AddressBroker

Reference Manual for Windows, UNIX

Version 4.10

Information in this document is subject to change without notice and does not represent a commitment on the part of the vendor or its representatives. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, without the written permission of Precisely, 1700 District Ave Ste 300, Burlington, MA 01803-5231.

© 1994, 2021 Precisely. All rights reserved.

Precisely is a wholly-owned subsidiary of Syncsort Incorporated. See www.precisely.com for information about our valuable trademarks.

The following trademarks are owned by the United States Postal Service®: CASS, CASS Certified, DPV, eLOT, FASTforward, First-Class Mail, Intelligent Mail, LACSLink, NCOALink, PAVE, PLANET Code, Postal Service, POSTNET, Post Office, RDI, SuiteLink, United States Postal Service, Standard Mail, United States Post Office, USPS, ZIP Code, and ZIP+4. This list is not exhaustive of the trademarks belonging to the Postal Service.

USPS Notice: Precisely holds a nonexclusive license to publish and sell ZIP+4 databases on optical and magnetic media. The price of the Precisely product is neither established, controlled, nor approved by the U.S. Postal Service.

Precisely is a non-exclusive licensee of USPS® for NCOALink® processing. Prices for the Precisely products, options and services are not established, controlled or approved by USPS® or United States Government. When utilizing RDI™ data to determine parcel-shipping costs, the business decision on which parcel delivery company to use is not made by the USPS® or United States Government.

Spectrum Geocoding Datasets used within Precisely applications are protected by various trademarks and by one or more of the following copyrights:

Copyright © United States Postal Service. All rights reserved.

© 2021 TomTom. All rights reserved. This material is proprietary and the subject of copyright protection and other intellectual property rights owned by or licensed to TomTom or its suppliers. The use of this material is subject to the terms of a license agreement. Any unauthorized copying or disclosure of this material will lead to criminal and civil liabilities.

© 2021 HERE

Copyright © United States Census Bureau

© Copyright Canada Post Corporation
The delivered database contains data from a compilation in which Canada Post Corporation is the copyright owner.

The Master Location Data (MLD) product is a produced work that referenced the Microsoft US Building Footprints dataset. This dataset is available at https://github.com/Microsoft/USBuildingFootprints and is licensed under the Open Database License (ODbL). The license is available at https://opendatacommons.org/licenses/odbl/.

Contents

Contents

1 – Before You Begin

In this chapter

This chapter discusses the purpose and use of this guide, how its conventions are presented, and how to obtain assistance from Precisely.

Purpose of this guide

This guide provides information on using AddressBroker functionality including the underlying GeoStan library, Demographics Library, and Spatial+ functionality. In addition to reviewing the client, including related API (Java, .NET, C and C++) and server applications, this guide also provides information regarding properties, fields, and pertinent codes (match, location, and status).

If you need more help

If you are unable to resolve a problem, a Precisely Technical Support Representative can help guide you to a solution. To open a Support case, go to https://support.precisely.com/casemanagement/. When you contact Precisely Technical Support, please have the following information ready:

- A description of the task you were performing.
- The resulting reports (specifically, the Execution Log and Parameter Record Listing).

Reporting complete details to Technical Support will help you and the technical support representative resolve the problem quickly .

The Website

You can also find out about Precisely software products and services on our website: https://www.precisely.com.

To obtain additional user guides

To obtain electronic copies of our product manuals, go to :https://support.precisely.com.

2 – Introduction to AddressBroker

In this chapter

About AddressBroker

Your customer database is the heart of your business. You use it to maintain the valuable relationships you have established, to generate monthly billing statements, and to forecast where your business is being generated and where you should focus your marketing and sales efforts. Centrus AddressBroker can improve the way you manage your customer data by processing it for address standardization, geocoding, demographic enhancement, and spatial analysis. With AddressBroker, you can write an application to retrieve demographic information from Precisely' demographic data (.dld) files. You can also use spatial data (polygons, lines, and points) in a proprietary and optimized (.gsb) file format. Separately available loader programs convert other data files into the formats AddressBroker uses for these purposes.

AddressBroker combines and extends the functionality of the Precisely core programming libraries: GeoStan, GeoStan Canada, Demographics Library, Spatial+™, and the Centrus Geographic Determination Library. Use AddressBroker to rapidly develop applications that can run in client/server or Internet environments. AddressBroker provides the following application programming interfaces (APIs) to develop your applications: C, C++, Java, .NET, and ActiveX.

AddressBroker processes address data in programmatic and interactive applications. Use AddressBroker for multiple record processing or to find matches for incomplete addresses.

AddressBroker "brokers" transactions between your client application and the underlying programming libraries (GeoStan, Spatial+, Demographics, GDL, and GeoStan Canada) that can best service the transactions. Based on the information your application requests, AddressBroker divides up the task of providing that information among these components. It then gathers the information that they provide and returns that information to your application.

Brokering also lets you process multiple records with a single processing call. In client/server and Internet environments, brokering enhances performance and reduces network traffic. AddressBroker hides low-level network communication protocols from your application. In client/server and Internet environments, AddressBroker supports direct connections via TCP/IP sockets.

AddressBroker functionality

This section describes AddressBroker's functionality in terms of its underlying products. The specific combination of functionality and data files available to you depends on your licensing agreement with Precisely.

GeoStan functionality

To standardize addresses, Address Broker:

- Compares an input address to a database of addresses (either USPS or Canadian data, if the address is in Canada).
- If GeoStan data is conflated, combines the USPS data with street vendor data to standardize the address.

You can also geocode your address data. Geocoding is the process of assigning geographic designations, such as latitude and longitude, to an address using premium geographic vendors or Precisely Enhanced data files.

After your address database is standardized and geocoded, you can make demographic and spatial enhancements to it through AddressBroker's Demographics Library and Spatial+ components. AddressBroker's address standardization and geocoding functionality form the foundation of address database management.

Note: AddressBroker also supports the USPS Delivery Point Validation (DPV) and the Locatable Address Conversion System process (LACS^{Link}). For information on adding this functionality, contact your Precisely customer representative.

Valid addresses

AddressBroker incorporates sophisticated algorithms from GeoStan to improve match rates for poorly formed input addresses. For the highest match rate, your address data should be as close to USPS standards as possible. To be processed according to Postal Service guidelines, addresses are required to have at least the following items:

- A street address that has at least a house number and a street name
- Either a city and state, or a ZIP Code.

When a match is returned, you can retrieve any elements missing from your input address.

Note: Although Postal Service guidelines require a city and state, or a ZIP Code in the input last line, GeoStan can perform matching with a lastline that only contains the city. For more information, see "City-only lastline matching" on page 32.

Address elements

Street address elements include: House Number, Prefix Direction, Street Name, Street Type, Postfix Direction, Unit Type, and Unit Number. Last line address elements include City, State, ZIP Code, and ZIP + 4. Not all addresses contain all elements. For example, the street address:

123 Elm St.

does not contain any directionals or unit information, yet it may be a valid U.S. Postal Service street address. Street suffix and pre- and post-directional elements may not be critical elements of some addresses. AddressBroker can parse address lines into their component parts. It processes address lines as single elements or as collections of fields.

AddressBroker automatically processes building names, city names, and hyphenated addresses.

Building name matching

AddressBroker standardizes building name addresses to the correct street address. For example:

Empire State Building
New York, NY

is correctly standardized to:

Empire State Building
350 5th Ave.
New York, NY 10118-0110

City name matching

With AddressBroker, a large number of city abbreviations can be recognized and standardized.

Centerline matching

Centerline matching is used with point-level matching to tie a point-level geocode with its parent street segment. This functionality is useful for routing applications.

This provides you with additional data about the parent street segment that is not retrievable using only the point-level match. When retrieving the information, AddressBroker also supplies the bearing and distance from the point data geocode to the centerline match.

Optionally, a centerline offset distance may be specified. The offset specifies, in feet, a distance to move the point from the street centerline toward the parcel centroid. This is useful in routing applications. If the specified distance places the geocode beyond the parcel centroid, the parcel centroid is returned.

Centerline matching requires that you are licensed for point-level matching.

Hyphenated address support

Address ranges are checked for missing or misplaced hyphens. Alphanumeric ranges are checked for proper sequence.

For example, if a house number is incorrectly entered with a hyphen, the number is first concatenated. If no match is resolved, the portion of the number following the hyphen is tested as a unit number.

Address match methodology

AddressBroker searches all records in the locale (city) or Finance Areas for a given city. A Finance Area is a collection of ZIP Codes within a contiguous geographic region. The result of an address search is a set of possible matches between an input address line and the search area.

Each possible match is assigned a score. A confidence score is assigned to each address element. The scoring system takes into account all address anomalies such as dropped or transposed characters, minor misspellings, and “identical” address ranges, where one range is for a street and the other range is for a high rise. The list of possible matches is then sorted by score. The match with the best (lowest) score is returned. A match with a score of 0 exactly matches all scored address elements.

If the best rating is not unique, your application can supply routines to select the best match. You can do this programmatically or interactively.

When matching addresses, select one of the following match modes:

- The “Exact” match mode requires a nearly perfect match. This provides very fast processing. Precisely recommends using this match mode when an address list is known to contain previously standardized addresses.
- The “Close” match mode is optimized to return as many correct addresses as possible from a “dirty” list. Precisely recommends using this match mode for processing lists that have not previously been standardized.
- The “Relax” match mode is the slowest to process, but attempts numerous transpositions to create a match. It may return incorrect matches when it should return no matches at all. Thus, Precisely recommends using this match mode only for interactive use, when a user views each address as it is processed.
- The “Interactive” match mode is only for use with single-line address matching. It is designed for interactive matching such as used, for example, in mobile applications, so it permits more flexible matching patterns. It may, in some cases, return additional possible matches than the “Relax” match mode. For more information on Interactive mode, see “Using Interactive match mode” on page 13.
- The “CASS” match mode processes an address according to USPS CASS rules. The purpose of this mode is to create a list of mailable addresses. This mode generates a large number of match candidates. This mode does not perform intersection, building

name, spatial alias (TIGER, HERE and TomTom street name alias), or Centrus alias matches. It also does not match to candidates from data sources that do not have USPS equivalent records. This mode recognizes and parses two unit numbers on the same address line, for example a building and unit number.

- The “Custom” match mode allows applications to specify individual “must match” field matching rules for address number, addressline, city, Zip Code, state.

Note: The CASS and Custom match modes are not supported in single-line address matching.

See “Pre-defined property values” on page 354 for information about defined constants for each match mode.

Note: Although AddressBroker has a CASS processing mode, AddressBroker is not CASS certified.

Using Interactive match mode

Interactive mode is designed for interactive mobile/web applications. In this use case, it is expected that users may enter single-line addresses that contain misspelled, inaccurate, and/or missing information, so GeoStan processes this input utilizing a looser set of criteria for matching than the other match modes. As a result, the matching output could include multiple match candidates. The list of matches can be presented to the user who would then select the desired match candidate. If an exact match is found, then that single match candidate is returned; a mix of accurate and inaccurate results will not be presented. The following table shows a comparison of the match results when running in interactive vs. close or relaxed modes.

Single-line input address	Interactive mode match candidates	Close/Relaxed mode single match candidate
HIGHLAND VIEW WINCHESTER 01890	5 HIGHLAND VIEW AVE, WINCHESTER, MA 01890 5 HIGHLAND TER, WINCHESTER, MA 01890 5 HIGHLAND AVE, WINCHESTER, MA 01890	5 HIGHLAND VIEW AVE, WINCHESTER, MA 01890
414 PINE WILLIAMSFIELD 61489	414 N PINE ST, WILLIMAMSFIELD, IL 61849 414 PINE ST, WILLIAMSFIELD, IL 61489	414 N PINE ST, WILLIMAMSFIELD, IL 61849
46 HORNBEAM ST CRANSTON RI (conflict with street type)	46 HORNBEAM DR, CRANSTON, RI	46 HORNBINE ST, CRANSTON, RI
611 W 13TH JOPLIN MO 64801 (conflict between directional and ZIP Code)	611 E 13TH ST, JOPLIN, MO 64801 611 W 13TH ST, JOPLIN, MO 64804	611 W 13TH ST, JOPLIN, MO 64804

Capabilities and restrictions:

- Interactive match mode is only available in single-line address processing. If an attempt is made to run a non-single-line address when the match mode is set to `AB_MODE_INTERACTIVE`, the match mode is temporarily changed to `AB_MODE_RELAX` and the address is processed in relaxed mode. When the matching process completes, the match mode is automatically reverted back to `AB_MODE_INTERACTIVE`.
- Interactive match mode allows users to break the cardinal rule: If the user enters 123 **S** Main and there is only 123 **N** Main, a match is made and a match code is returned that reflects the modified directional.
- Interactive match mode handles cases where users transpose pre-directionals with post-directionals without penalty.
- Interactive match mode ignores the `PREFER_ZIP_OVER_CITY` setting. When the city and ZIP Code don't match correctly, the best geocoding result will be returned based on an analysis of all the input address elements.
- When operating in interactive mode, in cases where a point address or interpolated street address result cannot be determined, ZIP-9, ZIP-7 or ZIP-5 centroid(s) may be returned.

Setting up Interactive mode

To set up Interactive mode, set `MATCH_MODE` to:

- For C, C++, .NET, Java APIs: `AB_MODE_INTERACTIVE`
- For ActiveX API: `ABX_MODE_INTERACTIVE`

To process a single-line address:

- Set the `INPUT_MODE` property to `NORMAL`; and
- Place the single-line address into the addressline field and leave the other address fields empty.

Point-Level option

The Point-Level option incorporates data that locates addresses at the center of the actual building footprint or parcel. This provides enhanced geocoding accuracy for internet mapping, flood hazard determination, property and casualty insurance, telecommunications, the utility industries, and others. You must use the Point-Level option with a street network data set from an appropriate vendor; you cannot use the option as a standalone product.

Additional Point-Level datasets may be licensed that allow you to retrieve supplemental information about the parcel. The Centrus APN dataset allows you to retrieve the APN. The Centrus Elevation dataset allows you to retrieve the parcel centroid elevation.

For more information on adding the Point-Level option to your AddressBroker license, contact Precisely Sales.

Using Master Location Data

Master Location Data (MLD) is a comprehensive, multi-sourced dataset that includes every known, addressable location in the United States. Because MLD is sourced from multiple data resources, it is a more complete universe of addresses than any single data source. A unique identifier, PreciselyID, is assigned to each physical addressable location within MLD, which allows users to more easily manage their address data and unlock a wealth of information linked to it.

