LonTo, LatTo, 1);
```

ScaleConvergence

void **ScaleConvergence** (double[] ***X,*** double[] ***Y***,
double[] ***Scale***, double[] ***Convergence***, int ***PointCount***)

Calculates the scale factor and convergence angle for the given XY points of a projection against the ideal scale and true north. ScaleConvergence() is a method of an instance of CoordSys.

Input:

X: A managed array of double float values with the X (Easting) values.

Y: A managed array of double float values with the Y (Northing) values.

Output:

Scale: A managed array of double float to receive the results of the scale factor calculation.

Convergence: A managed array of double float to receive the convergence angle in radians.

PointCount : The number of coordinate points to convert.

This function uses the T. Vincenty modified Rainsford's method which is accurate to the ellipsoid.

Example:

VB:

```
Try
  Dim latlong as CoordSys = CoordSys.GetCS("LAT_LONG", "LAT-LONG", "WGS84", "METERS")
  Dim utm17 as CoordSys = CoordSys.GetCS("UTM", "UTM-17N", "WGS84", "METERS")

  Dim LonX(1), LatY(1), Z(1) As Double
  LonX(0) = -82.0
  LatY(0) = 43.1
  Z(0) = 0.0

  CoordSys.Transform(latlong, utm17, LonX, LatY, Z, 1)

  ' Get the scale and convergence for this conversion
  Dim Scale(1), Convergence(1) As Double
  utm17.ScaleConvergence(LonX, LatY, Scale, Convergence, 1)

Catch ex As CSConfigException
  MsgBox(ex.Message, MsgBoxStyle.OkOnly, "ProLat Message")
End Try
```

GetDMS

static int **GetDMS** (string ***sData,*** int ***Order,***
out double[] ***v1,*** out double[] ***v2***, out double[] ***v3***,
out string ***ErrMsg***)

GetDMS converts text coordinates to double floating point values suitable for coordinate conversion. It reads most common latitude longitude and decimal formats. See the syntax details below for complete details.

A few common formats are:

102d49'12.81"W 35d15'38.36"N 100.72 (100.72 is an optional height or altitude)
W102°49'12.81" N35°15'38.36" 100.72
W 102 49.2135 N 35 15.63933 100.72
102.820225 W 35.2606556 N 100.72

Input:

sData: The string containing text coordinates. A string may contain more than one coordinate separated by newline characters. The results are placed in the v1, v2, v3 arrays.

Order: Determines how latitude longitude values are place in arrays v1 or v2.

0: No latitude longitude detection. The first coordinate value is placed in v1, the second in v2, and the third in v3.

1: Place longitude values in v1, latitude in v2, and altitude in v3. Uses heading letters 'E' and 'W' to detect longitudes, and 'N' and 'S' to detect latitude. For example: N10.5 W30.5 would get v1=30.5 v2=10.5 v3=0.0.

2: Place latitude values in v1, longitude in v2, and altitude in v3. Uses heading letters 'E' and 'W' to detect longitudes, and 'N' and 'S' to detect latitude. For example: W30.5 N10.5 would get v1=10.5 v2=30.5 v3=0.0.

Note: Coordinates that have no heading letter will always be placed in v1 and v2 in the order they are detected. It is up to the calling code to determine their order, usually by some user selectable option.

Output:

v1, v2, v3: Managed arrays of double floating point. The coordinates found in the sData string will be placed into these arrays. The Order parameter determines if ProLatGetDMS detects latitude verses longitude values and in which array they land. The array v3 receives the optional altitude.

ErrMsg: A string that will receive an error message if an error is encountered. This may be NULL to avoid returning an error string..

Returns a count of errors detected. If the count is greater than 0, check the ErrMsg for additional information. Common errors are related to unrecognized elements in a string.

Syntax:

ProLatGetDMS uses pattern recognition to flexibly detect many different coordinate formats. Here is the general syntax specification:

sData contains a list of one or more text coordinates separated by carriage returns or line feeds. The string may contain the following elements:

Comments: Comments are allowed with the following notation:

//	Comment to end of line
#	Comment to end of line
%	Comment to end of line
/* … */	Comment between /* and */

Ignored characters:
space, tab, comma, semicolon, colon, parenthesis, square brackets,
and space followed by minus and another space ‘ - ‘

Coordinates:
A coordinate has the following format:

DMS DMS [optional altitude]

DMS is a flexible format for degrees minutes seconds and heading. Both DMS values in a coordinate need to have the same format. Possible formats include:

Degrees Minutes Seconds Heading	Degrees Minutes Seconds Heading [altitude]
Degrees Minutes Heading	Degrees Minutes Heading [altitude]
Degrees Heading	Degrees Heading [altitude]
Heading Degrees Minutes Seconds	Heading Degrees Minutes Seconds [altitude]
Heading Degrees Minutes	Heading Degrees Minutes [altitude]
Heading Degrees	Heading Degrees [altitude]
Decimal	Decimal [altitude]

Degrees	unsigned decimal value followed by a space, d, D, or °
Minutes	unsigned decimal value followed by a space, or '
Seconds	unsigned decimal value followed by a space, or "
Heading	the letters E, e, W, w, N, n, S, or s The letters W, w, S, and s will create a negative value. Minus signs are ignored for Degrees Minutes and Seconds.
Decimal	signed decimal value with no heading letter

Note that **commas** are used as separators only. They may not be used as a decimal mark or thousands marker.

Syntax Examples:

80d25’49.12”W 35d41’29”N // USGS common
80 25 49.12 E 35 41 29 S // Degrees Minutes Seconds
35°41’29”N 80°25’49.12”W 100.7 // With altitude
W80d25’49.12” N35d41’29”
80°25.81867’W 35°41.48333’N // Degrees Minutes
35 41.48333 S 80 25.81867 E
W80d25.81867 N35d41.48333
W80 25.81867 N35 41.48333 // Garmin common
80.4303111dW 35.6913889dN // Degrees
80.4303111 W 35.6913889 N
S 35.691389° E 80.4303111°
W80.4303111 N35.691389

-80.4303111 35.691389 // Decimal
500145.387 2457353.25 // Decimal UTM easting, northing

Example:

VB:

```
Dim Lon(), Lat(), Z() As Double
Dim sData As String = "80 25 49.12 W  35 41 29 N" & vbCr & vbLf & _
                      "12 25 49.12 E  15 41 29 S  225.4"

Dim msg As String
Dim ErrorCount = DMS.GetDMS(sData, 1, Lon, Lat, Z, msg)

Dim Output As String = "GetDMS returned:" & vbCr & vbLf
If (ErrorCount > 0) Then
    Output = Output & "Errors detected: " & ErrorCount.ToString() & vbCr & vbLf
End If

If (msg.Length > 0) Then
    Output = Output & "Return message: " & msg & vbCr & vbLf
End If

For i = 0 To 1
    Output = Output & Lon(i).ToString() & " " & Lat(i).ToString() & vbCr & vbLf
Next