Having a more complete universe of addresses available for matching results in an increase in high confidence address matches, and a decrease in false-positive matches. A false-positive match results when an incomplete input address is compared against an incomplete dataset, and the wrong match is returned because there is not enough information in either the input, or the matching dataset, to know that the address has been mismatched.

An example of this is an input address of “100 Main St”. In one matching data source there may be only a “100 E Main St”, and in another matching data source there may only be a “100 W Main St”, even though both “100 W Main St” and “100 E Main St” are valid. In both cases, the “100 Main St” input address would match to the record in the matching data source, and there would be a high level of confidence that the match was correct because it was only compared against a single address in each data source. In both cases, it would be a false-positive match since the input of address “100 Main St” could mean either “100 E Main St”, or “100 W Main St”. However, in the case of MLD, since the addresses come from multiple sources, both “100 W Main St” and “100 E Main St” would exist in the matching data. In this case, a multiple match would be returned for the input address “100 Main St”, rather than a false-positive match to either “100 W Main St”, or “100 E Main St”.

The premium matching confidence of MLD is further enhanced by the availability of more high-precision geocodes for the addressable locations included in the MLD dataset. MLD considers location information from multiple data sources to provide the highest precision geocode available for each address. This provides an increase in high-precision geocodes when compared with any single source.

Additional features for Master Location Data

Optional matching features:

- PreciselyID ZIP Centroid Locations
- Point of Interest matching

Optional geocoding features:

- Expanded Centroids
- Extended Attributes

Optional PreciselyID features:

- PreciselyID Fallback
- Reverse PreciselyID Lookup

The PreciselyID unique identifier

The PreciselyID is a unique identifier assigned to each physical addressable location within the Master Location Dataset. The PBKEY field is returned when a match is made to MLD. It is a 12-character (+1 null) field that has ‘P’ as the leading character, and is a persistent identifier for an address.

Use Cases

Some of the benefits provided by the PreciselyID include:

- Access to attribute data that provides additional information about an address such as demographics, proximity to hazards, availability of services and other property information.
- Improved efficiency in managing and maintaining consistent and accurate data for customer address lists.
- The ability to generate an address list of customers targeted for products and services based on specific attributes associated with their address.

The following sections provide more detailed information.

GeoEnrichment of Address Data

The PreciselyID unique identifier serves as a lookup key with Precisely GeoEnrichment datasets to add attribute data for an address location. Depending on the GeoEnrichment dataset(s) you install, the attribute data can include property ownership, real estate, census, consumer expenditure, demographic, geographic, fire and flood protection, and/or telecommunication and wireless systems information and more. Some of these datasets return point location specific data, such as property ownership and real estate, whereas others provide polygonal-based data, for example, fire and flood protection, which can identify flood plains, wildfire or rating territories.

Address Master Data Management using Reverse PreciselyID Lookup

To ensure the latest address information and most accurate locations are being used, businesses may regularly geocode their customer address list. There is a cost in terms of computing power to this intensive process, as well as a small chance of changes to the address match. Some businesses monitor these changes since it's integral to their business. Additionally, many businesses have multiple address databases across different business functions, and have the need for consistent representation of a single address across multiple systems and databases. The Reverse PreciselyID Lookup feature removes the need to re-geocode the address by using the PreciselyID unique identifier rather than the address as input. The address together with latitude/longitude coordinates are returned. The Reverse PreciselyID Lookup process is substantially faster and therefore less costly than using the address to retrieve this information. In addition, since a PreciselyID is persistent, there is no chance of matching to a different address.

Identifying Addresses from GeoEnrichment Data using Reverse PreciselyID Lookup

The GeoEnrichment Fabric products are a variety of text-based data files that contain different attributes for each address in the Master Location Dataset. You can use the attributes in one or more of these GeoEnrichment datasets to identify customers for products or services based on those specific attributes. The lookup key for these products is the PreciselyID unique identifier rather than the address. This allows you to easily link customers across multiple datasets if you need to consider attributes included in more than one GeoEnrichment dataset. For example, using Ground View Family Demographics Fabric, in conjunction with Property Attribute Fabric, you would be able to generate a list of PreciselyIDs for records that represent young families, with 4 or more persons, in large houses, to target for specific products and services. Once records with the desired attributes have been identified, the PreciselyIDs from those records can be used to return the address and location information for those customers using PreciselyID Reverse Lookup.

Optional matching features

PreciselyID ZIP Centroid Locations

The default behavior of GeoStan is to return matches from Master Location Data only for addressable locations that have an address-level geocode. ZIP centroid returns are optionally available when matching to Master Location Data. For addresses that don't have a high-quality location, this provides access to the PreciselyID which can be used to unlock additional information about an address using GeoEnrichment data, as well as to realize operational processing efficiencies. This allows us to ensure maximum address coverage and integrity in geocoding. The inclusion of these addresses enables us to provide a higher match rate, lower false-positive match rate, and access to the PreciselyID for all known addressable locations in the US.

Implementation

Server (.ini file)

If you are using a Server initialization (.ini) file, make sure you define:

- `GEOSTAN_PATHS` - the data paths to the DVDMLD and DVDMLD2 folders, as well as any other datasets you have installed for your application.
- `LICENSE_PATH` - the paths to the licenses for the datasets defined in `GEOSTAN_PATHS`.

- `LICENSE_KEY` - the password for the associated license.

Batch application

To enable PreciselyID ZIP Centroid Locations in your batch application:

- In the `abbatch.ini` configuration file:
  - `ZIP_PBKEYS=TRUE`

Client application

To enable PreciselyID ZIP Centroid Locations in your client application, use the appropriate **`SetProperty`** method based on your language:

- Java:
  - `ab.setProperty("ZIP_PBKEYS", “True”);`
- .NET:
  - `ab.setProperty("ZIP_PBKEYS", "True");`
- ActiveX:
  - `ab.SetPropertyXBool("ZIP_PBKEYS", true)`
- C:
  - `QABSetPropertyStr(broker, "ZIP_PBKEYS", "True");`
- C++:
  - `broker->SetProperty("ZIP_PBKEYS", "True");`

Point of Interest matching

The optional Point Of Interest (POI) Index file (poi.gsi) included with Master Location Data provides expanded support in alias name matching. For more information, see “Using building name, firm and Point of Interest matching” on page 28 and “Using the optional POI Index file” on page 29.

Optional geocoding features

Expanded Centroids

In some cases, more than one point-level geocode is available for an address matched in Master Location Data (MLD) (for more information on the different types of point-level geocodes, see the "APnn" definitions in Address location codes). Expanded Centroids are available automatically with MLD and the presence of an optional dataset `us_cents.gsc`. If an address match is found in MLD, and the optional dataset `us_cents.gsc` is loaded, AddressBroker will search the optional `us_cents.gsc` for additional geocodes for the matched address. When more than one point-level geocode is available from MLD data, only the highest quality geocode available (based on license) is returned with the matched address returns using `ProcessRecords`. The returned location code for an Expanded Centroids match will have an "APnn" value with a data type of "`MASTER LOCATION`".

Extended Attributes

The MLD Extended Attributes dataset used in conjunction with MLD returns the Assessor's Parcel Number (APN) and elevation data for the matched address, as well as additional extended attributes when available. A complete listing of available fields can be found in "GeoStan output fields" on page 400.

Requirements

The following is required to return data from the MLD Extended Attributes data set:

- Master Location Dataset (.gsd and .gsi files).
- Streets data set.
- MLD Extended Attributes data set (`extatt*p.dld` files).
- It is recommended that the vintages of the MLD and MLD Extended Attributes data sets be within 4 months of each other.

Implementation

Server (.ini file)

If you are using a Server initialization (.ini) file, make sure you define:

- `GEOSTAN_PATHS` - the data paths to the DVDMLD and DVDMLD2 folders and the folder where you installed the MLD Extended Attributes dataset, as well as any other datasets you have installed for your application.
- `LICENSE_PATH` - the paths to the licenses for the data sets defined in `GEOSTAN_PATHS`.
- `LICENSE_KEY` - the password for the associated license.

Batch application

- APN and elevation returns, as well as additional extended attributes when available, from MLD Extended Attributes is supported using any `INPUT_MODE` value, except for `REVERSE_APN`. Reverse APN matching is only available with Centrus Points and Centrus APN data; it is not supported using MLD and MLD Extended Attributes data.
- To return APN and elevation data, include `ApnID and ParcelCentroidElevation` (elevation of the parcel centroid is returned in feet; the return of the elevation in meters is not supported) in the Outfield Field Layout definition.

Client application

The client application should set the following properties:

- `GEOSTAN_PATHS` - the data paths to the DVDMLD and DVDMLD2 folders and the folder where you installed the MLD Extended Attributes dataset, as well as any other datasets you have installed for your application.
- `LICENSE_PATH` - the paths to the licenses for the datasets defined in `GEOSTAN_PATHS`.
- `LICENSE_KEY` - the password for the associated license.
- `Output_Field_List` - define your desired output fields using this property. To return APN and elevation data, include `ApnID and ParcelCentroidElevation` (elevation of the parcel centroid is returned in feet; the return of the elevation in meters is not supported).

Optional PreciselyID features

PreciselyID Fallback

When using PreciselyID Fallback, if an address match is not made to Master Location Data, but a match is made to a different dataset, the PreciselyID unique identifier of the nearest MLD point located within the search distance is returned. To distinguish when a fallback PreciselyID is returned, the `PBKEY` return value contains a leading character of "X" rather than "P", for example: X00001XSF1IF. Note, all of the other fields returned for the address match, including the geocode and all associated data returns from AddressBroker, reflect the match results for the input address. The fallback PreciselyID can then be used for the lookup to the GeoEnrichment dataset(s), and the attribute data for the fallback location is returned for the match.

The relevance and accuracy of the returned attribute data using a PreciselyID Fallback location is highly dependent on the type of GeoEnrichment data, as well as the PreciselyID Fallback search distance. PreciselyID Fallback is intended for use with GeoEnrichment datasets that have polygonal-based data, rather than point-specific data. For example, the PreciselyID Fallback option may be suitable for determining the FEMA flood zone for a given location using the Flood Risk Pro GeoEnrichment dataset since it contains data that represents a polygonal region rather than a single coordinate. However, it is important to note that the accuracy of the returned data would very much depend on the size and nature of the individual polygonal features described in the GeoEnrichment data, combined with the search distance used to locate the nearest Master Location Data point. The search distance is configurable with an allowable search radius of 0-5280 feet and a default value of 150 feet.

Requirement

PreciselyID Fallback requires that you have licensed and installed Master Location Data.

Implementation

Note: PreciselyID Fallback can only be used in forward and reverse geocoding.

PreciselyID Fallback is disabled by default; the following steps describe how to enable this feature:

1. Enable the Approximate PBKey find property.
2. Optional. Set the search distance. Valid values = 0-5280 feet. Default = 150 feet.

Batch application

Your AddressBroker batch application should include:

- In the `abbatch.ini` configuration file:
  - Define an output field for `PBKEY` in the Output Field Layout section.
  - APPROX_PBKEY=TRUE.
- Optionally, in the `server.ini` file:
  - Set the reverse geocoding search distance, “`RevGeoSearchDistance`”.

Client application

In all api’s, include the `PBKEY` output field in the `OUTPUT_FIELD_LIST`.

To enable PreciselyID Fallback from the client, use the appropriate **`SetProperty`** method based on your language. The following methods use the property string names:

- Java:
  - `ab.setProperty(“ApproxPbKey”, “True”);`
  - optional: `ab.setProperty(“RevGeoSearchDistance“, value);`
- .NET:
  - `ab.setProperty(“ApproxPbKey”, "True");`
  - optional: `ab.setProperty(“RevGeoSearchDistance“, value);`
- ActiveX:
  - `ab.SetPropertyXBool("ApproxPbKey", true)`
  - optional: `ab.SetPropertyXLong("RevGeoSearchDistance", value)`
- C:
  - `QABSetPropertyStr(broker, "ApproxPbKey", "True");`
  - optional: `QABSetPropertyStr(broker, “RevGeoSearchDistance“, value);`
- C++:
  - `broker->SetProperty("ApproxPbKey", "True");`
  - optional: `broker->SetProperty(“RevGeoSearchDistance“, value);`

Reverse PreciselyID Lookup

Reverse PreciselyID Lookup is an optional licensed matching feature. This features uses a PreciselyID unique identifier as input and returns all standard returns that are provided as part of address matching.

Licensing

Reverse PreciselyID Lookup requires a special license. There are two levels of licensing for Reverse PreciselyID Lookup:

- Standard - This license allows Reverse PreciselyID Lookup of all of the standard MLD addresses.
- Enhanced - This license allows Reverse PreciselyID Lookup of a portion of MLD addresses that require an additional royalty due to address sourcing constraints.

Requirements

The Reverse PreciselyID Lookup feature includes the following requirements:

- You have licensed and installed the Master Location Dataset (MLD).
- You have licensed and installed the DVDMLDR dataset.
- The MLD and DVDMLDR datasets must be the same vintage.

Implementation

The following sections describe how to implement the Reverse PreciselyID Lookup feature in your AddressBroker application.

Server (.ini file)

If you are using a Server initialization (.ini) file, make sure you define:

- `GEOSTAN_PATHS` - the data paths to the DVDMLD, DVDMLD2 and DVDMLDR folders, as well as any other datasets you have installed for your application.
- `LICENSE_PATH` - the paths to the licenses for the datasets defined in `GEOSTAN_PATHS`.
- `LICENSE_KEY` - the password for the associated license.

Batch application

Your AddressBroker batch application should include:

- In your Input record - include `PBKEY` input fields (see “GeoStan input fields” on page 393).
- In your Configuration file - include `INPUT_MODE=REVERSE_PBKEY` (see “Configuration parameters” on page 94).

Client application

The client application should set the following properties:

- `GEOSTAN_PATHS` - the data paths to DVDMLD, DVDMLD2 and DVDMLDR, as well as any other datasets you have installed for your application.
- `LICENSE_PATH` - the paths to the licenses for the datasets defined in `GEOSTAN_PATHS`.
- `LICENSE_KEY` - the password for the associated license.

- `Input_Mode` - set this control property to one of the supported Reverse PreciselyID Lookup input modes, either:
  - `AB_INPUT_NORMAL` or `ABX_INPUT_NORMAL` (ActiveX),
  - `AB_INPUT_PARSED` or `ABX_INPUT_PARSED` (ActiveX), or
  - `AB_INPUT_PARSED_LASTLINE` or `ABX_INPUT_PARSED_LASTLINE` (ActiveX).
- `Input_Field_List` - set this property to `PBKEY`.
- `Output_Field_List` - define your desired output fields using this property, which can include the address and the standard set of return fields for a match.

Reverse PreciselyID Lookup Search Results

When using Reverse PreciselyID Lookup, the search results can return zero to many MLD point address variations that match the input PreciselyID. There will be no matches returned if the given PreciselyID is not found. While many PreciselyIDs map to a single point-level address, some PreciselyIDs map to multiple point address variations. Getting multiple point address variations from one PreciselyID can occur in two circumstances:

1. **Alias matches**. Some streets are known by their common name and one to many aliases. In this case, MLD may contain all variations of street names. An example of multiple alias match returns for an input PreciselyID (P00008BCG8WM) is shown below:
   - `AP02. Normal match (non-alias). 1206 W 600 S, FOUNTAINTOWN, IN 46130-9409`
   - `AP02. Alias match. 1206 W 1200 N, FOUNTAINTOWN, IN 46130-9409`
   - `AP02. Alias match. 1206 W COUNTY ROAD 1200 N, FOUNTAINTOWN, IN 46130-9409`
   - `AP02. Alias match. 1206 W COUNTY ROAD 600 S, FOUNTAINTOWN, IN 46130-9409`
2. **Multi-unit buildings with/without units.** In some cases, there are multi-unit addresses without individual unit address records. In this case, you may see multiple address records returned for the same input PreciselyID, some without unit designations and others with ranged unit designations. In the case of multi-unit addresses that have individual suite/unit number address designations, each will have their own distinct PreciselyID. The following example shows address results for a PreciselyID that maps to a building with and without units, which share the same PreciselyID/location (P00003PZZOIE):
   - `AP02. Normal match (non-alias). 4750 WALNUT ST, BOULDER, CO 80301-2532`
   - `AP02. Normal match (non-alias). 4750 WALNUT ST STE 100-103, BOULDER, CO 80301-2532`
   - `AP02. Normal match (non-alias). 4750 WALNUT ST STE 205-205, BOULDER, CO 80301-2532`
   - `AP02. Normal match (non-alias). 4750 WALNUT ST, BOULDER, CO 80301-2538`

Reverse PreciselyID Lookup Match Codes

The following table lists the match codes returned with Reverse PreciselyID Lookup.

License	Input PreciselyID	Point Results	getFields() Match Code
Enhanced	Found	One Enhanced	V000
Enhanced	Found	Multiple Standard and/or Enhanced	V001
Enhanced	Not Found	None	E040
Standard	Found	One Standard	V000
Standard	Found	Multiple Standard	V001
Standard	Found	One Standard, some Enhanced	V002
Standard	Found	Multiple Standard, some Enhanced	V003
Standard	Found	All Enhanced	E041
Standard	Not Found	None	E040
No license	N/A	N/A	E000

DPV option

Delivery Point Validation (DPV) is a United States Postal Service technology that validates the accuracy of address information down to the physical delivery point. DPV is only available through a CASS-certified vendor, such as Precisely, and is an optional feature.

Previous address-matching software could only validate that an address fell within the low-to-high address range for the named street. By incorporating the DPV technology, you can resolve multiple matches and determine if the actual address exists. Using DPV reduces undeliverable-as-addressed (UAA) mail that results from inaccurate addresses, reducing postage costs and other business costs associated with inaccurate address information.

DPV also provides unique address attributes to help produce more accurate mailing lists. For example, DPV provides information on if a location is vacant, and can identify commercial mail receiving agencies (CMRAs) and private mail boxes.

See Appendix C: USPS Link products for more information on DPV. For more information on adding DPV processing to your AddressBroker license, contact Precisely Sales.

LACSLink option

The Locatable Address Conversion System (LACS) converts rural addresses to city-style addressees. LACSLink is a USPS technology that provides mailers with an automated process to correct addresses in areas that have undergone LACS processing. Address conversions occur when the LACS process modifies, changes, or replaces an address. This usually occurs due to one of the following: the conversion of rural routes and box numbers to city-style addresses, the renaming or renumbering of existing city-style addresses to avoid duplication, or the establishment of new delivery addresses.

See Appendix C: USPS Link products for more information on LACSLink. For more information on adding LACSLink processing to your AddressBroker license, contact Precisely Sales.

Understanding SuiteLink

The purpose of SuiteLink™ is to improve business addressing by adding known secondary (suite) numbers to allow delivery sequencing where it would otherwise not be possible. SuiteLink uses the input business name, street number location, and 9 digit ZIP+4 to return a unit type (i.e. "STE") and unit number for that business.

As an example, when entering the following address with SuiteLink enabled in CASS mode.

UT Animal Research
910 Madison Ave
Memphis TN 38103

AddressBroker returns the following:

UT Animal Research
910 Madison Ave STE 823
Memphis TN 38103

Or

UT Animal Research
910 Madison Ave #823
Memphis TN 38103

If you have licensed the SuiteLink processing option, you must install the SuiteLink data and set the SuiteLink initialization properties for AddressBroker to process your address through SuiteLink. For more information on SuiteLink, see Appendix C: USPS Link products.

Reverse geocoding option

Reverse geocoding is an optional processing feature that provides you with a way to enter a point consisting of a longitude and latitude (geocode) and receive information about that point.

To use the reverse geocoding option, you need additional data files, called GSX files. There is an option to install these files when you install the standard AddressBroker data. By default, AddressBroker installs these files in the GSX directory. You must specify this directory when initializing AddressBroker.

Note: Reverse geocoding is currently not available for Guam.

Using reverse geocoding to points matching

The reverse geocoding to points matching feature provides the option to match to the nearest point address within the search radius, rather than to the closest feature (e.g. street segment or intersection as well as point addresses).

Note: This feature requires that at least one points data set and one streets data set are loaded; otherwise, the match will be made to the closest feature.

By setting the value of AB_CLOSEST_POINT, you can specify whether AddressBroker searches for the following:

- TRUE = Matches to the closest point address within the search radius.
- FALSE = *default*. Matches to the closest feature including street segments and intersections in addition to address points.

Reverse APN option

Reverse APN lookup is an optional processing feature that provides you with a way to enter FIPS and APN codes to receive information on the corresponding parcel. To use the reverse APN lookup functionality, you need the Centrus APN dataset.

To make a match, AddressBroker must exactly match against the input APN ID, state FIPS code, and county FIPS code.

Match location (geocodes)

You can enhance address information with a match location (geocode). Geocodes are expressed in latitude and longitude. A geocode is determined by the street segments available in a street database. Street coordinates are calculated to millionths of a degree, enabling you to clearly display a match location point on top of a base map derived from the same street network.

A house number's range is calculated and mapped onto the appropriate street segment. AddressBroker correctly handles street segment shape points when assigning the match location.[1] An offset distance may be used. Offsets are calculated perpendicular to the street segment range associated with an input address. This approach yields the best visual representation for mapping packages and gives the most accurate location possible from the geographic data.

Census Block data can also be generated for the match location.

Street intersections can be geocoded but do not return USPS information, such as ZIP Codes, as they are not valid addresses for postal delivery.

Street locator geocoding

Street locator geocoding is an optional feature. When this feature is enabled, if a street name is encountered while geocoding, and there is no matching address range, AddressBroker will attempt to locate the street within the input ZIP Code or city if there is no input ZIP Code. If AddressBroker is able to locate the street, it will return a geocode along the matched street segment rather than the geocode for the entered ZIP Code or ZIP + 4.

If a street number is entered, AddressBroker will return the coordinates of the end point of the closest numeric street segment within the input ZIP Code. When there is no input ZIP Code, the closest numeric street segment of all the ZIP Codes within the input city will be returned.

If no street number is entered, the centroid of a matching street segment within the input ZIP Code will be returned. The centroid of a street segment for all the ZIP Codes within the input city will be returned when there is no input ZIP Code.

When using street locator geocoding, it is likely that a match code of either E029 (no matching range, single street segment found), or E030 (no matching range, multiple street segment) returns. For example, if you enter Main St and there are both an E Main St and a W Main St within the input ZIP Code then an E030 returns and the location code returned is reflective of the input ZIP Code. The location code returned begins with a 'C' when matched to a single street segment, indicated by E029. For more information regarding the match and location codes associated with this feature, see "Status Codes" on page 442 and "Geographic centroid location codes" on page 440.

Note: This option is not available in CASS mode.

1. Street segments are described by shape points. A straight segment has a point at each end. A street segment with one or more curves is described by multiple shape points defining the curve(s).

Using building name, firm and Point of Interest matching

AddressBroker can enhance standard address matching by matching to building and business names.

By default, AddressBroker is able to match building names with unit numbers in the address line, the Chrysler building as an example:

Firm:
Address: 5001 Chrysler Bldg
Last Line: New York New York 10174

The returned information is the address of the Chrysler building. AddressBroker returns a standardized address in place of the building name:

Firm:
Address: 405 Lexington Ave RM 5001
Last Line: New York, NY 10174-5002

Entering the White House, as an example, into the address line, the address for the White House returns in the address field:

Firm:
Address: White House
Last Line: Washington DC 20500

AddressBroker returns the following address:

Firm:
Address: 1600 Pennsylvania Ave NW
Last Line: Washington DC 20500-0004

The ability to search by building name entered in the address line is controlled by modifying "BUILDING_SEARCH".

Entering a firm name in the Firm name field returns the address for the input firm in the address field:

Firm: White House
Address:
Last Line: Washington DC 20500

AddressBroker returns the following address:

Firm: White House
Address: 1600 Pennsylvania Ave NW
Last Line: Washington DC 20500-0004

AddressBroker attempts to match a firm name entered in the input firm name field to a firm name in the data files. The firm name field must contain only the firm name. This is an optional search that will occur after all other address searching has failed to find a match.

By setting the value of "ALTERNATE_LOOKUP", you can specify whether AddressBroker searches for the following:

- 1 = Matches to the address line, if a match is not made, then GeoStan matches to the Firm name line.
- 2 = Matches to the Firm name line, if a match is not made, then GeoStan matches to address line.
- 3 = (Default) Matches to the address line.

To enable firm name matching in the AddressBroker server, modify the abserver.ini file to include the `ALTERNATE_LOOKUP` property. For example:

`ALTERNATE_LOOKUP` = 1

To activate Alternate Lookup from the client, use the appropriate **`SetProperty`** method based on your language:

- Java - The Java client API uses a “setter” to enable Firm Name Matching:
  - `ab.setProperty(“ALTERNATE LOOKUP”, “1”);`
- .NET - The .NET client API uses a “setter” to enable Firm Name Matching:
  - `ab.setProperty(“ALTERNATE LOOKUP” , “1”);`
- ActiveX - Does not set properties using enumerators (enums), so no changes are required.
- C – Use the appropriate SetProperty method based on your language:
  - `QABSetPropertyStr(broker, "ALTERNATE LOOKUP", "1");`
- C++ - `broker->SetProperty("ALTERNATE LOOKUP", "1");`

Note: Neither building nor firm name searches are available when processing in CASS mode.

Using the optional POI Index file

The optional Point Of Interest (POI) Index file (poi.gsi) included with the Master Location Data and HERE Point Addresses datasets provides expanded support in alias name matching.

To enable matching to the POI Index file:

1. Define the data paths, license paths and license keys to the MLD or HERE Points datasets, as well as any other datasets you have installed for your application.
2. Set the “BUILDING_SEARCH” property to true. The POI Index file will automatically be searched when the `BUILD_SEARCH` option is enabled and a firm, building or POI name is specified in the address line.

3. Process the match. If an alias match is made to the POI Index file, the `IsAlias` output field returns "A11".

Using correct last line

"CORRECT_LAST_LINE", when set to True, corrects elements of the output last line, providing a good ZIP Code or close match on the soundex even if the address would not match or was non-existent.

The feature works when "CENTROID_PREFERENCE" is True and the address does not match a candidate or when `AB_CENTROID_NO_ADDRESS`, on page 355, is True and only last line information is input.

For example when `CENTROID_PREFERENCE = True`

Address: 0 MAIN
LastLine: BOLDER CA 80301

Returns:

MATCH_CODE=E622
LASTLINE=BOULDER, CO 80301
CITY=BOULDER
STATE=CO
ZIP=80301

For example, `AB_CENTROID_NO_ADDRESS  = True`

Address:
LastLine: BOLDER CA 80301

Returns:

MATCH_CODE=Z6
LASTLINE=BOULDER, CO 80301
CITY=BOULDER
STATE=CO
ZIP=80301

The following elements are corrected:

- City correction - The city correction is based on input ZIP unless a match to city and state exists in which case both search areas are retained. The state input must be correct or spelled out correctly when no ZIP is input, location code, and coordinates based on input ZIP.

– Input city is incorrect:

HAUDENVILLE MA 01039

Returns LASTLINE=HAYDENVILLE, MA 01039
LAT= 42396500 LON= -72689100

- State correction - State is abbreviated when spelled out correctly or corrected when a zip is present. There are some variations of state input which are recognized, ILL, ILLI, CAL, but not MASS. GeoStan does not consideration the abbreviation of the variation a change so ILL to IL is not identified as a change in the match code. In addition the output of the ZIP for a single ZIP city is not considered a change.

– Input city exists:

Bronx NT, 10451
Returns LASTLINE= BRONX, NY 10451

Bronx NT
Returns LASTLINE= BRONX NT
No ZIP Code for correction

– Input city does not exist - preferred city for ZIP Code returned:

60515
Returns LASTLINE=DOWNERS GROVE, IL 60515
MATCH_CODE=E622

ILLINOIS 60515 (or ILL 60515 or IL 60515 or ILLI 60515)
Returns LASTLINE=DOWNERS GROVE, IL 60515
MATCH_CODE=E222

- ZIP correction - ZIP is corrected only when a valid city/state is identified and has only one ZIP.