MsgBox(Output)
```

GetDMSSingle

int **GetDMSSingle** (string ***sData,*** out double[] ***v,*** out int[] ***heading,*** out string ***ErrMsg***)

ProLatGetDMSSingle converts a single value instead of a full coordinate pair.

A few common formats are:

102d49'12.81"W
W 102 49 12.81
117 23.714 E
35.2606556 N

Returns 0 for success or a count of errors detected. When an error occurs the returned coordinates are invalid and the source should be corrected before continuing.

sData: A managed string containing text single values. A string may contain more than one value separated by newline characters. The results are placed in the v output array.

v: A managed array of double floating point. The value found in the sData string will be placed into this array.

heading: A managed array of 4-byte integers. Indicates the heading value detected: 0 no heading detected, 1 North, 2 South, 3 East, 4 West. Note that when South and West headings are detected the value in v is negative.

ErrMsg: A managed string that will receive an error message if an error is encountered.

See the syntax description of GetDMS for details of the possible text format that may be detected. This function however, just detects the one value instead of a full coordinate pair.

GetLat

double **GetLat** (string ***sPoint***)

This function converts a text latitude to a double floating point value.

Throws CSConfigException for a syntax error, out of range, or an invalid heading.

sPoint: A managed string with a latitude value, which may be a DMS formatted value such as N 47 35 11.35.

See the syntax description of GetDMS for details of the possible text format that may be detected. This function however, just detects the one value instead of a full coordinate pair.

GetLon

double **GetLon** (string ***sPoint***)

This function converts a text longitude to a double floating point value.

Throws CSConfigException for a syntax error, out of range, or an invalid heading.

sPoint: A managed string with a longitude value, which may be a DMS formatted value such as W 47 35 11.35.

See the syntax description of GetDMS for details of the possible text format that may be detected. This function however, just detects the one value instead of a full coordinate pair.

MGRS Military Grid Reference System Functions

string **MGRS.ConvertFromGeodetic** (double ***Longitude***, double ***Latitude***, int ***Precision = 5***)
string **MGRS.ConvertFromUPS** (char ***Hemisphere***, double ***Easting***, double ***Northing***,
int ***Precision = 5***)
string **MGRS.ConvertFromUTM** (int ***Zone***, char ***Hemisphere***,
double ***Easting***, double ***Northing***, int ***Precision = 5***)
void **MGRS.ConvertToGeodetic** (int ***InMGRS,*** out double ***Longitude***, out double ***Latitude***)
void **MGRS.ConvertToUPS** (int ***InMGRS,*** out int ***Zone***, out char ***hemisphere***,
out double ***Longitude***, out double ***Latitude***)
void **MGRS.ConvertToUTM** (int ***InMGRS,*** out int ***Zone***, out char ***hemisphere***,
out double ***Longitude***, out double ***Latitude***)

The MGRS support in ProLat for .Net is derived from GeoTrans and has equivalent functionality. To use these functions, an MGRS object needs to be created to specify the ellipsoid.

C#:
```
MGRS mgrs = new MGRS(); // Uses default WGS84 ellipsoid.
// For NAD27 ellipsoid use: new MGRS("datum=NAD27")
string outMGRS = mgrs.ConvertFromGeodetic(lon, lat);
```

VB.Net:
```
Dim Mgrs1 As MGRS = New MGRS()  ' Uses default WGS84 ellipsoid.
' For NAD27 ellipsoid use: New MGRS("datum=NAD27")
Dim OutMGRS As String = Mgrs1.ConvertFromGeodetic(Lon, Lat)
```

These functions throw CSConfigException when an error is detected.

Outputs: The functions **ConvertFromGeodetic, ConvertFromUPS, and ConvertFromUTM** return an MGRS string. The ConvertToXX functions return data in the parameters as described below.

Paramaters:

Longitude: A double value. West longitudes are negative.

Latitude: A double value. South latitudes are negative.

Precision: An optional parameter to indicate the number of digits. This defaults to the standard of 5 digits precision.

Zone: An integer specifying the UTM zone. For ConvertToUPS and ConvertToUTM the zone will return 0 for UPS and 1-60 for UTM.

Hemisphere: A char. 'N' for Northern hemisphere, and 'S' for Southern hemisphere.

Easting: A double value.

Northing: A double value.

InMGRS: A string containing an MGRS value.

For performance it is helpful to group coordinates into like zones because internally it will recreate coordinate systems when the zone changes.

ConvertToUTM will succeed even if the MGRS string is a UPS region. In this case it returns UPS values with the zone set to 0. In the same way, ConvertToUPS can return a UTM value with the zone set from 1 to 60. In these cases the MGRS.warningMessage will contain a warning when the function returns.

Examples:

C#:

```
try
{
    MGRS mgrs = new MGRS(); // Uses default WGS84 ellipsoid.
    // For NAD27 ellipsoid use: new MGRS("datum=NAD27")
    string outMGRS = mgrs.ConvertFromGeodetic(lon, lat);
    mgrs.ConvertToGeodetic(outMGRS, out lon, out lat);
}
catch(CSConfigException ex)
{
    MessageBox.Show(ex.Message);
}
```

VB.Net:

```
 Try
    Dim Mgrs1 As MGRS = New MGRS()  ' Uses default WGS84 ellipsoid.
    ' For NAD27 ellipsoid use: New MGRS("datum=NAD27")

    Dim Lon As Double = 5
    Dim Lat As Double = 57
    Dim OutMGRS As String = Mgrs1.ConvertFromGeodetic(Lon, Lat)
    MessageBox.Show("MGRS example:  lon/lat: " & _
            Lon.ToString() & " " & Lat.ToString() & vbCrLf & _
            "MGRS: " + OutMGRS)
 Catch ex As CSConfigException
    MessageBox.Show(ex.Message)
 End Try
```

Programming in Windows 32/64 Native Environments

This section covers programming in VB6 which includes VBA for Excel and Access. It also covers C++ and other languages compiled into native Windows 32/64 bit environments.

Setup Instructions

1. Make sure prolat.dll is located in the same folder as the executable or is located on the system Path.
2. Add the installation directory to the PATH environment variable. In Windows use Start > Control Panel > System > Advanced > Environment Variables. Edit the PATH variable to add ";C:\ProLat\Win32", or the path where ProLat was uncompressed. Note that prolat.dll does not need to be registered because it is a standard Windows DLL like kernel32.dll, not a COM DLL. It just needs to be on the path to be found by Windows.
3. When redistributing ProLat DLL files, it is usually necessary to copy the files to the same directory as the .EXE file. Windows will look in the .EXE's directory for DLL files before it looks on the path. This simplifies installing the application that use ProLat.

Steps for using ProLat in VB6, Excel, Word, and Access

1. Add the file, prolatdll.bas, into your project. In Excel it is necessary to create a new module and copy/paste the contents of prolatdll.bas.
2. See the examples for details or perhaps copy an example to get started.
3. Make sure the file, prolat.dll, is on the path as described above.

Steps for using ProLat in C++

1. Insert the line:

```
#include "..\..\prolatdll.h"
```

2. Make sure the file, prolat.dll, is on the path as described above.
3. See the examples for details.

Function Reference

ProLatDefineGeodesy and ProLatDefineDef

int **ProLatDefineGeodesy** (const char ***group***, const char ***system***, const char ***datum***, const char ***unit**)

int **ProLatDefineDef** (const char ***projdef***)

Creates a coordinate system object that may be used with ProLatTransform to convert coordinates.

Input Parameters:

- **group** The name of a Geodesy group. (Use ProLatGetGroups() to get a list of groups.)
- **system** The name of a system within the group. (Use GetSystems() to get a list.)
- **datum** The name of a datum. (Use GetDatums() to get a list.)
- **units** The name of the units for this coordinate system. (Use GetUnits() to get a list.)
- **projdef** A Proj.4 compatible definition string. This supports all Proj.4 parameters except +init. See the appendix for more information about these parameters.

Outputs:

Returns an integer handle to a coordinate system that can be used with ProLatTransform().

Errors:

Throws an exception named CSConfigException if an error is detected. Use the exception message for more information. Note that some errors such as typos in values are not detected, so it is important to check a few known reference points.

Returns a 4-byte handle to be used with other ProLat DLL functions. The handle must be closed with ProLatClose() when no longer needed. A zero is returned for an error. Use ProLatErrNo() and ProLatGetStrErr() for error checking.

Examples with Geodesy definitions:

```
C++:
int latlong = ProLatGetGeodesy("LAT_LONG", "LAT-LONG", "WGS84", "METERS");
int utm17   = ProLatGetGeodesy ("UTM", "UTM-17N", "WGS84", "METERS");

VBA and VB6:
Dim latlong As Integer, utm17 as Integer
latlong = ProLatGetGeodesy("LAT_LONG", "LAT-LONG", "WGS84", "METERS")
utm17   = ProLatGetGeodesy ("UTM", "UTM-17N", "WGS84", "METERS")
```

Examples with Proj.4 definitions:

C++:
```
int utm17 = ProLatGetDef("proj=utm zone=17 datum=WGS84");
```

VBA and VB6:
```
Dim utm17 as Integer
latlong = ProLatGetDef("LAT_LONG", "LAT-LONG", "WGS84", "METERS")
utm17   = ProLatGetDef("UTM", "UTM-17N", "WGS84", "METERS")
```

More examples:

UTM: `utm17 = ProLatGetGeodesy("UTM", "UTM-17N", "WGS84", "METERS")`

ECEF (Earth Centered Earth Fixed) Geocentric XYZ:

```
ecef = ProLatGetGeodesy("ECEF", "ECEF", "WGS84", "METERS")
```

State Plane Coordinate System:

```
utm17 = ProLatGetGeodesy("US_SPC83", "CA83-4", "NAD83", "METERS")
```

ProLatTransform

int **ProLatTransform** (int ***FromHandle,*** int ***ToHandle***,
double ****dfX,*** double ****dfY***, double ****dfZ***, int ***iCount***)

Transforms coordinates between coordinate systems.

Returns 0 for success and non-zero for an error condition.

FromHandle: The handle returned from a ProLatDefine function. It represents the source coordinate system

ToHandle: The handle returned from ProLatDefine function. It represents the destination coordinate system.

****dfX***: A pointer to an array of double float values with the source X values or a longitude values. The conversion result is done in-place and *dfX will return with the results.

****dfY***: A pointer to an array of double float values with the source Y values or a latitude values. The conversion result is done in-place and *dfY will return with the results.

****dfZ***: A pointer to an array of double float values with the source Z values. The conversion result is done in-place and *dfZ will return with the results. If no Z height values are present in the source coordinates, the Z array should be filled with zeros.

iCount : The number of coordinate points to convert.

Note that if a conversion is not successful, ProLatTransform may return a non-zero error. Or, it may return 0 for success but the destination variables hold a very large value. For C/C++ the error value is the standard HUGE_VAL. For Visual Basic, the value will be greater than 1e12 (a one with 12 zeros after it.)

There are several possible reasons for an error. Double check the parameters for the coordinate systems. The two most likely problems encountered are: 1. Coordinates out of range for valid conversion with the defined parameters; and 2. ProLat was unable to find supporting files needed to shift datums, etc. Also, note that ProLat DLL functions are not able to check for valid parameters because of the unlimited possible combinations.

It is highly recommended to test conversion parameters using control points. This involves getting several coordinate points with their known converted values. Test using these control points with ProLat functions to verify the desired conversion is produced.

ProLatClose

void **ProLatClose**(int ***Handle***)

Closes the data structure and frees any allocated memory. It is important to call this function to provide proper cleanup and release memory that was used for the coordinate definition.

ProLatGetErrNo

int **ProLatGetErrNo**()

Returns the last error code set by ProLat DLL functions. Typically this function is only called if any of the ProLat DLL functions return 0 which indicates an error. Pass the result of this function to ProLatStrErr to get an error message.

ProLatStrErr

void **ProLatStrErr**(int ***iErrNo***, char ****sMsg***, int ***iMaxChars***)

iErrNo is the return value from ProLatGetErrNo. It is typically called only after a ProLatDefine…() function returns false.

*sMsg: A pointer to a text string with zero terminator to receive the message.

iMaxChars: The size of the sMsg text buffer. This is so ProLatStrErr does not fill past the end of the buffer area and cause a memory error.

ProLatSetFilePath

void **ProLatSetFilePath**(const char ***sDir***)

Defines the directory where the ProLat support files are located. Normally prolat.dll will be able to find its support files in the same directory as the prolat.dll file. However, in some cases you may wish to place the support files in a different directory. This function can then specify where they are located.

ProLatEllipsoidInverse

int **ProLatEllipsoidInverse** (int ***Handle,*** double ***dfLatFrom,*** double ***dfLonFrom***,
double ***dfLatTo***, double ***dfLonTo***, double ***dfGeodesic***,
double ***dfAzForward***, double ***dfAzBack***, int ***iCount***)

Calculates the geodesic (shortest distance), the Forward Azimuth (angle from North), and the Back Azimuth accurate to the ellipsoid.

Returns 0 for success and non-zero for an error condition.

Handle: The handle returned from a ProLatDefine function. It represents the coordinate system with an xy projection.

**** dfLatFrom***: A pointer to an array of double float values with the starting latitude values.

**** dfLonFrom***: A pointer to an array of double float values with the starting longitude values.

**** dfLatTo***: A pointer to an array of double float values with the destination latitude values.

**** dfLonTo***: A pointer to an array of double float values with the destination longitude values.

**** dfGeodesic***: A pointer to an array of double float to receive the results of the geodesic calculation.

**** dfAzForward***: A pointer to an array of double float to receive the forward azimuth values.

**** dfLatTo***: A pointer to an array of double float to receive the back azimuth values.

iCount : The number of coordinate points to convert.

This function uses the T. Vincenty modified Rainsford's method which is accurate to the ellipsoid. The To and From positions should not be at a geographic pole.

int **ProLatEllipsoidInverse2** (double ***a*** double ***rf***, double ***dfLatFrom,*** double ***dfLonFrom***,
double ***dfLatTo***, double ***dfLonTo***, double ***dfGeodesic***,
double ***dfAzForward***, double ***dfAzBack***, int ***iCount***)

This function takes the ellpsoid parameters 'a' and 'rf' as parameters instead of a ProLat handle. Otherwise it is the same.

ProLatScaleConvergence

int **ProLatScaleConvergence** (int ***Handle,*** double ****dfX,*** double ****dfY***,
double ****dfScale***, double ****dfConvergence***, int ***iCount***)

Calculates the scale factor and convergence angle for the given XY points of a projection against the ideal scale and true north.

Returns 0 for success and non-zero for an error condition.

Handle: The handle returned from a ProLatDefine function. It represents the coordinate system with an xy projection.

**** dfX***: A pointer to an array of double float values with the X (Easting) values.

**** dfY***: A pointer to an array of double float values with the Y (Northing) values.

**** dfScale***: A pointer to an array of double float to receive the results of the scale factor calculation.

**** dfConvergence***: A pointer to an array of double float to receive the convergence angle.

iCount : The number of coordinate points to convert.

This function uses the T. Vincenty modified Rainsford's method which is accurate to the ellipsoid.

ProLatGetDMS

int **ProLatGetDMS** (const char ***sData,***
double ***v1,*** double ***v2***, double ***v3***,
int ***iNSpace,*** int ***piNFound,***
char ***sErrMsg,*** int ***iMsgSize,*** int ***iOrder***)

ProLatGetDMS converts text coordinates to double floating point values suitable for coordinate conversion. It reads most commonly used latitude longitude and decimal formats. See the syntax details below for complete details.

A few common formats are:

102d49'12.81"W 35d15'38.36"N 100.72 (100.72 is an optional height or altitude)
W102°49'12.81" N35°15'38.36" 100.72
W 102 49.2135 N 35 15.63933 100.72
102.820225 W 35.2606556 N 100.72

Returns 0 for success and non-zero for an error condition. When an error occurs the returned coordinates are invalid and the source should be corrected before continuing.

sData: The string containing text coordinates. A string may contain more than one coordinate separated by newline characters. The results are placed in the v1, v2, v3 arrays.

v1, *v2, *v3: Arrays of double floating point. The coordinates found in the sData string will be placed into these arrays. The iOrder parameter determines if ProLatGetDMS detects latitude verses longitude values and in which array they land. For example, iOrder=1 will cause longitude values to go in v1 and latitude values to go in v2 even if the coordinates are written in latitude longitude order. The array v3 receives the optional altitude.

iNSpace: The size of the passed v1, v2, v3 arrays.

piNFound: A pointer to a 4-byte integer returns the number of coordinates that were detected.

sErrMsg: A pointer to a string that will receive an error message if an error is encountered. This may be NULL to avoid returning an error string..

iMsgSize : The size of the sErrMsg string to avoid writing past the end of the string memory.

iOrder: Determines how latitude longitude values are place in arrays v1 or v2.

0: No latitude longitude detection. The first coordinate value is placed in v1, the second in v2, and the third in v3.

1: Place longitude values in v1, latitude in v2, and altitude in v3. Uses heading letters 'E' and 'W' to detect longitudes, and 'N' and 'S' to detect latitude. For example: N10.5 W30.5 would get v1=30.5 v2=10.5 v3=0.0.

Note: Coordinates that have no heading letter will always be placed in v1 and v2 in the order they are detected. It is up to the calling code to determine their order, usually by some user selectable option.

Syntax:

ProLatGetDMS uses pattern recognition to flexibly detect many different coordinate formats. Here is the general syntax specification:

sData contains a list of one or more text coordinates separated by carriage returns or line feeds. The string may contain the following elements:

Comments: Comments are allowed with the following notation:

//	Comment to end of line
#	Comment to end of line
%	Comment to end of line
/* … */	Comment between /* and */

Ignored characters:
space, tab, comma, semicolon, colon,
and space followed by minus and another space ‘ - ‘

Coordinates:
A coordinate has the following format:

DMS DMS [optional altitude]

DMS is a flexible format for degrees minutes seconds and heading. Both DMS values in a coordinate need to have the same format. Possible formats include:

Degrees Minutes Seconds Heading	Degrees Minutes Seconds Heading [altitude]
Degrees Minutes Heading	Degrees Minutes Heading [altitude]
Degrees Heading	Degrees Heading [altitude]
Heading Degrees Minutes Seconds	Heading Degrees Minutes Seconds [altitude]
Heading Degrees Minutes	Heading Degrees Minutes [altitude]
Heading Degrees	Heading Degrees [altitude]
Decimal	Decimal [altitude]

Degrees	unsigned decimal value followed by a space, d, D, or °
Minutes	unsigned decimal value followed by a space, or '
Seconds	unsigned decimal value followed by a space, or "
Heading	the letters E, e, W, w, N, n, S, or s The letters W, w, S, and s will create a negative value. Minus signs are ignored for Degrees Minutes and Seconds.
Decimal	signed decimal value with no heading letter

Note that **commas** are used as separators only. They may not be used as a decimal mark or thousands marker.

Syntax Examples:

80d25’49.12”W 35d41’29”N // USGS common
80 25 49.12 E 35 41 29 S // Degrees Minutes Seconds
35°41’29”N 80°25’49.12”W 100.7 // With altitude
W80d25’49.12” N35d41’29”
80°25.81867’W 35°41.48333’N // Degrees Minutes
35 41.48333 S 80 25.81867 E
W80d25.81867 N35d41.4833
W80 25.81867 N35 41.4833 // Garmin common
80.4303111dW 35.6913889dN // Degrees
80.4303111 W 35.6913889 N
S 35.691389° E 80.4303111°
W80.4303111 N35.691389

-80.4303111 35.691389 // Decimal
500145.387 2457353.25 // Decimal UTM easting, northing

ProLatGetDMS Programming Example:

```
Dim V1[100] as Double
Dim V2[100] as Double
Dim V3[100] as Double
Dim sData as String
Dim sErr as String * 256
Dim iRet as Long
Dim iNDetected as Long

sData = "80 25 49.12 W  35 41 29N" & Chr(13) & _
        "12 25 49.12 E  15 41 29 S  225.4"

iRet = ProLatGetDMS(sData, V1, V2, V3, 100, iNDetected, sErr, 256, 1)

if iRet > 0 then MsgBox sErr

' Returns iNDetected = 2
' V1[1] = -80.430311111    V2[1] = 35.820225  V3[1] = 0.0
' V1[1] = -12.430311111    V2[1] = -15.820225  V3[1] = 225.4
```

ProLatGetDMSSingle

int **ProLatGetDMSSingle** (const char ***sData,*** double ****v1,*** int ****piHeading***,
int ***iNSpace,*** int ****piNFound,*** char ***sErrMsg,*** int ***iMsgSize***)

ProLatGetDMSSingle converts a single value instead of a full coordinate pair.

A few common formats are:

102d49'12.81"W
W 102 49 12.81
117 23.714 E
35.2606556 N

Returns 0 for success and non-zero for an error condition. When an error occurs the returned coordinates are invalid and the source should be corrected before continuing.

sData: The string containing text coordinates. A string may contain more than one coordinate separated by newline characters. The results are placed in the v1, v2, v3 arrays.

v1: Array of double floating point. The value found in the sData string will be placed into this array.

piHeading: Array of 4-byte integers. Indicates the heading value detected: 0 no heading detected, 1 North, 2 South, 3 East, 4 West. Note that when South and West headings are detected the value in v1 is negative.

iNSpace: The size of the passed v1 and piHeading arrays.

piNFound: A pointer to a 4-byte integer returns the number of coordinates that were detected.

sErrMsg: A pointer to a string that will receive an error message if an error is encountered. This may be NULL to avoid returning an error string..

iMsgSize : The size of the sErrMsg string to avoid writing past the end of the string memory.

See the syntax description of ProLatGetDMS for details of the possible text format that may be detected. This function however, just detects the one value instead of a full coordinate pair.

ProLatGetLat

int **ProLatGetLat** (const char ***sPoint,*** double ****Lat***)

This function converts a text latitude to a double floating point value.

Return value is 0 for success, and 1 syntax error, 2 out of range, 3 invalid heading

***** ***sPoint***: The latitude value which may be a DMS formatted value such as N 47 35 11.35.

****Lat***: A double variable pointer that receives the latitude value.

See the syntax description of ProLatGetDMS for details of the possible text format that may be detected. This function however, just detects the one value instead of a full coordinate pair.

ProLatGetLon

int **ProLatGetLon** (const char ***sPoint,*** double ****Lon***)

This function converts a text longitude to a double floating point value.

Return value is 0 for success, and 1 syntax error, 2 out of range, 3 invalid heading

***** ***sPoint***: The longitude value which may be a DMS formatted value such as W 47 35 11.35.

****Lon***: A double variable pointer that receives the longitude value.

See the syntax description of ProLatGetDMS for details of the possible text format that may be detected. This function however, just detects the one value instead of a full coordinate pair.

ProLatDMSFormat

int **ProLatDMSFormat** (char ***sFormat,*** double ****pdfX,*** double ****pdfY***, double ****pdfZ***,
int ***iNCount,*** char * ***sOut***, int ***iOutSpace***)

This function provides flexible Degree Minute Second formatting of coordinate points.

sFormat: The string containing output formatting. See below for the syntax.

pdfX, *pdfY, *pdfZ: Arrays of double floating point. The coordinates points to be formatted.

iNCount: The number of coordinates to format. This is the size of the above arrays.

sOut: A pointer to a text string where the formatted output is to be placed.

iOutSpace : The number of characters of space available in the sOut string.

Returns 0 for success and non-zero for an error.

ProLatDMSFormat provides easy and flexible output formatting features. It provides an Auto format option and a custom format may be entered to get exactly the text format needed.

Examples:

Format:		**Example Output:**		
Auto	Lon Lat Alt for < 180	W81 25.12345	N37 42.12345	0.00
	XYZ for > 180	410500.12	1510100.12	0.00
ED M.MMMM\tND M.MMMM\tZ.ZZ		W81 25.1234	N37 42.1234	0.00
D M S.SSN\tD M S.SSE		35 12 7.12N	91 27 54.12W	
X.XX Y.YY		410578.12 1500100.12		
\X: X.XX\t\Y: Y.YY\t\Z: Z.ZZ		X: 410578.12	Y: 1500100.12	Z: 300.12

Output Format Syntax:

Code	**Generates**
XX.XX	X value without heading letter. Number of X's before and after the period control space padding and precision.
YY.YY	Y value without heading letter Number of Y's before and after the period control space padding and precision.
ZZ.ZZ	Z value without heading letter Number of Z's before and after the period control space padding and precision.
E or O	Heading letter for Longitude. Outputs ‘E’ for eastern and ‘W’ for western (negative).

	Use O to identify the value as longitude without printing a heading letter.
N or A	Heading letter for Latitude. Outputs 'N' for northern and 'S' for southern (negative). Use A to identify the value as the latitude without printing a heading letter.
D.DD	Degrees. Number of D's before and after the period control zero padding and precision. D without decimal provides unsigned integer degrees. For decimal degrees without a heading letter, use X.XX or Y.YY. If D.DD is used without a heading letter (E or N) Longitude-Latitude order is assumed.
M.MM	Minutes. Number of M's before and after the period control zero padding and precision. M without decimal provides unsigned integer minutes.
S.SS	Seconds. Number of S's before and after the period control zero padding and precision.
\c	Special character. Example to insert a D character use \D, for tab use \t, new line use \n
c	Characters may be inserted without a \ if they don't conflict with format characters. Examples: Any lowercase letter, space, comma, colon, semicolon, etc.

Helper functions:

In special cases it may be desirable to parse the format string once and produce one formatted output at a time. ProLatDMSFormat is just as efficient and is the recommended solution, so only use the following functions if you know they are necessary.

```
int ProLatDMSFormatParse(const char *sFormat, int *FormatArray);
```

This function requires an empty integer array of at least 100 in length for the function to store a parsed format. The FormatArray is then passed to the following function to efficient

```
int ProLatDMSFormat1(int *FormatArray, double dfX, double dfY, double dfZ,
                     char *sOut, int iOutSpace);
```

This function formats one output value at a time using a FormatArray produced by ProLatDMSFormatParse.

Reading and Writing PRJ and WKT files

Three functions are available to process PRJ and WKT (Well Known Text) files. ProLat treats both file types the same. The functions take a string of one type and converts it. It is necessary to provide the code to read and write the files.

ProLatConvertPRJToStr	Converts a PRJ/WKT/ string to a ProLat custom parameter string.
ProLatConvertStrToESRI	Converts a ProLat custom string to a PRJ string.
ProLatConvertStrToWKT	Converts a ProLat custom string to WKT (Well Known Text)
ProLatConvertHandleToStr	Converts a ProLat coordinate system handle to a custom string.

A typical approach would be to use ProLatConvertPRJToStr to produce a string that may be used with ProLatCustomAdd and ProLatDefineCustom to get a ProLat coordinate system handle. This handle may then be used with ProLatTransform to convert coordinates.

ProLatConvertPRJToStr

int **ProLatConvertPRJToStr** (const char ***sPRJ**,* char ****sOutCust**,* int ***iOutLen***)

sPRJ: A string containing a PRJ or WKT coordinate system.

sOutCust: A string that receives the coordinate system converted to ProLat custom format.

iOutLen: The length of allocated space in bytes reserved for the sOutCust string.

Return value: Returns 0 for success and non-zero for an error.

This function attempts to convert common PRJ and WKT coordinate system definitions into the ProLat custom format which may be used with the ProLatCustomAdd and ProLatCustomDefine functions. The function needs to be used with care and the user should very that parameters are converted successfully.

ProLatConvertStrToESRI

int **ProLatConvertStrToESRI** (const char ***sCustStr**,* char ****sESRI**,* int ***iOutLen***)

sCustStr: A string containing a ProLat custom coordinate system string.

**** sESRI***: A string that receives the coordinate system converted to ESRI PRJ format.

iOutLen: The length of allocated space in bytes reserved for the sESRI string.

Return value: Returns 0 for success and non-zero for an error.

This function attempts to convert a ProLat custom string to the ESRI PRJ format. The function needs to be used with care and the user should very that parameters are converted successfully.

ProLatConvertStrToWKT

int **ProLatConvertStrToWKT**(const char ***sCustStr**,* char ****sWKT**,* int ***iOutLen***)

sCustStr: A string containing a ProLat custom coordinate system string.

**** sWKT***: A string that receives the coordinate system converted to WKT PRJ format.

iOutLen: The length of allocated space in bytes reserved for the sESRI string.

Return value: Returns 0 for success and non-zero for an error.

This function attempts to convert a ProLat custom string to the ESRI PRJ format. The function needs to be used with care and the user should very that parameters are converted successfully.

Currently support is not provided for State Plane Coordinate Systems using the “+init=NAD27” or “+init=NAD83”.

This function is different than the ProLatConvertStrToESRI function because it does not apply ESRI specific conventions to the coordinate system.

ProLatConvertHandleToStr

int **ProLatConvertHandleToStr** (int ***Handle***, char ****sCustStr***, int ***iOutLen***)

Handle: ProLat handle.

**** sCustStr***: A string that receives the ProLat custom string definition.

****iOutLen***: The length of allocated space in bytes reserved for the sCustStr string.

Return value: Returns 0 for success and non-zero for an error.

This function attempts to extract the ProLat custom string defintion for the coordinate system defined in the Handle. If for some reason the coordinate system cannot be represented by a string, a value of 1 is returned.

ProLatX ActiveX DLL

The ProLat DLL functions are provided in a convenient ActiveX DLL for use by Visual Basic and VBScript applications with ASP (Active Server Pages). The files are located in examples\ProLatX.

It works only on the server side because the ProLatX ActiveX DLL is a wrapper for the standard ProLat Windows DLL.

To use ProLatX with ASP VBScript, you will need prolatx.dll, prolat.dll and the other support files depending on type of conversions needed. You must also be familiar with ASP programming.

From a DOS shell prompt run REGSVR32 PROLATX.DLL to register it.

ProLatX is an apartment threaded ActiveX DLL. To make it work in Visual Basic it is required to use the Project -> References and check the ProLatX option. After this is done you can create an object from it using one of two ways.

Method 1: Set a global variable of type ProLat such as:

```
Private mProLat As ProLat
```

Then use the New operator to create the object:

```
Private Sub Form_Load()
  Set mProLat = New ProLat
End Sub

Private Sub Form_Unload(Cancel As Integer)
  Set mProLat = Nothing
End Sub
```

Method 2: Use the CreateObject method. Note this still requires the ProLatX to be selected in the References list.

```
Set oProLat = Server.CreateObject("ProlatX.Prolat")
```

List of ProLatX Methods and Properties

ProLatX Method	Equivalent ProLat DLL Function
DefineLatLong	ProLatDefineLatLong
DefineUTM	ProLatDefineUTM
DefineSPCS	ProLatDefineSPCS
DefineGeocentric	ProLatDefineGeocentric
DefineFromFile	ProLatDefineFromFile
CustomAdd	ProLatCustomAdd
DefineCustom	ProLatDefineCustom
Transform (Use properties X,Y,Z and TransformInternal for ASP)	ProLatTransform
CloseDefinition	ProLatClose
GetErrNo	ProLatGetErrNo
StrErr	ProLatStrErr
TransformInternal *	
GetDMS	ProLatGetDMS
X (property)	
Y (property)	
Z (property)	

* Note that TransformInternal performs the transform on X,Y, and Z properties. The result will be these properties being in the destination coordinate system.

ProLatX source code is included in the file ProLatX.vbp and ProLat.cls. An example standard VB application is provided in PLCalcX.vbp.

Creating DMS Values From Decimal Values

For readability, DMS (Degrees Minutes Seconds) format is commonly used. Use the functions in the preceding section to format DMS output, or use the following steps to manually convert the decimal latitude/longitude results into DMS format.

1. Get the heading from the sign of the value. For longitudes, a negative number is West and a positive number is East. For Latitudes, a negative number is South and a positive number is North.
2. Convert negative values to be positive, because all DMS values are positive with heading shown by a direction letter such as W, E, N, S
3. Get degrees by taking the integer part of the decimal value.
   ```
   Degrees = Int(Value)
   ```
4. Get minutes by taking the decimal part and multiplying by 60.
   ```
   Minutes = (Value – Int(Value)) * 60.0
   ```
   This gives minutes with a decimal fraction. Stop here if you just need degrees with fractional minutes. Go to Step 5 to get seconds.
5. Get seconds by taking the decimal part of the minutes and multiplying by 60.
   ```
   Seconds = (Minutes – Int(Minutes)) * 60.0
   ```
   Remove the decimal portion from the minutes
   ```
   Minutes = Int(Minutes)
   ```

DMS values are formatted in many ways. There does not seem to be a universal standard. We recommend using pure decimal values when possible because they are easily readable by code. Decimal does not include heading letters, so it is necessary to document which value is longitude and which value is latitude.

Decimal format lon/lat:	-98.185741 35.821771
Decimal with heading:	W98.185741 N35.821771
Heading Degrees Minutes:	W98 11.14446 N35 49.30626
Degrees Minutes Seconds Heading:	98d11'8.6676"W 35d49'18.3756"N

Custom Coordinate Systems

It is possible to define custom coordinate systems using a Proj.4 compatible definition string with the ProLat CoordSys.GetCS() function. See below for parameter details.

Example Coordinate System Definition

To create a custom coordinate system using a parameter list, it is necessary to know something about the selected projection's details. It may be necessary to refer to other literature to get complete details and usage of a projection. However, it is sometimes possible to select parameters based on the coordinate specifications.

For example, you receive a set of coordinates that have the following specs:

> Lambert Conformal Conic, WGS84 datum, Central meridian of W122 degrees, latitudes of intersection at N41d40 and N39d20, central latitude at N39d20, with false easting of 2,000,000 and false northing of 500,000.

After looking through the available parameters in the list below, it is possible to determine the parameters and create the following string for use with CoordSys.GetCS():