– Exists on input:

HAUDENVILLE MA 01039
Returns LASTLINE=HAYDENVILLE, MA 01039

– Incorrect on input - ZIP Code correction is not performed, both search areas are retained:

HAUDENVILLE MA 01030
Returns LASTLINE=HAYDENVILLE, MA 01030
City and ZIP do not correspond

– Does not exist on input:

DOWNRS GROVE, IL
Returns LASTLINE=DOWNERS GROVE, IL
City with multiple ZIP Codes

LILSE IL
Returns LASTLINE=LISLE, IL 60532
City with a single ZIP Code

DOWNERS GROVE LL
Returns LASTLINE=DOWNERS GROVE LL,
No ZIP Code for correction

DOWNRS GROVE, LL
Returns LASTLINE=DOWNRS GROVE, LL
No ZIP Code for correction

LILSE ILLINOIS
Returns LASTLINE= LISLE, IL 60532
Correct spelled out state

LISLE ILLINOS
Returns LASTLINE= LISLE ILLINOS
Incorrect spelled out state, no ZIP Code for correction

To enable firm name matching in the AddressBroker server, modify the abserver.ini file to include the `CORRECT_LAST_LINE` property. For example:

```
CORRECT_LAST_LINE = True
```

Options are True or False. The default value for `CORRECT_LAST_LINE` is `False`.

To activate Correct Last Line from the client, use the appropriate **`SetProperty`** method based on your language:

- Java - The Java client API uses a “setter” to enable Correct Last Line:
  - `ab.setProperty(“CORRECT LAST LINE”, “True”);`
- .NET - The .NET client API uses a “setter” to enable Correct Last Line:
  - `ab.setProperty(“CORRECT LAST LINE” , “True”);`
- ActiveX - Does not set properties using enumerators (enums), so no changes are required.
- C – Use the appropriate SetProperty method based on your language:
  - `QABSetPropertyStr(broker, CORRECT LAST LINE” , “True”);`
- C++ - `broker->SetProperty(“CORRECT LAST LINE” , “True”);`

Note: For information on returned match codes see Correct last line match codes.

City-only lastline matching

City-only lastline matching permits address matching with only a city in the input lastline. The input address should be provided using Normal or Parsed lastline rather than Multiline input fields. With city-only lastline input, AddressBroker will search all of the states in which the input city exists. Therefore, there is the possibility of an increase in multi-matches (E023 and E030) when matching with city-only input instead of city+state lastline input.

Restrictions:

- City-only lastline input matching is not supported in CASS mode.
- City-only lastline is not supported when matching to User Dictionaries.
- When matching using city-only lastline, the `PREFER_ZIP_OVER_CITY` setting is ignored.
- It is strongly recommended to not use city-only lastline matching in Relaxed match mode to avoid the return of false-positive matches.

Using predictive lastline

Predictive lastline allows you to match an address when only an input street address and latitude/longitude coordinates are provided, rather than the traditional street address with lastline input. For example, an input of 4750 Walnut with latitude/longitude coordinates located in Boulder, will return full address information.

Additional feature information:

- Predictive lastline uses the search radius designated for reverse geocoding.
- If the input lat/lon falls near the borders of multiple cities, GeoStan processes all cities and returns the results of the best match. If the results are determined as equal, then a multi-match is returned.
- This feature does not require a license for reverse geocoding.
- This feature will work with any type of data set.

Preferring a ZIP Code over a city

The “PREFER_ZIP_OVER_CITY” property allows a user to prefer candidates that match to input ZIP over candidates that match to input city. GeoStan creates multiple search areas when input city and ZIP do not correspond and this feature helps establish how the candidates should be scored.

Note: GeoStan ignores the ZIP over city preference if processing in Interactive and CASS modes.

When there is more than one candidate in the input ZIP, some attempt is made to alleviate a multiple match, or, where all the candidates get the same last line score. If a candidate also matches the city and/or preferred city, that candidate gets a better score. Matching to just preferred city is a lesser score than matching both.

Input Address: 24 GLEN HAVEN RD
Input Last Line: NEW HAVEN CT 06513

Found:
24 GLEN HAVEN RD
NEW HAVEN, CT 06513-1105

Possible candidates:

			score	pref.last line city	
2 98	GLEN HAVEN RD	06513-1105 S	0.8100000	NEW HAVEN	* best match
24 98	GLEN HAVEN RD	06513-1248 S	2.2500000	EAST HAVEN	
16 66	GLEN RD	06511-2825 S	46.3925000	NEW HAVEN	
2 86	GLEN PKWY	06517-1415 S	52.1525000	HAMDEN	
2 28	GLEN RD	06516-6509 S	52.1525000	WEST HAVEN	
2 98	GLENHAM RD	06518-2517 S	75.0100000	HAMDEN	
2 72	GLEN VIEW TER	06515-1519 S	97.0900000	NEW HAVEN	

When there is more than one candidate, candidates matching the input ZIP score better.

Input Address: 301 BRYANT ST
Input Last Line: SAN FRANCISCO CA 94301

Found:
301 BRYANT ST
PALO ALTO, CA 94301-1408

Possible candidates:

			score	pref.last line city
301 301	BRYANT ST	94301-1408 S	3.2400000	PALO ALTO * ZIP preferred match
301 305	BRYANT CT	94301-1401 S	28.2400000	PALO ALTO
300 306	BRYANT CT	94301-00ND T	35.6600000	PALO ALTO
301 301	BRYANT ST	94107-4167 H	39.6900000	SAN FRANCISCO * default match
301 319	BRYANT ST	94107-1406 S	39.6900000	SAN FRANCISCO

When there is more than one candidate, candidates that match the ZIP search area score better. The ZIP search area is the finance area for the input ZIP.

This example with match mode set to Relax or Cass. With match mode set to Exact or Close the match is made to EAST AURORA 14052 as there is no candidate in 14166 the input ZIP.

Input Address: 100 MAIN ST
Input Last Line: EAST AURORA NY 14166

Found:
100 MAIN ST
DUNKIRK, NY 14048-1844

Possible candidates:

			score	pref.last line city
100 198	MAIN ST	14048-1844 S	3.2400000	DUNKIRK * same finance as input ZIP 14166
100 168	MAIN ST	14052-1633 S	39.6900000	EAST AURORA

This example with the match mode set to Exact or Close.

Input Address: 4200 arapahoe
Input Last Line: denver co 80301

Found:
4200 ARAPAHOE AVE
BOULDER, CO 80303-1164

Possible candidates:

				score	pref.last line city
4200	4210	ARAPAHOE AVE	80303-1164 S	38.7400000	BOULDER *same city as input zip 80301
4200	4210	ARAPAHOE RD	80303-1164 S	40.7000000	BOULDER (A06)
4200	4298	E ARAPAHOE PL	80122-00ND T	62.0900000	LITTLETON
4200	4498	E ARAPAHOE RD	80122-00ND T	62.0900000	LITTLETON
4181	4499	E ARAPAHOE RD	80122-00ND T	68.3400000	LITTLETON

To enable Prefer ZIP Over City in the AddressBroker server, modify the abserver.ini file to include the `PREFER_ZIP_OVER_CITY` property. For example:

```
PREFER_ZIP_OVER_CITY = True
```

Options are True or False. The default value for `PREFER_ZIP_OVER_CITY` is `False`.

To activate Prefer ZIP Over City from the client, use the appropriate **`SetProperty`** method based on your language:

- Java - The Java client API uses a “setter” to enable Prefer ZIP Over City:
  - `ab.setProperty(“PREFER ZIP OVER CITY”, “True”);`
- .NET - The .NET client API uses a “setter” to enable Prefer ZIP Over City:
  - `ab.setProperty(“PREFER ZIP OVER CITY” , “True”);`
- ActiveX - Does not set properties using enumerators (enums), so no changes are required.
- C – Use the appropriate **`SetProperty`** method based on your language:
  - `QABSetPropertyStr(broker, PREFER ZIP OVER CITY” , “True”);`
- C++ - `broker->SetProperty(“PREFER ZIP OVER CITY” , “True”);`

Matching address ranges

Some business locations are identified by address ranges and can be geocoded to the interpolated mid-point of the range. Address ranges are different from hyphenated (dashed) addresses that occur in metropolitan areas. For example, a hyphenated address could be 243-20 Main St, which represents a single residence and is geocoded as a single address. If a hyphenated address similar to this example returns as an exact match, then there is no attempt to address range match.

Address range matching is disabled by default and is an optional mode. To enable address range matching, use the settable property “RANGED_ADDRESS”. Address range matching is only available in Close and Extend modes. It is not available in Exact or CASS™ modes since an address range is not a deliverable USPS® address. The following fields are not returned by address range geocoding:

- ZIP+4® (in multiple segment cases)
- Delivery Point
- Check Digit
- Carrier Route
- Record Type
- Multi-Unit
- Default flag

Address Range matching capabilities and guidelines

Address Range matching works within the following guidelines:

- There must be two numbers separated by a hyphen.
- The first number must be lower than the second number.
- Both numbers must be of the same parity (odd or even) unless the address range itself has mixed odd and even addresses.
- Numbers can be on the same street segment or can be on two different segments. The segments do not have to be contiguous.
- If both numbers are on the same street segment, the geocoded point is interpolated to the approximate mid-point of the range.
- If the numbers are on two different segments, the geocoded point is based on the last valid house number of the first segment. The ZIP Code and FIPS Code are based on the first segment.
- In all cases, odd/even parity is evaluated to place the point on the correct side of the street.

The Address Range match in the example below is to a single street segment with the geocode being placed on the mid-point of the range:

Input: 4750-4760 Walnut St, Boulder, CO
Output: 4750-4760 Walnut St, Boulder, CO

A close match to a single address number is preferred over a ranged address match. AddressBroker attempts a close match on the recombined address number before making a ranged match, as seen in the following example:

Input: 47-50 Walnut St, Boulder, CO
Output: 4750 Walnut St, Boulder, CO

In the example below, the second number is not larger than the first so AddressBroker treats this as a unit number rather than a ranged address:

Input: 4750-200 Walnut St, Boulder, CO
Output: 4750 Walnut St STE 200, Boulder, CO

See Match codes and Location Codes for more information on the return codes.

To enable Address Range Geocoding in the AddressBroker server, modify the abserver.ini file to include the following:

```
RANGED_ADDRESS = TRUE
```

Options are TRUE or FALSE. The default value for `RANGED_ADDRESS` is `FALSE`.

To activate Address Range Geocoding from the client, use the appropriate **`SetProperty`** method based on your language:

- Java - The Java client API uses a “setter” to enable Address Range Geocoding:
  - `ab.setProperty(“RANGED_ADDRESS”, “TRUE”);`
- .NET - The .NET client API uses a “setter” to enable Address Range Geocoding:
  - `ab.setProperty(“RANGED_ADDRESS“, “TRUE”);`
- ActiveX - Does not set properties using enumerators (enums), so no changes are required.
- C – Use the appropriate **`SetProperty`** method based on your language:
  - `QABSetPropertyStr(broker, "RANGED_ADDRESS", "TRUE");`
- C++ - `broker->SetProperty("RANGED_ADDRESS", "TRUE");`

Please note that the default state is set to false: no Address Range Geocoding.

Understanding missing and wrong first letter

The missing and wrong first letter feature enables AddressBroker to look for the correct first letter of a street address if the first letter is missing or incorrect. AddressBroker searches through the alphabet looking for possible, correct first letters to complete the street address.

The feature’s default is disabled, except in `EXACT` mode or the equivalent `CUSTOM` mode settings. To enable this feature, modify "FIRST_LETTER_EXPANDED".

Below are some examples of wrong, missing first letter, and duplicate first letter input addresses and the corresponding AddressBroker output:

The example include an incorrect first letter:

Input: 4750 nalnut boulder co 80301
Output: 4750 Walnut St Boulder CO 80301-2532

This example excludes a first letter:

Input: 4750 alnut boulder co 80301
Output: 4750 Walnut St Boulder CO 80301-2532

This example includes an extra first letter:

Input: 4750 wwalnut boulder co 80301
Output: 4750 Walnut St Boulder CO 80301-2532

Permitting relaxed address number matching

When AddressBroker matches an input address, its default behavior is to match to the address number. This default behavior corresponds to "MUST_MATCH_ADDR_NUM" set to True.

If "MUST_MATCH_ADDR_NUM" is set to False, then AddressBroker no longer must match the address number, therefore permitting relaxed address number matching. By permitting relaxed address number matching, an inexact match can be found. If the input address number is missing, no matches are returned unless `STREET_CENTROID` is also enabled.

When using Relaxed Address Number Matching, if there is no match to the input house number, or if the input house number is blank, the result returned from AddressBroker indicates a non-match. This is because AddressBroker is not able to make a match based on the input data. However, AddressBroker will return a geocode to the nearest available house number on the input street.

Note: Relaxed Address Number Matching is only available with Custom match mode.

To enable Relaxed Address Number Matching in the AddressBroker server, modify the abserver.ini file to include the following:

```
MUST_MATCH_ADDR_NUM = FALSE
```

Options are TRUE or FALSE. The default value for `MUST_MATCH_ADDRR_NUM` is `TRUE`.

To activate Relaxed Address Number Matching from the client, use the appropriate **`SetProperty`** method based on your language:

- Java - The Java client API uses a “setter” to enable Relaxed Address Number Matching:
  - `ab.setProperty(“MUST_MATCH_ADDRR_NUM“, “FALSE”);`
- .NET - The .NET client API uses a “setter” to enable Relaxed Address Number Matching:
  - `ab.setProperty(“MUST_MATCH_ADDRR_NUM“, “FALSE”);`
- ActiveX - Does not set properties using enumerators (enums), so no changes are required.
- C – Use the appropriate **`SetProperty`** method based on your language:
  - `QABSetPropertyStr(broker, "MUST_MATCH_ADDRR_NUM", " FALSE");`
- C++ - `broker->SetProperty("MUST_MATCH_ADDR_NUM", " FALSE");`

Please note that the default state is set to true: no Relaxed Address Number Matching processing – must match on address number.

Understanding address point interpolation

Address point interpolation uses a patented process that improves upon regular street segment interpolation by inserting point data into the interpolation process. When an address point User Dictionary (UD) or a point GSD is present, more precise address geometry is used for interpolation than what is available by the use of street segments alone. Please note that this feature does not work with point addresses in the Auxiliary File.

In order to implement this feature, you must have:

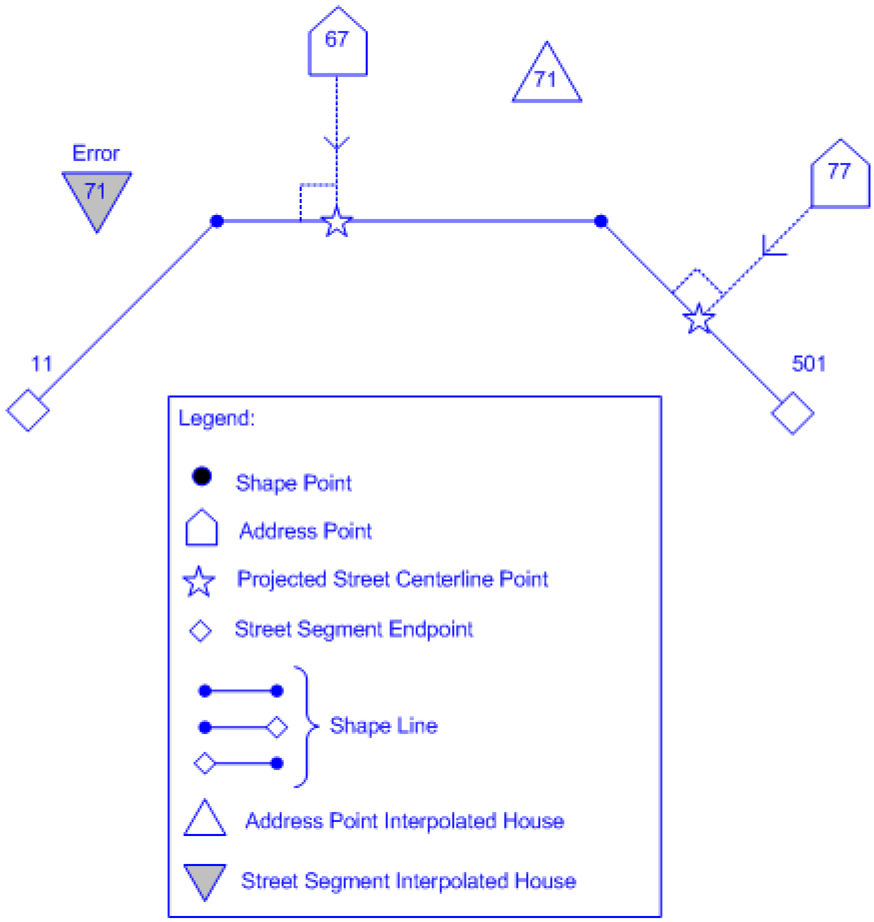
- point data (UD or licensed GSD) loaded

- "ADDR_POINT_INTERP" property set to "`True`"

AddressBroker will first attempt to find a match using the loaded point data, in priority order. If an exact point match is found in the point data, then searching ceases and the point match is returned. If an exact point match was not found, AddressBroker attempts to find high and low boundary address points to use for address point interpolation. To illustrate the use of this feature, view the example below.

In this example, the input house number is 71. The point GSD contains address points for house numbers 67 and 77. The street segment ranges from house number 11 to 501 and contain shape lines describing the physical layout of the street.

AddressBroker attempts to map the points for addresses 67 and 77 onto the closest shape line. After finding a point on the centerline of the street, GeoStan then performs the interpolation for the input house number 71 with the new street centerline points of 67 and 77. Without this feature, GeoStan performs an interpolation with the street segment end points of 11 and 501. This creates a far less accurate result (labeled in the diagram) than using the centerline points of the closest surrounding high and low address points.



See Match codes and Location Codes for information on the return codes related to this feature.

To enable address point interpolation in the AddressBroker server, modify the abserver.ini file to include the following:

```
ADDR_POINT_INTERP = TRUE
```

Options are TRUE or FALSE. The default value for `ADDR_POINT_INTERP` is `FALSE`.

To activate address point interpolation from the client, use the appropriate **`setProperty`** method based on your language:

- Java - The Java client API uses a “setter” to enable address point interpolation:
  - `ab.setProperty(“ADDR_POINT_INTERP“, “TRUE”);`
- .NET - The .NET client API uses a “setter” to enable address point interpolation:
  - `ab.setProperty(“ADDR_POINT_INTERP“, “TRUE”);`
- ActiveX - Does not set properties using enumerators (enums), so no changes are required.
- C – Use the appropriate **`setProperty`** method based on your language:

- `QABSetPropertyStr(broker, "ADDR_POINT_INTERP", "TRUE");`
- C++ - `broker->SetProperty("ADDR_POINT_INTERP", "TRUE");`

Note that the default state is set to false: no address point interpolation.

Note: Also see Using User Dictionaries with address point interpolation.

Matching to a geographic centroid

Geographic centroids can be returned by inputting valid combinations of city, county, and state or as a fallback. You can geocode to the city, county, or state centroid. Although geographic centroid geocoding is less precise than street or postal geocoding, it may be suitable for certain applications.

Geographic centroid matching uses a new option called "FALLBACK_GEOGRAPHIC". When this option is enabled, AddressBroker will return a geographic centroid match when it cannot match a record to the level of precision originally requested, such as street level or ZIP Code level. For geographic geocoding, AddressBroker returns the most precise geographic centroid that it can, based on the user input.

See Match codes and Location Codes for more information on the return codes related to this feature.

A number of prominent U.S cities can be matched even if no other information is provided. (Ex.: Chicago (input city) but no input state, matches to Chicago, IL. The ability to match on an input city is determined by the geographic rank of the city (1-7). Cities with a geographic rank of 1 or 2 are able to be matched without an input state.

Output fields:

"Geographic Rank" - Relative city ranking (1-7). Options are True or False. The default value for `FALLBACK_GEOGRAPHIC` is `False`.

To activate fallback to geographic centroid matching from the client, use the appropriate **`SetProperty`** method based on your language:

- Java - The Java client API uses a "setter" to enable Geographic Centroid Matching:
  - `ab.setProperty("FALLBACK_GEOGRAPHIC", "True");`
- .NET - The .NET client API uses a "setter" to enable geographic centroid matching:
  - `ab.setProperty("FALLBACK_GEOGRAPHIC ","True");`
- ActiveX - Does not set properties using enumerators (enums), so no changes are required.
- C – Use the appropriate **`SetProperty`** method based on your language:
  - `QABSetPropertyStr(broker, " FALLBACK_GEOGRAPHIC ", "True");`
- C++ - `broker->SetProperty("FALLBACK_GEOGRAPHIC ", "True");`

Understanding Extended Match Codes

The Extended Match Code property enables the return of additional information about any changes in the house number, unit number and unit type fields. In addition, it can indicate whether there was address information that was ignored. The Extended Match Code is only returned for address-level matches (match codes that begin with A, G, H, J, Q, R, S, T or U), in which case a 3rd hex digit is appended to the match code (see GeoStan return codes).

Note: A typical match code contains up to 4 characters: a beginning alpha character followed by 2 or 3 hex digits. The third hex digit is only populated for intersection matches or as part of the Extended Match Code.

For information about the 3rd hex digit values for:

– Intersection matches, see "Definitions for 1st-3rd hex digit match code values" on page 424.
– Extended Match Codes, see "Definitions for Extended Match Code (3rd hex digit) values" on page 425.

"Address information ignored" is specified when any of the following conditions apply:

- The output address has a mail stop (`Mailstop`).
- The output address has a second address line (`AddressLine2`).
- The input address is a dual address (two complete addresses in the input address). For example, "4750 Walnut St. P.O Box 50".
- The input last line has extra information that is not a city, state or ZIP Code, and is ignored. For example, "Boulder, CO 80301 USA", where "USA" is ignored when matching.

Input Addressline	Output Addressline	Extended Code	Description
4750 WALNUT ST STE 200	4750 WALNUT ST STE 200	0	Matched on all address information on line, including Unit Number and Unit Type if included.
4750 WALNUT ST C/O JOE SMITH	4750 WALNUT ST	1	Matched on Unit Number and Unit Type if included. Extra information on address line ignored. Extra information not considered for matching moved to AddressLine2 or Mail Stop field.
4750 WALNUT ST UNIT 200	4750 WALNUT ST STE 200	2	Matched on Unit Number. Unit Type changed.

Input Addressline	Output Addressline	Extended Code	Description
4750 WALNUT ST UNIT 200 C/O JOE SMITH	4750 WALNUT ST STE 200	3	Matched on Unit Number. Unit Type changed. Extra information on address line ignored. Extra information not considered for matching moved to AddressLine2 or Mail Stop field.
4750 WALNUT ST STE 2-00	4750 WALNUT ST STE 200	4	Unit number changed or ignored.
4750 WALNUT ST STE 2-00 C/O JOE SMITH	4750 WALNUT ST STE 200	5	Unit Number changed or ignored. Extra information on address line ignored. Extra information not considered for matching moved to AddressLine2 or Mail Stop field.
4750 WALNUT ST STE 400	4750 WALNUT ST STE 400	6	Unit Number changed or ignored. Unit Type changed or ignored. In this example, Suite 400 is not valid for the input address, but the address match is not prevented because of an invalid unit number.
4750 WALNUT ST UNIT 2-00 C/O JOE SMITH	4750 WALNUT ST STE 200	7	Unit Number changed or ignored. Unit Type changed or ignored. Extra information on address line ignored. Extra information not considered for matching moved to AddressLine2 or Mail Stop field.
47-50 WALNUT ST STE 200	4750 WALNUT ST STE 200	8	Matched on Unit Number and Unit Type if included. House Number changed or ignored.
47-50 WALNUT ST STE 200 C/O JOE SMITH	4750 WALNUT ST STE 200	9	Matched on Unit Number and Unit Type if included. House Number changed or ignored. Extra information not considered for matching moved to AddressLine2 or Mail Stop field.
47-50 WALNUT ST UNIT 200	4750 WALNUT ST STE 200	A	Matched on Unit Number. Unit Type changed. House Number changed or ignored.
47-50 WALNUT ST UNIT 200 C/O JOE SMITH	4750 WALNUT ST STE 200	B	Matched on Unit Number. Unit Type changed. House Number changed or ignored. Extra information on address line ignored. Extra information not considered for matching moved to AddressLine2 or Mail Stop field.
47-50 WALNUT ST STE 20-0	4750 WALNUT ST STE 200	C	House Number changed or ignored. Unit Number changed or ignored.
47-50 WALNUT ST STE 20-0 C/O JOE SMITH	4750 WALNUT ST STE 200	D	House Number changed or ignored. Unit number changed or ignored. Extra information on address line ignored. Extra information not considered for matching moved to AddressLine2 or Mail Stop field.

Input Addressline	Output Addressline	Extended Code	Description
47-50 WALNUT ST UNIT 20-0	4750 WALNUT ST STE 200	E	House Number changed or ignored. Unit Number changed or ignored. Unit Type changed or ignored.
47-50 WALNUT ST UNIT 2-00 C/O JOE SMITH	4750 WALNUT ST STE 200	F	House Number changed or ignored. Unit Number changed or ignored. Unit Type changed or ignored. Extra information on address line ignored. Extra information not considered for matching moved to AddressLine2 or Mail Stop field.

To enable the return of Extended Match Codes in the AddressBroker server, modify the abserver.ini file to include the `MATCH_CODE_EXTENDED` property. For example:

```
MATCH_CODE_EXTENDED = True
```

Options are True or False. The default value for `MATCH_CODE_EXTENDED` is `False`.

To activate Extended Match Codes from the client, use the appropriate **`SetProperty`** method based on your language:

- Java - The Java client API uses a “setter” to enable Extended match codes:
  - `ab.setProperty(“MATCH_CODE_EXTENDED”, “True”);`
- .NET - The .NET client API uses a “setter” to enable Extended match codes:
  - `ab.setProperty(“MATCH_CODE_EXTENDED”, “True”);`
- ActiveX - Does not set properties using enumerators (enums), so no changes are required.
- C – Use the appropriate **`SetProperty`** method based on your language:
  - `QABSetPropertyStr(broker, “MATCH_CODE_EXTENDED”, “True”);`
- C++ - `broker->SetProperty(“MATCH_CODE_EXTENDED”, “True”);`

Note: For information on returned match codes see Definitions for Extended Match Code (3rd hex digit) values.

Understanding User Dictionaries

A User Dictionary is a table of streets and address ranges that you use as a source for geocoding. If you have newer or more precise data than what is available in GSD files, creating a dictionary with this data can help you obtain more accurate geocoding results. For example, if you have address point data you can create a User Dictionary that enables you to take advantage of the AddressBroker address point interpolation capabilities.

A User Dictionary can be used by itself to geocode records, or can be used in combination with the supplied GSD.

For more information see User Dictionary.

Note: The USPS does not consider matches to data that they did not create and these are not considered valid addresses for postal delivery. Therefore, AddressBroker does not match to User Dictionaries when processing in CASS mode

GeoStan Canada

A standardized Canadian address contains the street address, municipality, province, and complete postal code, corrected to Canada Post Corporation (CPC) standards. A geocoded address contains the address as found in the CPC data files, as well as the latitude and longitude. A detailed match code is also returned for each process.

GeoStan Canada uses data from Precisely's Enhanced Address Geocoding database and the CPC Postal Code data files.

Note: More predictable results are achieved when the postal code is entered. Canadian addresses are standardized according to SERP regulations, which are very different than USPS CASS regulations.

Demographics Library functionality

AddressBroker lets you query a Demographic Library data (.dld) file for demographic information. Attach this data to an address record to enhance targeted marketing efforts. When processing records, AddressBroker automatically uses GeoStan data, such as the Census Block and ZIP9 fields, as input to the Demographics Library.

Currently, the available data source is the U.S. Census. All the Demographics Library data is available with AddressBroker.

Spatial+ functionality

By incorporating Spatial+ functionality, AddressBroker lets you compare geocoded addresses to spatial (.gsb) files. Point-in-polygon analysis determines within which geographic areas (spatial polygons) a known point falls. For example, you could use point-in-polygon analysis to assign a sales territory for new customer records, or for calculating which store trade areas contain the most overlap by measuring the amount of customer overlap rather than the area of overlap. Assigning closest site or radial analysis determines a geocode's distance from a site point, its orientation to the site point, and the site point's name. For example, you could use closest site or radial analysis to determine which store is closest to each customer, to determine the five closest doctors to a potential client, or listing up to 20 store locations within 15 miles of a proposed site.

Note: AddressBroker automatically uses Latitude and Longitude field values from GeoStan with the Spatial+ Library.

Spatial objects describe various types of either topographic items (such as mountains, streets, or buildings) or geographic features (conceptual areas such as municipal boundaries, auto rating territories, or statistical analysis areas). These features can be described mathematically as points, lines, or polygons and are saved in a spatial (.gsb) file.