```
proj=lcc datum=NAD83 lon_0=-122 lat_1=41d40 lat_2=40 lat_0=39d20
x_0=2000000 y_0=500000
```

Required Parameters

A coordinate system requires `proj=` parameter. It also requires either `datum=` parameter or else a definition of the ellipse along with a `nadgrids=` or or a `towgs84=` parameter which tells ProLat how to convert to the WGS84 datum. It also needs projection parameters. Most projections use `lon_0=` for the central meridian and `lat_0=` for the central parallel. All Cartesian projections allow x_0= and y_0= to provide false easting and northing. See the projection descriptions later in this manual for special parameters needed by some projections.

Parameter List

The following parameters and usage varies with the projection selected. The options are processed in left to right order. Reentry of an option is ignored with the first occurrence assumed to be the desired value.

No spaces may be placed around the equal sign. A parameter without an equal sign shown below will activate that option without requiring additional parameter information. A plus sign before the parameter is optional.

proj=name Required for selection of the transformation, and name is from the list of available projections. See Projection Descriptions later in this section for additional parameter information for selected projections.

Example: `proj=tmerc`

Available Projection Names

For detailed parameter information see the section on Projection Descriptions. Please contact Effective Objects if a projection is needed that is not listed here.

aea	Albers Equal Area
aeqd	Azimuthal Equidistant
airy	Airy
aitoff	Aitoff
eck3	Eckert III
eck6	Eckert VI
geocent	Geocentric, ECEF, XYZ
gn_sinu	General Sinusoidal Series
kav7	Kavraisky VII
krovak	Krovak projection
latlong	Latitude/Longitude (non-projected) (also latlon, longlat, lonlat)
lcc	Lambert Conformal Conic
leac	Lambert Equal Area
lsat	Space oblique for LANDSAT
mbtfps	McBryde-Thomas Flat-Polar Sinusoidal
moll	Mollweide
nzmg	New Zealand Map Grid
omerc	Oblique Mercator
ortho	Orthographic
poly	Polyconic (American)
putp1	Putnins P1
sinu	Sinusoidal (Sanson-Flamsteed)
somerc	Swiss Oblique Mercator
stere	Stereographic
sterea	Oblique Stereographic Alternative
tmerc	Transverse Mercator
ups	Universal Polar Stereographic
urmfps	Urmaev Flat-Polar Sinusoidal
utm	Universal Transverse Mercator
vandg	van der Grinten I
vandg2	van der Grinten II
vandg3	van der Grinten III
vandg4	van der Grinten IV
wag1	Wagner I (Kavraisky VI)
wag2	Wagner II
wag3	Wagner III
wag4	Wagner IV
wag5	Wagner V
wag6	Wagner VI
wintry	Winkel Tripel

+ellps=name The +ellps option allows selection of standard, predefined ellipsoid figures. This parameter is required if the +datum parameter is not used. For spherical only projections, the major axis is used as the radius. Alternatively, it is possible to specify the ellipse with the `a=` and `rf=` parameters.

Available Ellipsoids

MERIT	a=6378137.0	rf=298.257	MERIT 1983
SGS85	a=6378136.0	rf=298.257	Soviet Geodetic System 85
GRS80	a=6378137.0	rf=298.257222101	GRS 1980(IUGG, 1980)
IAU76	a=6378140.0	rf=298.257	IAU 1976
airy	a=6377563.396	b=6356256.910	Airy 1830
APL4.9	a=6378137.0.	rf=298.25	Appl. Physics. 1965
NWL9D	a=6378145.0.	rf=298.25	Naval Weapons Lab., 1965
mod_airy	a=6377340.189	b=6356034.446	Modified Airy
andrae	a=6377104.43	rf=300.0	Andrae 1876 (Den., Iclnd.)
aust_SA	a=6378160.0	rf=298.25	Australian Natl & S. Amer. 1969
GRS67	a=6378160.0	rf=298.2471674270	GRS 67(IUGG 1967)
bessel	a=6377397.155	rf=299.1528128	Bessel 1841
bess_nam	a=6377483.865	rf=299.1528128	Bessel 1841 (Namibia)

clrk66	a=6378206.4	b=6356583.8	Clarke 1866
clrk80	a=6378249.145	rf=293.4663	Clarke 1880 mod.
CPM	a=6375738.7	rf=334.29	Comm. des Poids et Mesures 1799
delmbr	a=6376428.	rf=311.5	Delambre 1810 (Belgium)
engelis	a=6378136.05	rf=298.2566	Engelis 1985
evrst30	a=6377276.345	rf=300.8017	Everest 1830
evrst48	a=6377304.063	rf=300.8017	Everest 1948
evrst56	a=6377301.243	rf=300.8017	Everest 1956
evrst69	a=6377295.664	rf=300.8017	Everest 1969
evrstSS	a=6377298.556	rf=300.8017	Everest (Sabah & Sarawak)
fschr60	a=6378166.	rf=298.3	Fischer (Mercury Datum) 1960
fschr60m	a=6378155.	rf=298.3	Modified Fischer 1960
fschr68	a=6378150.	rf=298.3	Fischer 1968
helmert	a=6378200.	rf=298.3	Helmert 1906
hough	a=6378270.0	rf=297.	Hough
intl	a=6378388.0	rf=297.	International 1909 (Hayford)
krass	a=6378245.0	rf=298.3	Krassovsky, 1942
kaula	a=6378163.	rf=298.24	Kaula 1961
lerch	a=6378139.	rf=298.257	Lerch 1979
mprts	a=6397300.	rf=191.	Maupertius 1738
new_intl	a=6378157.5	b=6356772.2	New International 1967
plessis	a=6376523.	b=6355863.	Plessis 1817 (France)
SEasia	a=6378155.0	b=6356773.3205	Southeast Asia
walbeck	a=6376896.0	b=6355834.8467	Walbeck
WGS60	a=6378165.0	rf=298.3	WGS 60
WGS66	a=6378145.0	rf=298.25	WGS 66
WGS72	a=6378135.0	rf=298.26	WGS 72
WGS84	a=6378137.0	rf=298.257223563	WGS 84
sphere	a=6370997.0	b=6370997.0	Normal Sphere (r=6370997)

+datum=name Allows selection of a standard predefined datum name. The supported datum names are shown below. If +datum is not used, it is required to specify the `+ellps=` parameter, and if necessary the `+nadgrids=` or `+towgs84=` parameters.

name	**ellipse**	**definition/comments**
WGS84	WGS84	towgs84=0,0,0
GGRS87	GRS80	towgs84=-199.87,74.79,246.62 Greek_Geodetic_Reference_System_1987
NAD83	GRS80	towgs84=0,0,0 North_American_Datum_1983
NAD27	clrk66	nadgrids=@conus,@alaska,@ntv2_0.gsb,@ntv1_can.dat North_American_Datum_1927
potsdam	bessel	towgs84=606.0,23.0,413.0 Potsdam Rauenberg 1950 DHDN
carthage	clark80	towgs84=-263.0,6.0,431.0 Carthage 1934 Tunisia
hermannskogel	bessel	towgs84=653.0,-212.0,449.0 Hermannskogel
ire65	mod_airy	towgs84=482.530,-130.596,564.557,-1.042,-0.214,-0.631,8.15 Ireland 1965
nzgd49	intl	towgs84=59.47,-5.04,187.44,0.47,-0.1,1.024,-4.5993 New Zealand Geodetic Datum 1949

OSGB36	airy	towgs84=446.448,-125.157,542.060,0.1502,0.2470,0.8421,-20.4894 Airy 1830

+x_0= False easting is added to x value of the Cartesian coordinate. May be used in most projections except longlat and geocent coordinates.

+y_0= False northing is added to y value of the Cartesian coordinate. See +x_0.

+lon_0= Central meridian. Along with +lat_0, normally determines the geographic origin of the projection.

+lat_0= Central parallel. See +lon_0.

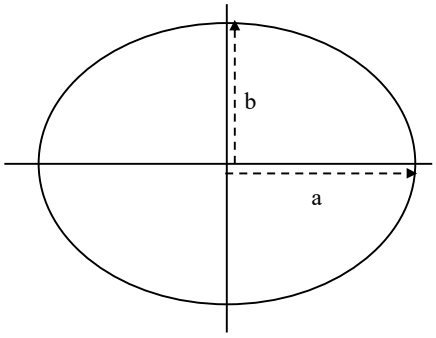
+k or **+k_0=** Scale factor at the central meridian. The default value is 1.

+a= Specifies an elliptical Earth's major axis *a*.

+b= Specifies an elliptical Earth's minor axis *b*.

+es= Defines the elliptical Earth's squared eccentricity, e^2. Optionally, either b= (minor axis), e= (eccentricity), rf=1/f (reciprocal flatten), or f= (flattening) may be used.

$e^2 = (a^2 - b^2) / a^2$



+e= Eccentricity.

+**f**= Flattening. f = (a – b) / a

+**rf**= Reciprocal Flattening. rf = 1/f

+**pm**= Prime meridian relative to Greenwich.

+R= Specifies that the projection should be computed as a spherical Earth with radius *R*. This parameter takes precedence over the elliptical parameters.

The following radius parameters are used with elliptical Earth parameters. They allow projections to be computed in the spherical form when specified. For projections that only perform computations for a sphere, this method is preferable to the default of using the major axis as the radius.

+R_A Determines that spherical computations be used with radius of a sphere that has a surface area equivalent to the selected ellipsoid.

+R_V Used with elliptical Earth parameters. Radius of a sphere with equivalent volume of specified ellipse.

+R_a Used with elliptical Earth parameters. Spherical radius of the arithmetic mean of the major and minor axis is used. R_a = (a+b)/2

+R_g Used with elliptical Earth parameters. Geometric mean of the major and minor axis, $R_g = (ab)^{1/2}$

+R_h Used with elliptical Earth parameters. Harmonic mean of the major and minor axis, R_h = 2ab/(a+b)

+R_lat_a= Used with elliptical Earth parameters. Spherical radius of the arithmetic mean of the principle radii of the ellipsoid at latitude R_lat_a is used. +R_lat_g = R_lat_a can be use for equivalent geometric mean of the principle radii.

+R_lat_g= Used with elliptical Earth parameters. Geometric mean of the principle radii at latitude R_lat_g. See R_lat_a.

+units=name Selects conversion of Cartesian values to units specified by name. When used, other + metric parameters must be in meters.

Example: +units=us-ft

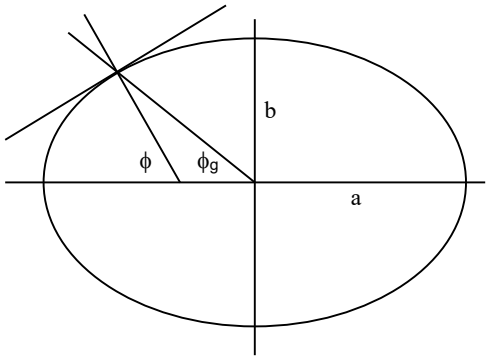
Unit Parameter	Conversion from meters	Description
km	1000.	Kilometer
m	1.0	Meter
dm	1/10	Decimeter
cm	1/100	Centimeter
mm	1/1000	Millimeter
kmi	1852.0	International Nautical Mile
in	0.0254	International Inch
ft	0.3048	International Foot
yd	0.9144	International Yard
mi	1609.344	International Statute Mile
fath	1.8288	International Fathom
ch	20.1168	International Chain
link	0.201168	International Link
us-in	1.0/39.37	U.S. Surveyor’s Inch
us-ft	0.304800609601219	U.S. Surveyor’s Foot
us-yd	0.914401828803658	U.S. Surveyor’s Yard
us-ch	20.11684023368047	U.S. Surveyor’s Chain
us-mi	1609.347218694437	U.S. Surveyor's Statute Mile
ind-yd	0.91439523	Indian Yard
ind-ft	0.30479841	Indian Foot
ind-ch	20.11669506	Indian Chain

+geoc When this option is present, it treats the latitude angle of the *other* coordinate system as geocentric instead of the normal geodetic. In the diagram below, ϕ_g is the geocentric latitude relative to the center of the earth, and ϕ is the commonly used geodetic latitude relative to the tangent line at the earth's surface at that location.

It is important to remember that this option treats the *other* coordinate system as having a geocentric latitude. This may not make sense for some coordinate systems, so care is required.

The two latitudes are related by:

$$\tan \phi_g = \frac{b^2}{a^2} \tan \phi$$



+over Inhibit reduction of input longitude values to a range within +/-180degrees of the central meridian.

+towgs84= Datum shifts can be approximated by 3 parameter spatial translations (in geocentric xyz space), 7 parameter shifts (translation + rotation + scaling), and 10 parameter shifts (translation + rotation + scaling + point of rotation).

In the **3 parameter** case, the three arguments are the translations to the geocentric location in meters. For example the EPSG database uses the following 3 parameter towgs84 for the Greek GGRS87 datum to WGS84.