The Spatial+ functionality within AddressBroker requires .gsb files, which is a proprietary format. Contact your Precisely sales representative for information about creating .gsb files. For additional information about the Spatial+ Library, see the *Spatial+ Reference Manual*.

Spatial attributes

You can also include your own data by using “attributes”. These Spatial attributes might include population counts, revenue figures, demographic characteristics, or other information specific to a region or specific location. You can view the attribute information after you choose your input file, specify the input and outfield fields, and process the data.

Geographic Determination Library functionality

The Centrus Geographic Determination Library (GDL) is designed to be used in conjunction with GeoStan. By accessing the actual run-time values created by GeoStan, GDL can generate a dynamic geo-variance buffer around the geocode and then perform several spatial comparison operations to generate a numeric confidence value.

Geo-variance buffer generation

GeoStan performs geocoding based on address data. A geocode, that is, the particular latitude/longitude coordinates associated with an address, can have one of four basic levels of quality associated with it. This quality level is determined by the address information provided and the data available in a data look-up table.

GDL uses this quality level to create a polygon or geo-variance buffer around the geocoded point. This polygon describes the maximum probable geographic variance that point may have. For example, an address level geocode may have a variance of +/-165 feet East/West and +/-50 feet North/South. The geo-variance polygon outlines the boundary of this 33,000 square foot area. After this buffer is calculated, it can be compared to other spatial objects for accurate determinations.

Comparison operations

GDL is able to access spatial .gsb files and the objects they contain to perform linear and percentage overlap comparison operations.

Comparison Operation	Description
Linear Distance	GDL can determine distance relationship such as: - How far is this house from a fire station? - How far from the edge of a potential mudslide area does a building stand? GDL can return a distance value in feet which describes either how close or far away a given point or line is from a geo-variance buffer.
Percentage Overlap	A typical problem might be whether or not an address falls inside a specific area. For example, is this house in a flood zone? Once GDL has created a geo-variance buffer, it is able to calculate if the buffer overlaps with another polygon and, if it does, how much it overlaps. The percentage value returned describes the probability that a point falls in the comparison area.

Geo-demographic data

AddressBroker is more than a programming library. AddressBroker's geo-demographic data is also integral to the AddressBroker product. The geo-demographic data available to you depends on your license agreement with Precisely. Geo-demographic data is required to process your address records. This data is available via the Precisely eStore. You provide your own data converted to the .gsb format for use with Spatial+ and GDL. GeoStan Canada data is installed with the software.

The data installation includes a file named `datasets.txt` in the *Datasets* and the *Datasets\UNIX directories*. Refer to this file for detailed descriptions of the data sets.

Note: In some instances, you provide your own data for use within AddressBroker, for example, specific spatial data for spatial analysis.

In client/server applications, better performance is achieved when the geo-demographic data is stored on the server. Data can be accessed remotely, but be aware of possible issues concerning permissions. See "Accessing remote data on UNIX platforms" on page 87 if you are running a UNIX operating system.

Types of data

AddressBroker uses several types of geo-demographic data. The data you need depends on the type of address processing you want AddressBroker to do and your license.

Geo-demographic data types include:

- Address standardization data—Used to standardize addresses to USPS and CPC specifications.
- Geocoding data—Used to enhance your address data with geographic information (latitude and longitude).
- Demographics data—Used to enhance your address data with valuable demographic information, for example Census2k.dld. Demographics data must be in .dld format.
- Spatial data—Used to perform spatial analysis using polygon, line, and point files such as `States.gsb` and `Counties.gsb`. These files are included as sample data; provide your own spatial data for your specific data analysis and use your own spatial data to perform spatial enhancement. Spatial data must be in .gsb format.
- Geographic Determination data—Used to enhance your data by working in conjunction with GeoStan to assign a confidence rating to the geocode. Geographic determination data must be in .gsb format.
- Point-Level Option—Used to enhance your data by locating addresses at the center of the building footprint or parcel. This provides enhanced geocoding accuracy for Internet mapping, flood hazard determination, property and casualty insurance, telecommunications, and the utility industries.
- DPV® data — Used to enhance your data by using the USPS Delivery Point Validation (DPV) technology that validates the accuracy of address information down to the physical delivery point.
- RDI™ data — Used to enhance your data by using the USPS Residential Delivery Indicator to verify if an address is a residence or a business.
- LACS^{Link} data — Used to enhance your data by correcting address lists for areas that have undergone Locatable Address Conversion System (LACS) processing. Address list conversion occurs when the LACS process modifies, changes, or replaces an address. This usually occurs due to one of the following: the conversion of rural routes and box numbers to city-style addresses, the renaming or renumbering of existing city-style addresses to avoid duplication, or the establishment of new delivery addresses.

Updating data

AddressBroker provides you with a way to swap the GSB files without having to re-initialize the application; this is referred to as a hot data swap.

GSB and GSA file dependencies

AddressBroker processes hot swapable GSB and GSA files in the following manner:

- During server initialization, if a GSA file is found in the HotSwap or Working directory with an associated GSB file, AddressBroker associates the GSA file with the GSB files. Once a GSB is considered to have attributes, it must have a valid attribute file from this point forward.
- If a GSA file appears in a directory without an associated GSB file, AddressBroker does not take any action. AddressBroker only recognizes GSA files when the associated GSB file is in the same directory.
- If a GSB file has previously had an associated GSA file, you must provide a new GSA file when you update the GSB file in the HotSwap directory. If you update an attributed GSB file and do not provide the associated GSA file, AddressBroker displays an informational message and will not update the GSB file until the associated GSA file is present.
- If you introduce a GSA file for a GSB file that did not previously have an associated GSA file, AddressBroker recognizes the file and henceforth associates a GSA file with the GSB file.

Configuring your system for hot data swap

To use the hot data swap option, you need to include the following properties in your initialization file (`abserver.ini`):

- `HOTSWAP_DIRECTORY`
  The path and name of the directory where the server administrator places the GSB files that AddressBroker loads and the corresponding attribute files (.GSA files). This must refer to a single directory.
- Do NOT put static GSB files (files that are not hot data swapable) in the HotSwap directory.
- `WORKING_DIRECTORY`
  The path and name of the directory where the server holds GSB files it is currently using for processing. Users should not place files into or remove files from this directory. AddressBroker appends the version number to the file name when it moves the file to the working directory.
- `DISCARD_DIRECTORY`
  The path and name of the directory where the server places old versions of the GSB files. Users should monitor and clean this directory as part of regular maintenance activities.
- `ERROR_DIRECTORY`
  The path and name of the directory where the server places GSB files that have failed verification. Users should monitor and clean this directory as part of regular maintenance activities. AddressBroker appends a time stamp and the suffix “Error” to the file name when it moves the file to the error directory.

- `POLLING_TIME`
  The time interval, in seconds, between successive polls of the hot swap directory. The range is between 1 and 86400 seconds.

The discard and error directories may occupy the same location. All other directories must be distinct.

Note: Precisely strongly recommends that you configure the preceding directions on the same file system to avoid longer latency times when you add a new file to the hot swap directory.

To indicate that AddressBroker can hot swap a particular file, add the `HOTSWAP` keyword to the definition for the GSB file in the initialization file.

Logging

AddressBroker logs the following events for the hot swap files:

Event	Debug	Server
File moves or relocates	X	X
New version of a file becomes available for use by the handles.	X	X
The last transaction processed with a file before the server places the file in the discard directory.	X	X
File fails validation.	X	X
Server finds more than one version of a file in the working directory at startup.	X	X
Server recognizes a new version of a file in the hot swap directory.	X	
Validation success.	X	

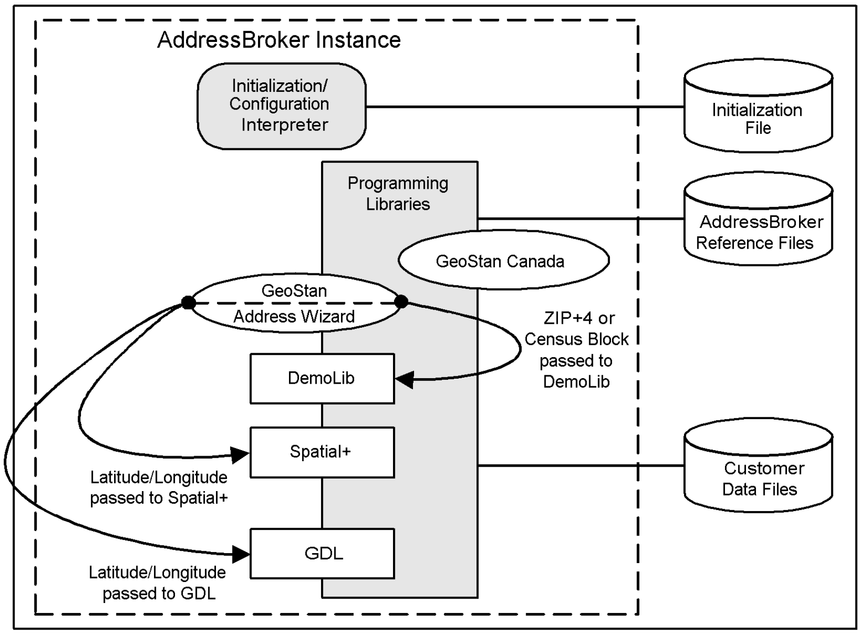
Example code

The following is an example fragment for the initialization file. It indicates that AddressBroker can hot swap the files for `AUTO` and `HOME`, but cannot hot swap the files for `FLOOD`.

```
; Directory configuration for hot swapping
HOT_SWAP_DIRECTORY = “C:\Datasets\Hotswap”
WORKING_DIRECTORY = “C:\Centrus\Working”
DISCARD_DIRECTORY = “C:\Datasets\old”
ERROR_DIRECTORY = “C:\Datasets\errors”
; Check for new data files every 10 seconds
POLLING_TIME = 10

; Spatial Paths
SPATIAL_PATHS = [AUTO]HOTSWAP\auto.gsb | \ [HOME]HOTSWAP\home.gsb
SPATIAL_PATHS = [FLOOD]C:\Data\fema.gsb
```

AddressBroker components

The AddressBroker product is made up of three main components—the AddressBroker initialization/configuration file interpreter, programming libraries, and geo-demographic data files.



The file interpreter reads an .ini file in text format. See Using Initialization Files for information about .ini files. AddressBroker's programming libraries access the appropriate reference files and customer data files needed to complete a processing request. Information is passed between programming libraries as required.

Client/Server model

You can run many client applications using a single AddressBroker server. Typically, the AddressBroker Server is executed with several instance handles. The Handle Manager distributes incoming client requests among these handles to expedite processing. The server includes a queuing mechanism to handle simultaneous client requests.

An individual client request may consist of a single record or multiple records. AddressBroker bundles all of the required information on the client side, then sends it as a single request to the server. The server instance processes the entire request, then returns a single response to the requesting client.

Communication transactions are made at only four points in an AddressBroker client/server application:

- The AddressBroker client object is created.
- AddressBroker properties are validated.

- Address records are processed or an address lookup is done.
- Field attributes are requested.

Application programming model

AddressBroker uses a common application programming model across all its interfaces. See Chapter 5 Client Applications for additional information about client applications.

Basically, all AddressBroker applications include calls to:

1. Create and initialize the AddressBroker object or handle.
2. Set AddressBroker properties to control the processing behavior of AddressBroker. All of AddressBroker's interfaces support the use of a `SetProperty` function to achieve this task. In C, C++, and ActiveX you can also set properties using an .ini file. In ActiveX, you can also use ActiveX properties to accomplish this step.
3. Ensure that a complete set of legal and compatible AddressBroker properties are available to the application using the `ValidateProperties` function.
4. Load one or more input address records into AddressBroker for processing using `SetField` and `SetRecord`.
5. Process the records using either `ProcessRecords` or `LookupRecord`.
6. Retrieve the output using `GetRecord` and `GetField`.
7. Delete the AddressBroker object.

Memory management

Because each client request contains all information necessary for processing, the AddressBroker server does not maintain state information about each connected client.

Within your application, you need only allocate memory to create an AddressBroker instance and free it when the instance is destroyed. AddressBroker allocates and clears all data input and output buffers automatically.

In addition, AddressBroker functions let you query the type, size, and description of AddressBroker property and field values before actually retrieving them. You can retrieve the size of a field's value to efficiently allocate memory in your code.

Threads and multi-threading

AddressBroker client objects are thread-safe when each client instance resides in its own thread. The AddressBroker server is multi-threaded and configured for multiple client applications.

Note that the multi-threaded functionality of AddressBroker requires the multi-threaded support library. Clients developed for AddressBroker and installed on machines other than the development machine require installation of the development environment's libraries that support multi-threaded applications. For example, clients developed using the Microsoft Visual C/C++ environment and installed on machines without the development environment require that `msvcrt.dll` be installed with the client.

Programming interfaces

AddressBroker provides the following programming interfaces:

- Java – Client/server, Internet applications
- .NET – Client/server, Internet applications
- C – Client/server, Internet applications
- C++ – Client/server, Internet applications
- ActiveX component – Client/server, Internet applications

The APIs are available as import libraries and DLLs for 32- and 64-bit Windows developers and as JAR files for Java developers. On UNIX platforms, the APIs are available as either static or dynamic libraries.

In all of the programming languages supported, AddressBroker is an easy-to-use property and field keyword-driven interface. For convenience and readability, AddressBroker keywords are case insensitive, as well as insensitive to spaces and underscores. For example, "`FIRMNAME`" is equivalent to "`Firm Name`" and "`firm_name`".

3 – System Requirements

In this chapter

This chapter describes the system requirements for AddressBroker.

Platform support

You can install a thread-safe version of AddressBroker on the platforms listed in the table below.

Windows OS	UNIX OS
Windows 7 Windows 8 Windows Server 2008 Windows Server 2012	IBM AIX*
	HP-UX
	Sun Solaris
	SuSE Linux*
	Redhat Linux* Redhat Enterprise*

*GeoStan Canada not supported on these platforms.

Note: To see a list of the specific OS versions that Precisely supports, see the *GeoStan Suite Supported Platforms* document available at http://support.precisely.com.

Windows DLL files and UNIX libraries

AddressBroker is distributed as Windows DLL files and as UNIX C libraries. The table below lists the filenames for the UNIX libraries and Windows DLL files.

Platform	Filename	Example
UNIX	lib**<progname>**MT.**<suffix>**	libabMT.a
Windows	**<progname>**MT.dll	AB.dll

Operating system support for AddressBroker APIs

The table below lists the AddressBroker APIs that are supported on each OS.

OS	C/C++	JAVA	.Net
Windows OS	✓	✓	✓
UNIX OS	✓	✓	

4 – Using Initialization Files

In this chapter

An initialization/configuration file is an ASCII text file that sets AddressBroker properties. A language interpreter executes initialization (.ini) files from within AddressBroker. The language interpreter executes whenever you create an instance of the AddressBroker object or start an AddressBroker server.

Initialization files fulfill several important roles in AddressBroker:

- An .ini file sets server properties.
- An .ini file is used optionally to initialize client objects.
- End users can modify property settings using .ini files, without changing compiled code.

The information in this chapter applies to both client and server .ini files.

Guidelines for creating initialization files

The following list includes some general features and guidelines for using .ini files:

- Must be ASCII text files.
- Blank lines are ignored.
- Comments are permitted, except on the GEOSTAN_PATHS line. The first non-space character on comment and no-op instruction lines is a semi-colon (;).
- The instruction syntax for setting properties is:

  `PROPERTY_NAME` = **Value**

  where `PROPERTY_NAME` is a string name of an AddressBroker property and **Value** is a legal value specific to the property being set. See “Sample .ini file” on page 60.
- Property names are insensitive to case (upper/lower), extra spaces, and underscores.
- Property names are never quoted. The general rule is that any term occurring on the left side of an assignment statement may not be quoted.
- Keywords and values generally follow the syntax of their programmatic counterparts.
- Set Boolean values as follows:

  TRUE: `True, true, TRUE, T, t, Yes, yes, YES, Y, y,` `1` (numeric one)

  FALSE: `False, false, FALSE, F, f, No, no, NO, N, n,` `0` (numeric zero)
- When setting values for AddressBroker’s long integer properties, use the numeric representation *not* the preprocessor macro code. For example:

  `MATCH_MODE` = **1**

  not

  `MATCH_MODE` = **MODE_CLOSE**.

- All **values** may be quoted. The general rule is that any term occurring on the right side of an assignment statement may be quoted. Values that include a list delimiter (such as a space) *require* quotes. Values that do not include a list delimiter do not require quotes.
- Single or double quotes may be used.
- Unmatched quotes result in an error.
- If assigning more than one value to a property name, separate the values using either the tab (\t) or pip (|) delimiters.
- Properties that take lists are only required to quote the individual entries in the list, not the entire list:

  PROPERTY_NAME = "**Value**" | "**Value**" | "**Value**"
- Multiline instructions may be constructed by using a backslash (\ ) before end of the line. When a backslash precedes the end-of-line character, the following line is parsed as a continuation of the preceding line.
- Multiline instructions do not support quoting. (Quoted strings may not span multiple lines.)

Sample .ini file

The following code sample provides an example server .ini file. This example illustrates most of the features and requirements of the language, as described in the following sections. The example is specific to an AddressBroker server; however, client .ini files are similar.

Note: The `*_PATHS` properties are set only in the server .ini file.

```
; AddressBroker server .ini file
; This is a comment.

; Extra spaces and blank lines are permitted.

; All paths in this ini file reflect the default WINDOWS
; installation directory structures.
; No need to quote value here because value contains no
; characters that function as list delimiters.
STATUS LOG  = EVENTLOG

STATUS LEVEL = SERVER

; Set the required path properties
; Required to use quotes for these values because they
; contain characters that function as list delimiters
; (spaces, tabs, pipes).
; Single or double quotes OK

GEOSTAN PATHS =
[GEOSTAN] "C:\Program Files\Centrus\cd2tiger" | \
[GDT] "C:\Program Files\Centrus\cd2gdt"

; Multiline instructions.
; Note the backslash ( \ ) characters.
```

```
GEOSTAN Z9 PATHS = \
[GEOSTAN_Z9]"C:\Program Files\Centrus\cd2tiger\US.z9" | \
[GDT_Z9]"C:\Program Files\Centrus\cd2gdt\us.z9"

; Note that individual list entries are quoted,
; not the entire list.
DEMOGRAPHICS PATHS  =
[CENSUS2K] "C:\Program Files\Centrus\CENSUS2K.dld"
SPATIAL PATHS =
[States] "C:\Program Files\Centrus\states.gsb" | \
[Counties] "C:\Program Files\Centrus\counties.gsb"

INIT_LIST = GEOSTAN | GEOSTAN_Z9 | COUNTIES
INPUT_FIELD_LIST = FirmName | AddressLine | \
AddressLine2 | LastLine
OUTPUT_FIELD_LIST = FirmName | City | State |ZIP

; Set the two required license properties
LICENSE PATH = "C:\Program Files\Centrus\AB.lic"
LICENSE KEY  = "11111111"

; Set some additional properties on the server

; Match_Mode property set numerically to AB_MODE_CLOSE.
; No need to quote value here because value contains no
; characters that function as list delimiters.
Match Mode = 1
; Offset distance in feet
OFFSET_DISTANCE = 50

; Set Boolean values
KEEP_MULTIMATCH = True
KEEP_COUNTS = False
```

Initializing AddressBroker servers using .ini files

The AddressBroker server requires an initialization file based on the AddressBroker Interface Language. The server .ini file contains the full path location of the geo-demographic data AddressBroker requires and property settings that control the execution of the AddressBroker server. <Hypertext>"Typical AddressBroker property settings in a server .ini file" shows a fragment of a server .ini file. It shows the error handling properties and the properties server applications require. "Sample .ini file" on page 60 includes an example of a complete server .ini file.

Typical AddressBroker property settings in a server .ini file

```
STATUS_LOG = EVENTLOG
STATUS_LEVEL = SERVER
LICENSE_PATH = "C:\Program Files\Centrus\AB.lic"
LICENSE_KEY = 11111111
GEOSTAN_PATHS = \
[GEOSTAN] "C:\Program Files\Centrus\cd2tiger" | \
[GDT] "C:\Program Files\Centrus\cd2gdt"
GEOSTAN_Z9_PATHS = \
```

```
[GEOSTAN_Z9]"C:\Program Files\Centrus\cd2tiger\US.z9" |\
[GDT_Z9]"C:\Program Files\Centrus\cd2gdt\us.z9"
SPATIAL_PATHS = [COUNTIES] \
"C:\Program Files\Centrus\COUNTIES.gsb"
INIT_LIST = GEOSTAN | GEOSTAN_Z9 | COUNTIES
INPUT_FIELD_LIST = FirmName | AddressLine | \
AddressLine2 | LastLine
OUTPUT_FIELD_LIST = FirmName | City | State |ZIP
```

Logical names

Logical names provide a means for an application to find files or directories without knowing the actual directory or file name. AddressBroker uses logical names to abstract the details of AddressBroker's reference data file names and locations. Logical names may refer to either a data *file* or a data *directory* holding a set of related data files. You associate a unique logical name with a unique data source using AddressBroker's path properties. For information about assigning logical names using AddressBroker's path properties, see "Optional AddressBroker properties" on page 64.

Note: Logical names must be set for the environment where the files are located.

Set logical names only for server environments, not for a client. In environments using multiple AddressBroker servers, ensure that the logical names, and the data they point to, are the same across all servers. This ensures the results returned to a client are consistent, regardless of the server handling the request.

Logical names are limited to 32 bytes in length including the null terminator, and must be inclosed in brackets ([ ]). They are insensitive to spaces, underscores, and case. For example, "`[GEOSTAN_Z9]`" is equivalent to "`[geostan_z9]`" and "`[geostan z9]`".

AddressBroker properties

Properties are AddressBroker system-level variables that control how AddressBroker programs execute, or report on the status of program execution. AddressBroker property names are insensitive to case, spaces, and underscores. For example, "`INIT_LIST`" is equivalent to "`Init List`" and "`initlist`". Throughout this manual, AddressBroker properties are shown in monospaced capital letters (`INIT_LIST`).

This section includes a general discussion about properties, as well as in-depth descriptions of individual AddressBroker properties. For additional information not covered in the following sections, and a full listing of AddressBroker properties including their default values, valid input ranges, data type, and descriptions, see Chapter 13 Properties.

Note: AddressBroker and ActiveX make use of the term "property." Unless otherwise noted, all references in this manual to the term "property" refer to an AddressBroker property. ActiveX properties are discussed in Chapter 12 ActiveX Interface.

Guidelines for setting AddressBroker properties

The following list contains general features and guidelines for setting AddressBroker properties:

- **Defaults**—Most AddressBroker properties have a default setting. AddressBroker applications run with the default values, but you want to set them to better suit the requirements of your application.
- **Requirements**—There are a few properties that you are required to set before running AddressBroker. These are properties that hold information specific to your AddressBroker application such as your license file and password, and the location of your reference data. See "Required properties" on page 64 for additional information.
- **Delimiters**—Some properties are assigned list values. Valid list delimiters are tab (\t) and vertical bar (|). AddressBroker also supports spaces as list delimiters. Thus, you must set off in quotes (single or double) list items that include spaces (for example: `INIT_LIST = A|B|"C D"|'E F'`).
- **Client applications**—In client applications, properties can be set programmatically or in an .ini file.[1] *Most* client property settings override server properties.
- **AddressBroker server**—Server properties are always set in an .ini file. Initialization files consist of property names and values.
- **Names vs. IDs**—When you set properties in an .ini file, you must use character string property names instead of property IDs.
- **Predefined values**—Some AddressBroker properties have a set of predefined values you must choose from.
- **Boolean values**—Specify Boolean values as follows:

  TRUE: `True, true, TRUE, T, t, Yes, yes, YES, Y, y, 1` (numeric one)

  FALSE: `False, false, FALSE, F, f, No, no, NO, N, n, 0` (numeric zero)

AddressBroker properties in server applications

For server applications, you are (generally) required to set a minimum of seven AddressBroker properties. These properties hold your AddressBroker licensing information, name/location information about AddressBroker geo-demographic data, and default initialization information. Precisely also suggests you set the error reporting properties. After you set these properties, you can run your application using the default values for other AddressBroker properties, or configure them to better suit your processing requirements.

In server applications, all properties must be set in an .ini file.

1. The Java API requires you to set client properties programmatically. In ActiveX, you can also use ActiveX properties to set many AddressBroker properties.

Required properties

Set the following AddressBroker properties in the server .ini file. The exact number of properties required depends on the type of processing your application performs.

Type	Description
License Properties	Licensing properties hold information about your AddressBroker licensing agreement. These properties are always required: `LICENSE_PATH`—Fully specified license file name. `LICENSE_KEY`—License file key number.
Path Properties	AddressBroker's path properties hold name and location information about the geo-demographic data AddressBroker uses to standardize and geocode addresses. When setting path properties, Precisely suggests you list all your geo-demographic data. Use a unique logical name for each data set. Values for path properties take two forms: — A delimited list of fully specified *directory names*: Ex. `[GEOSTAN] "C:\Program Files\Centrus\cd2tiger" | \ [GDT] "C:\Program Files\Centrus\cd2gdt"` —A delimited list of fully specified *path and file names*: Ex. `[GEOSTAN_Z9] "C:\Program Files\Centrus\cd2tiger\US.z9" | \ [GDT_Z9] "C:\Program Files\Centrus\cd2gdt\...."` The most common AddressBroker path properties are: `GEOSTAN_PATHS`—Logical name and fully specified *directory name* of each GeoStan data source, such as a GDT or Precisely Enhanced data file. `GEOSTAN_Z9_PATHS`—Logical name and fully specified *path and file name* of each GeoStan us.z9 data file.
Initialization Properties	Initialization properties specify the set of all available geo-demographic data, input fields, and output fields that your application uses. `INIT_LIST`—Delimited list of logical names you are using. `INPUT_FIELD_LIST`—List of field names that correlate to the format of your input records. Valid input fields depend upon the `INPUT_MODE` you are using. See "INPUT_FIELD_LIST Property" on page 375 for more information. `OUTPUT_FIELD_LIST`—List of fully specified field names that you retrieve from your output records. Available outputs depend on the modules included in your Precisely license agreement. See "OUTPUT_FIELD_LIST Property" on page 384 for additional information.

Optional AddressBroker properties

AddressBroker's optional path properties are:

`DEMOGRAPHICS_PATHS`—Logical name and fully specified path and file name of each Demographics Library data file. These are `dld` files you purchased from Precisely.

`SPATIAL_PATHS`—Logical name and fully specified path and file name of each Spatial+ data file. The files must be in Precisely'.`gsb` format.

GEOSTAN_C_PATHS—Logical name and fully specified path and file names of GeoStan Canada data files.

GDL_SPATIAL_PATHS—DEPRECATED. Use SPATIAL_PATHS.

GEOSTAN_Z5_PATHS—Logical name and fully specified path and file name of the zip5.gsb file used with GDL.

STATUS_LOG—Output destination of all reported messages. Set AddressBroker's STATUS_LOG property to either the path and file name for a status log to save status messages, or the value CONSOLE to display status messages to a console window.

STATUS_LEVEL—Message reporting level. Set this property to the appropriate level of message reporting you require:

- FATAL – Fatal errors, errors, and warnings.
- ERROR – Errors and warnings only.
- WARN – Warnings only.
- INFO – All information messages.
- NONE – No messages.
- DEBUG – Debug messages; for development only.
- SERVER – To report server-level-only messages. Default.

The STATUS_* properties do not require validation to be used or changed.

Setting AddressBroker path properties

Before executing AddressBroker, you must specify the location of the data used by AddressBroker's libraries. This information is passed to AddressBroker in the following properties:

- GEOSTAN_PATHS
- GEOSTAN_Z9_PATHS
- GEOSTAN_CANADA_PATHS
- DEMOGRAPHICS_PATHS
- SPATIAL_PATHS
- GDL_SPATIAL_PATHS (*Deprecated*)

These properties associate a logical name with the name and location of your reference data. The GEOSTAN_PATHS and GEOSTAN_CANADA_PATHS properties require path and *directory* names. The other path properties require path and *file* names.