```
+towgs84=-199.87,74.79,246.62
```

A **7 parameter** example from the EPSG database is used for transforming from WGS72 to WGS84.

```
+towgs84=0,0,4.5,0,0,0.554,0.219
```

Effective Objects provides the following information from the USGS sources. You may need to refer to additional resources to get complete details in creating +towgs84 parameters.

The seven parameter case uses *delta_x*, *delta_y*, *delta_z*, *Rx - rotation X*, *Ry - rotation Y*, *Rz - rotation Z*, *M_BF - Scaling*. The three translation parameters are in meters as in the three parameter case. The scaling is the scale change in parts per million. For complete details see EPSG transformation method (trf_method's 9603 and 9606).

In ProLat, the following calculations are used to apply the **towgs84** transformation (going to WGS84). The x, y and z coordinate arrays are in geocentric coordinates. The 7 towgs84 parameters are stored in the array towgs84[]

Three parameter transformation (simple offsets):

```
x[i] = x[i] + towgs84[0];
y[i] = y[i] + towgs84[1];
z[i] = z[i] + towgs84[2];
```

Seven parameter transformation (translation, rotation and scaling):

```
Rx_BF = towgs84[3]; Ry_BF = towgs84[4];
Rz_BF = towgs84[5]; M_BF  = towgs84[6];

x_out = M_BF*(       x[i] - Rz_BF*y[i] + Ry_BF*z[i]) + towgs84[0];
y_out = M_BF*( Rz_BF*x[i] +       y[i] - Rx_BF*z[i]) + towgs84[1];
z_out = M_BF*(-Ry_BF*x[i] + Rx_BF*y[i] +       z[i]) + towgs84[2];
```

Note that EPSG method 9607 (coordinate frame rotation) coefficients can be converted to EPSG method 9606 (position vector 7-parameter) supported by ProLat by reversing the sign of the rotation vectors. The methods are otherwise the same.

A **10 parameter** Molodenski-Badekas example from EPSG Guidance Note number 7, part 2 is used for transforming from La Canoa:

+towgs84= -270.933,115.599,-360.226-5.266,1.238,-2.381,-5.109,2464351.59,-5783466.61,974809.81

+nadgrids=file Specify a grid file or list of files to use in shifting a coordinate from a datum to WGS84.

The convention in ProLat is for a grid file to shift from some datum such as NAD27 to the NAD83/WGS84 datum. This provides a convenient standard to allow converting between any coordinate system to another coordinate system, because WGS84 is the common intermediate datum.

Use of grid shifts is specified using the "nadgrids" keyword in a coordinate system definition.

```
+nadgrids=ntv1_can.dat
```

In this case the ntv1_can.dat grid shift file is loaded, and used to get a grid shift value for the selected point.

When +nadgrids is used, the parameter +datum should not be used. Instead, use the +ellps parameter, such as +ellps=clrk66. ProLat will use the +nadgrids option for shifting coordinates, and the +ellps parameter just lets the system know it is a different datum in order to activate the grid shift file.

It is possible to list multiple grid shift files, in which case each will be tried in turn till one is found that contains the point being transformed.

```
+nadgrids=conus,alaska,ntv1_can.dat,hawaii,stgeorge,stlrnc,stpaul
```

Important: Where grids overlap (such as conus and ntv1_can.dat for instance) the first valid file found for a point will be used regardless of whether it is appropriate or not. So, for instance, +nadgrids=ntv1_can.dat,conus would result in the Canadian data being used for some areas in the northern United States even though the conus data is the approved data to use for the area. Careful selection of files and file order is necessary. In some cases border spanning datasets may need to be pre-segmented into Canadian and American points so they can be properly grid shifted.

Skipping Missing Grids

The special prefix **@** may be prefixed to a grid to make it optional. If it not found, the search will continue to the next grid. Normally any grid not found will cause an error. For instance, the following would use the ntv2_0.gsb file if available, otherwise it would fallback to using the ntv1_can.dat file.

```
+nadgrids=@ntv2_0.gsb,ntv1_can.dat
```

Available Grids

		Extent			
Region	**Parameter**	**East**	**West**	**South**	**North**
Conterminous U.S.	conus.ncn	131° W	63° W	20° N	50° N
Alaska	alaska.ncn	194° W	128° W	46° N	77° N
Hawaii	hawaii.ncn	161° W	154° W	18° N	23° N
Puerto Rico and Virgin Islands	prvi.ncn	68° W	64° W	17° N	19° N
St. George Is., AK	stgeorge.ncn	171° W	169° W	56° N	57° N
St. Lawrence Is., AK	stlrnc.ncn	172° W	168° W	62° N	64° N
St. Paul Is., AK	stpaul.ncn	171° W	169° W	57° N	58° N
Canada	ntv1_can.dat				

An improved Canadian grid shift file (NTV2_0.GSB) is available for free download from the NRCan web site at http://www.geod.nrcan.gc.ca/index_e/products_e/software_e/ntv2_e.html. Save it in the directory with the other ProLat support files. Use "+nadgrids=NTV2.GSB" to access it.

HARN/HPGN grid files are included with ProLat. See the HARN and HPGN section below for complete details. Use CoordSys.GetDatums() to get a complete list of datums, which include the HARN datum definitions.

Projection Descriptions

A brief description of selected projections and their parameters are provided in the table below.[3] See parameter descriptions above for additional parameter information.

All projections require a parameter to specify the ellipsoid such as **+datum=**, **+ellps=**, or **+a**= and **+rf**=.

Projection	Parameters	Description

tmerc: Transverse Mercator	`+proj=tmerc`
	`+lon_0=`
	`+lat_0=`
	`+k=`
		A common projection for large scale maps oriented in north-south strips. The parameter k is the scale factor at the central meridian, lat_0. x_0 and y_0 are commonly supplied for false easting and northing so that the values are not negative.  tmerc with lon_0=90 and 15° grid

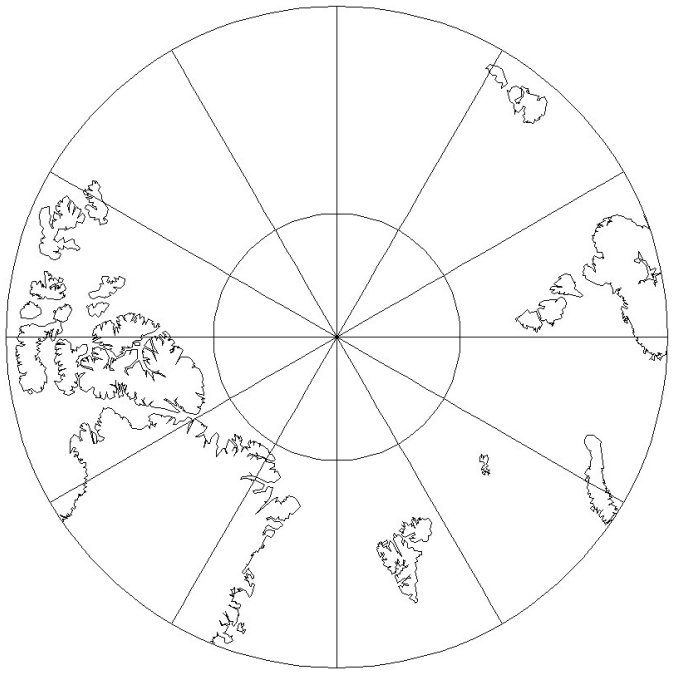
utm: Universal Transverse Mercator	`+proj=utm`
	`+zone=`
	`+south`
		See discussion of UTM earlier in this manual. For regions below the equator use the +south without an = sign.

omerc: Oblique Mercator Projection	`+proj=omerc`
	`+k=`
	`+lat_0=`
	`+no_rot`
	`and either`
	`  +lon_1=`
	`  +lat_1=`
	`  +lon_2=`
	`  +lat_2=`
	`or`
	`  +alpha=`
	`  +lonc=`
	`  [+gamma=]`
		There are three methods to specify the parameters.
		1. With two points (lon_1, lat_1) and (lon_2, lat_2) which will determine a great circle, central line through each point, or
		2. With a point of origin at (lonc, lat_0) and an azimuth alpha, measured clock-wise from north, of the central line of the projection.
		3. A gamma= option may be added to option 2. to get the Rectified Skew Orthomorphic (or Hotine Oblique Mercator).
		The presence of +alpha= determines which method is used. The Cartesian coordinates are rotated by –alpha unless +no_rot is used.
		Initialization will fail if control parameters nearly define a transverse or normal (equatorial) Mercator projection.

lcc: Lambert Conformal Conic	`+proj=lcc`
	`+lon_0=`
	`+lat_0=`
	`Secant`
	`  +lat_1=`
	`  +lat_2=`
	`Tangent`
	`  +lat_1=`
	`  +k_0=`
		A common projection for large scale maps oriented in east-west strips. For Secant method, lat_1 and lat_2 are the latitudes of intersection of the cone with the ellipsoid or sphere. For the Tangent method, lat_1 is the latitude of tangency of the cone with the ellipsoid or sphere.

lsat: LANDSAT	`+proj=lsat`
	`+lsat=`
	`+path=`
		This projection is for use with LANDSAT satellite data and is a limited form of the more general Space Oblique Mercator projection. The LANDSAT satellite number, lsat, must be in the range 1-5, and the path number, path, must be in the ranges 1-251 for lsat=1,2,3, or 1-233 for lsat=3,4. This is the projection coded by John P. Snyder that started his career with the USGS.

merc: Mercator	+proj=merc +lat_ts=	Used for equatorial regions. +lat_ts is the latitude of true scale.
ups: Universal Polar Stereographic	`+proj=ups` `+south`	The UPS projection is a special case polar aspect of the Stereographic projection designed to cover the regions where latitude ≥ 84°N or ≤ 80°S. The internal Stereographic parameters are fixed at k=0.994, lon_0=0, x_0 = y_0 = 2,000,000m, and lat_0 is either 90°N or 90°S when +south is specified. Also see the UTM projection.



ups projection from 74°N to 90°N. Note that center circle of 84°N represents the valid area.

airy: Airy	`+proj=airy +lat_b= +no_cut`	The Airy projection is an azimuthal minimum error projection for the region within the small or great circle defined by an angular distance, lat_b, from the tangency point of the plane (lon_0, lat_0). The default value for lat_b is 90° that is suitable for hemispherical maps. Extent of projection is limited to the hemisphere unless +no_cut is specified.
krovak	+proj=krovak +ellps=bessel	This projection requires few parameters. Results are negative with southing in x and westing in y. To get positive values with westing in x and southing in y, use +krovakrevert.

HARN and HPGN

High Accuracy Reference Network (HARN) and High Precision Geodetic Network (HPGN) are designations used for a statewide geodetic network upgrade. The acronyms HARN and HPGN refer to the same thing. HARN has been adopted as the official name to reduce confusion. A HARN is a statewide or regional upgrade in accuracy of NAD 83 coordinates using Global Positioning System (GPS) observations.

ProLat uses NADCON tables from the United States National Geodetic Survey (NGS) to convert NAD83 coordinates to HARN coordinates. To convert NAD83 to Kansas HARN, use ProLatDefineLatLong() as shown in the following example:

To define a Harn/HPGN coordinate system, use CoordSys.GetCS() with a datum containing the Harn description. Use CoordSys.GetDatums() for a complete list of available dataums.

```
harnAR = CoordSys.GetCS("LAT_LONG", "LAT-LONG", "NAD83-ARKANSAS-HARN", "METERS");

or, using Proj.4

harnAR = CoordSys.GetCS("proj=latlong ellps=GRS80 nadgrids=arharn.hrn");
```

To convert a different coordinate system such as NAD27, State Plane, etc. to HARN, it is only necessary to define that coordinate system and ProLatTransform will convert in one step. For example:

```
Nad27 = CoordSys.GetCS("LAT_LONG", "LAT-LONG", "NAD27", "METERS");
HarnAR = CoordSys.GetCS("LAT_LONG", "LAT-LONG", "NAD83-ARKANSAS-HARN", "METERS");
CoordSys.Transform(Nad27, HarnAR, Lon, Lat, Z, 1);
```

High Accuracy Reference Network Grid Files

ProLat HARN/HPGN grid shift files use the format **xxharn.ncn**. xx indicates the region. The .ncn extension indicates a nadcon grid shift file. These tables were prepared directly from NADCON data and converted to shift to the NAD83 datum to match the ProLat standard of all grid files shifting to WGS84/NAD83.

Area/State	File Name	Notes	Extents			
			East °W	West °W	South °N	North °N
Alabama	alharn.ncn		84	90	30	36
Arkansas	arharn.ncn		89	95	32	37
Arizona	azharn.ncn		108	116	30	38
California (North)	cnharn.ncn	Above 37 degrees latitude	114	125	36	43
California (South)	csharn.ncn	Below 37 degrees latitude	113	122	32	37
Colorado	coharn.ncn		101	110	36	42
Florida	flharn.ncn		80	88	24	32
Georgia	gaharn.ncn		80	86	30	36
Guam *	guharn.ncn		213	219	13	19
Hawaii	hiharn.ncn		154	162	18	24

Idaho-Montana (East)	emharn.ncn	East of 113 degrees longitude	103	113	41	50
Idaho-Montana (West)	wmharn.ncn	West of 113 degrees longitude	109	119	41	50
Iowa	iaharn.ncn		89	98	40	44
Illinois	ilharn.ncn		85	92	36	43
Indiana	inharn.ncn		81	89	37	46
Kansas	ksharn.ncn		94	103	36	41
Kentucky	kyharn.ncn		81	90	36	40
Louisiana	laharn.ncn		88	95	27	34
Maryland – Delaware	mdharn.ncn		74	80	37	41
Maine	meharn.ncn		66	72	42	48
Michigan	miharn.ncn		82	91	41	48
Minnesota	mnharn.ncn		88	98	43	50
Mississippi	msharn.ncn		86	92	29	36
Missouri	moharn.ncn		88	97	35	42
Nebraska	nbharn.ncn		95	105	40	44
Nevada	nvharn.ncn		114	121	35	43
New England	neharn.ncn	CT, MA, NH, RI, VT	69	75	40	46
New Jersey	njharn.ncn		70	76	38	44
New Mexico	nmharn.ncn		101	110	31	38
New York	nyharn.ncn		70	81	40	46
North Dakota	ndharn.ncn		95	105	45	50
Ohio	ohharn.ncn		80	86	38	43
Oklahoma	okharn.ncn		94	104	33	38
Pennsylvania	paharn.ncn		74	82	39	44
Puerto Rico-Virgin Is	pvharn.ncn		62	68	17	21
Samoa * (Eastern Is)	esharn.ncn	Islands of Ofu, Olosega, Ta'u	165	171	14	20
Samoa * (Western Is)	wsharn.ncn	Islands of Tutuila and Aunu'u	165	171	14	20
South Dakota	sdharn.ncn		95	105	41	47
Tennessee	tnharn.ncn		81	91	34	37
Texas (East)	etharn.ncn	East of 100 degrees longitude	88	100	25	35
Texas (West)	wtharn.ncn	West of 100 degrees longitude	99	107	25	37
Utah	utharn.ncn		107	115	36	43
Virginia	vaharn.ncn		75	84	36	40
Washington – Oregon	woharn.ncn		116	125	41	50
West Virginia	wvharn.ncn		77	84	36	41
Wisconsin	wiharn.ncn		86	94	42	48
Wyoming	wyharn.ncn		104	112	41	46

* Guam and American Samoa never went through the intermediate step of island datum to NAD83. Those islands were adjusted directly from their old island datums (Guam 1963 and American Samoa 1962) to HPGN. Consequently, positions computed on the island datums are considered to be NAD83 for the input/output purposes.

There are no HARN grid files for Alaska, North Carolina, and South Carolina.

Troubleshooting

Commonly encountered problems and solutions:

Problem		Solution
ProLatNet.dll was not found.		The file prolatnet.dll was not found by the application. The file prolatnet.dll should be located in the same directory as the applications executable file or in sub-folder. The file ProLatNet.dll does not need to be registered. It is generally found relative to the executable.
Error: -38 Failed to load NAD27-83 correction file.		The conversion requires a datum shift and ProLat was not able to find a NADCON grid file to perform the shift. There are two common reasons for this error. 1. The NADCON grid shift files could not be found. 2. The given coordinate point is outside the boundaries of the NADCON grid shift files. Check the coordinates, units, parameters, and lat-long / long-lat ordering. A negative value is needed for West and South directions. If using GetDMS, check that the proper coordinate syntax is used.
The return values are infinite (maximum possible value)		The algorithms detected a problem and returned the maximum possible double float value so that the results would not be inadvertently used. Check the ProLatErrNo and ProLatErrMsg for additional details. Check the parameters, units, and lat-lon / lon-lat order. A negative value is needed for West and South directions.
The values are close but a little off from my reference control points.		It is highly recommended to use reference control points in any conversion project. This allows you to make sure ProLat is configured properly for the desired conversion. For example, in lat-lon to State Plane, a control point would have a coordinate in lat-lon format and its known corresponding coordinate in State Plane format. A ProLat conversion should match very closely – within 1 cm or mm depending on the type of conversion. If you suspect it does not match, check the parameters carefully. Common problems include units, lat-lon / lon-lat order, datum shift needed, negative value is needed for West and South directions, etc.

License

ProLat for .Net may be used on one single computer within your company without charge. This is typically a development computer. Each additional computer requires a purchased license of ProLat for .Net regardless how how the software is transmitted to that computer. If ProLat for .Net is used on a server PC that delivers services to other PCs, a ProLat for .Net server license is required.

References

1. http://www.ngs.noaa.gov/faq ： National Geodetic Survey – Frequently Asked Questions

2. Snyder, *Map Projections – A Working Manual*, U.S. Geological Survey Professional Paper 1395.

3. Gerald Evenden, Cartographic Projection Procedures for the UNIX Environment – A User's Manual, USGS 1995