The following example shows how to associate logical names to data sources and assign AddressBroker's path properties. In the following example each data set has a unique logical name and logical names are shown in **bold**.

```
GEOSTAN_PATHS      =
[GEOSTAN] "C:\Program Files\Centrus\cd2tiger" | \
[GDT] "C:\Program Files\Centrus\cd2gdt"
GEOSTAN_Z9_PATHS   =
[GEOSTAN_Z9] "C:\Program Files\Centrus\cd2tiger\US.z9" \ |
[GDT_Z9] "C:\Program Files\Centrus\cd2gdt\us.z9"
GEOSTAN_CANADA_PATHS =
[GEOSTAN_C] "C:\Program Files\Centrus\AddressBroker\data"
SPATIAL_PATHS =
[COUNTIES] "C:\Program Files\Centrus\spatial\Counties.gsb"
SPATIAL_PATHS =
[STATES] "C:\Program Files\Centrus\spatial\States.gsb"
```

Note: You can have more than one SPATIAL_PATHS properties in the abserver.ini file, with each property defining one or more logicals. If you have a large number of data sources, Precisely recommends that you use more than one SPATIAL_PATHS property to avoid errors.

Setting logical names and the INIT_LIST property

AddressBroker's INIT_LIST property sets the list of logical names (and therefore the data) that an application can access. Assigning logical names to data in the path properties is not enough; you must use the INIT_LIST property to define the data sources your application accesses.

The list of logical names assigned to INIT_LIST is a subset of the list of logical names held in the LOGICAL_NAMES property. For example, you may have multiple spatial files available, however a given application may reference only a subset of these files. AddressBroker supports the use of multiple demographics and spatial files; you can list multiple logical names in INIT_LIST from the names in GEOSTAN_CANADA_PATHS, DEMOGRAPHICS_PATHS, and SPATIAL_PATHS and. You must list only one logical name for the GEOSTAN_* paths if you are performing GeoStan processing.

INPUT_FIELD LIST and OUTPUT_FIELD_LIST

AddressBroker processes each address record as a collection of fields. Fields identify the specific input and output data associated with an input address record. Field names are not case sensitive and spaces and underscores are permitted. For example, "FIRM NAME" is equivalent to "FirmName" and "firm_name".

To optimize processing, your application defines a subset of input and output fields from the lists of all available fields in the INPUT_FIELD and OUTPUT_FIELD lists. This subset minimizes the amount of memory allocated internally by AddressBroker, and in client/server and Internet applications, the amount of data sent over your network. For example, you may submit your address data for processing to AddressBroker's GeoStan module. This makes over forty GeoStan output fields available. However, you may only be interested in returning a subset of these, for example, the address standardization fields and the geocoding fields.

Defining the INPUT_FIELD_LIST

The `INPUT_FIELD_LIST` contains a delimited list of input field values that describe the address format of the data records you want to process. The input format determines what subset of the address input fields are valid for the `INPUT_FIELD_LIST`.

You can format address information to pass into AddressBroker in the following ways:

- Normal (addressline, addressline2, lastline). The default input format is Normal.
- Parsed lastline (addressline, addressline2, city, state, ZIP Code).
- Multiline. If it is unclear what fields may contain address information (line1, line2, line3, line4, line5, line6). For additional information about the multiline input mode, see Appendix A: Advanced Concepts

  Note: In addition to address fields, you may also pass in Longitude and Latitude fields for Spatial analysis.

Defining the OUTPUT_FIELD_LIST

To fully specify output fields, logical names are paired with output field names. This pairing lets you reference multiple data sources within a single application and control the files AddressBroker uses to generate output. A field name–logical name pair may not exceed 32 bytes, including the null terminator.

The default field specifier is the logical name for your GeoStan data. The default specifier is implied, that is, you do not need to list a field specifier with GeoStan output fields. Spatial, GDL, and Demographics fields *must* be fully specified using a logical name—even if you have only one source file for each data type.

Fully specified output field names are assigned to the `OUTPUT_FIELD_LIST` property. For example, a hypothetical application requires access to two Spatial+ .gsb files, `Counties.gsb` and `States.gsb`. To determine which source file to use when returning a value, a logical name is associated with PolygonName.

Note: You must provide logical names for Spatial+, GDL, and Demographics output fields even if you are not using multiple data sources.

Decimals in input/output field values

Some AddressBroker input/output field values refer to decimal values. In the general case, values are input and output as integers; you must interpolate the decimal point.

The table below is an excerpt from one of AddressBroker's output field lists. The example shows AVGHHSZ00, defined as a numeric field up to 11 characters long, interpolated with two decimal places.

(Ex.) AVGHHSZ00 = 235 Average household size equals 2.35.

(Ex.) AVGHHSZ00 = 300 Average household size equals 3.00.

Output String Field Name	Type	Width	Decimal	Description
AVGHHSZ00	N	11	2	2000 Average Household Size
:	:	:	:	:

Decimals Exceptional case

AddressBroker's COORDINATE_TYPE property lets you specify integer or floating decimal point for the Latitude and Longitude input and output fields. These fields either have or imply values with six decimal places.

(Ex.) COORDINATE_TYPE = AB_COORD_INTEGER (default).

Latitude = 40123456 value = 40.123456.

(Ex.) COORDINATE_TYPE = AB_COORD_FLOAT

Latitude = 40.123456 value = 40.123456.

Note: Setting COORDINATE_TYPE to AB_COORD_FLOAT lets you enter or retrieve floating decimal data only for the Latitude and Longitude input and output fields. All other AddressBroker fields are entered and retrieved as integers.

5 – Client Applications

In this chapter

This chapter discusses the initial steps necessary to get your client application up and running. First this chapter explains how to access the AddressBroker library followed by a discussion of object initialization via the an initialization file. The last section in this chapter details the AddressBroker properties that Precisely recommends for client applications.

You can build an AddressBroker client application using any of AddressBroker's programming interfaces. TCP/IP sockets are used as the transport protocol for AddressBroker clients.

Installing AddressBroker

The AddressBroker client object can be installed on the platforms listed in Chapter 3 System Requirements. The ActiveX component is supported on Windows platforms only. In addition, the Java client runs on any platform with a Java 2 Standard Edition (J2SE) virtual machine.

In addition to installing the executables, you may also need to install geo-demographic data files—although typically these data files are stored on a server in client/server applications.

Backward compatibility

AddressBroker versions 1.5 and later are not backwards compatible with previous versions. However, starting with AddressBroker version 1.5, all AddressBroker version 1.5 or later clients are operational with all AddressBroker servers of the same version number or higher.

Multi-threading support requirements

The multi-threaded functionality of AddressBroker requires the multi-threaded support library. Clients developed for AddressBroker and installed on machines other than the development machine require installation of the development environment's libraries that support multi-threaded applications. For example, clients developed using the Microsoft Visual C/C++ environment and installed on machines without the development environment require that `msvcrt.dll` be installed with the client.

Input/Output address records

AddressBroker is a record-based application. The information you provide is entered in terms of individual records, and the output data you retrieve is record-based.

AddressBroker can process input data from a variety of sources. Your application can read records from your own customer database, prompt a user to enter an address, or you can specify the data within your application.

Managing records

AddressBroker provides full functionality for managing records. Use the following functions to manage your input records:

- `SetField` sets the value of an input field in the current input record. This data is used as input for processing (validation, standardization, and enhancement).
- `SetRecord` adds data for the current input record (the combined information provided via multiple calls to `SetField`) to the input record buffer, then advances the input record pointer to the next empty record in the buffer.

Use `ProcessRecords` or `LookupRecord` to process your records (see "Processing records" on page 71). Then use the following functions to manage the output records:

- `GetField` retrieves fields from the current output record.
- `ResetField` resets the output field pointer to the first value of a multi-valued output field.
- `GetRecord` advances the output record pointer.
- `ResetRecord` resets the record pointer to the first output record.

Processing records

You can use either of the following functions to process records:

- `LookupRecord` processes one record at a time and finds potential matches iteratively from partial address information. Use this function when address input records are incomplete. See "Valid addresses" on page 10 for more information.
- Precisely recommends using `ProcessRecords` instead of `LookupRecord`.
- `ProcessRecords` processes one or more input records for which addresses *are* complete enough for standardization. Use this function for multiple (or single) records when sufficient address information is known.

Reserved characters

`RECORD_DELIMITER`, `FIELD_DELIMITER`, and `VALUE_DELIMITER` have default values of line feed, tab, and CTRL-A, respectively. If your data contains any of these characters, you **must** reset the appropriate AddressBroker property to a different character. In addition, your data may not contain the null character.

Optimizing performance

In client applications, Precisely recommends you read in your data in small batches and submit multiple transactions, rather than submitting your entire address database for processing at once.

With a little experimentation, you can set benchmarks and optimize performance. Adjust the number of records per processing transaction depending upon the your record size, your system capabilities, and the performance results. For optimal performance in a client/server application, Precisely suggests you start with approximately 100 records per processing transaction. (Client requests are limited to 16 MB of data, or roughly 100-250K records depending on what information is being transmitted.)

Initializing a client application

Client objects require initialization. Initialization involves assigning values to a short list of AddressBroker properties required by all client applications. This can be done programmatically or via an initialization (.ini) file based on the AddressBroker Interface Language.

Note: The AddressBroker .NET and Java APIs do not support the use of an initialization file.

Initializing with an initialization file

AddressBroker C, C++, and ActiveX clients can be initialized with an initialization file. A client initialization file contains information that controls the execution of the AddressBroker client and accesses information set on the server. Initialization files are discussed in detail in Chapter 4 Using Initialization Files.

The example below contains a code fragment for an AddressBroker client object. It shows the error handling properties and the properties required by client applications.

Typical AddressBroker property settings in a (Windows) client initialization file

```
STATUS_LOG = "C:\Program Files\Centrus\Log\status.log"
STATUS_LEVEL = DEBUG
INIT_LIST = GEOSTAN | GEOSTAN_Z9 | COUNTIES
INPUT_FIELD_LIST = FirmName | AddressLine | \
AddressLine2 | LastLine
OUTPUT_FIELD_LIST = FirmName | City | State |ZIP
```

Initializing programmatically

Client applications can also be initialized programmatically. To learn more about initializing a client object programmatically see the entries for `SetProperty` and `ValidateProperties` in the appropriate API section of this manual.

For examples showing how to initialize an AddressBroker client object programmatically see the following sections:

- “AddressBroker Java tutorial” on page 101.

- “.NET API” on page 141.
- “AddressBroker C tutorial” on page 191.
- “C++ API” on page 230.
- “AddressBroker ActiveX tutorial” on page 285.

Configuring clients for use with multiple servers

There is no special initialization on the client side to use multiple servers other than specifying that there are multiple servers. AddressBroker transparently switches between servers if a client has a problem establishing communication with its current server.

AddressBroker properties—client applications

For client applications, you need not set any AddressBroker properties; all can be set on the server (see “AddressBroker properties in server applications” on page 63). However, Precisely recommends at minimum that you set the following AddressBroker initialization properties that govern the list of logical names and input/output field names your application uses:

- `INIT_LIST`
- `INPUT_FIELD_LIST`
- `OUTPUT_FIELD_LIST`

Precisely also recommends setting the error reporting properties. Error reporting on the client is independent of error reporting on the server. After the basic properties are set, you can run your application using the default values for other AddressBroker properties, or configure them to better suit your processing requirements.

In all of the AddressBroker APIs, you can initialize AddressBroker client objects programmatically. In the C/C++ APIs and the ActiveX interface, you can also use an initialization file based on the AddressBroker Interface Language. The AddressBroker .NET and Java APIs do not support the use of an initialization file.

AddressBroker properties are referenced either by their character string property names—”`MIXED CASE`”—or by property IDs—`AB_MIXED_CASE`. Using Property IDs yields slightly faster performance, and permits error checking at compile time.

Managing AddressBroker properties

AddressBroker properties share many common features. They can all be set using the **`SetProperty`** function. Use **`GetProperty`** to obtain current property values. Use **`GetPropertyAttribute`** to retrieve general property information, such as size and type.

Assigning values to process control properties

You can assign process control properties as follows:

- **In an application using API function calls**. You can make repeated calls to `SetProperty` throughout your client application as needed.
- **In the ActiveX interface, with AddressBroker ActiveX properties**. AddressBroker's ActiveX interface includes properties that have a 1:1 correspondence with AddressBroker's properties.

Verifying properties

After properties have been assigned, they should be validated. Use `ValidateProperties` to ensure that a complete set of legal property settings is available to AddressBroker. `ValidateProperties` should be called successfully *after* you are finished setting properties and *before* you begin entering input records and field values or record processing begins.

Making a property assignment is only the first step in a two-step process. Many properties depend on the values of other AddressBroker properties. For example, ALL_OUTPUT_FIELDS, which holds a list of all available output fields, depends on the INIT_LIST property, among others. The ALL_OUTPUT_FIELDS property contains an unpredictable value until you have validated the properties it depends on.

It may not be possible to validate all properties in a single function call. In an interactive application, for example, you may want to let the user select some of the properties dynamically. Based on the user's selection, you may subsequently set other properties and then revalidate all properties. In this scenario, initial calls to the validate properties function returns **FALSE**. This does not mean that an error has occurred. It means that AddressBroker requires more information.

If all the information required to set AddressBroker's properties is not available, you can use iterative calls to the validate properties function to set properties incrementally. Each call validates some properties, making additional information available to you. Use the information from each call to set additional properties until all AddressBroker properties have been set and validated.

Follow these guidelines:

1. Use the get property functions to retrieve the value of the LOGICAL_NAMES (read-only) property.
2. In client applications, the LOGICAL_NAMES (read-only) property is available as soon as you create an AddressBroker client object. This is because the *_PATHS properties are set and validated on the server.
3. Use the information from the LOGICAL_NAMES (read-only) property to set and validate the INIT_LIST property.

4. Set and validate the INPUT_MODE property.
5. Now you can call the get property functions to retrieve the values in the ALL_INPUT_FIELDS and ALL_OUTPUT_FIELDS properties. These properties hold all of the input and output field names available to your application.
6. Use the information from the ALL_INPUT_FIELDS and ALL_OUTPUT_FIELDS (read-only) properties to set and validate the INPUT_FIELD_LIST and OUTPUT_FIELD_LIST properties. These properties are assigned the fields your application actually uses. Field names must be a subset of the names found in the ALL_INPUT_FIELDS and ALL_OUTPUT_FIELDS properties, respectively.
7. Next, get general information about the input and output fields your application uses by calling the get field attribute functions. Use this information to set and validate all remaining AddressBroker properties.

After all AddressBroker properties have been successfully validated, all of AddressBroker's methods become available.

Getting information about properties

You can retrieve information about individual AddressBroker properties including default settings or current value.

The **GetProperty** function returns the current value of the AddressBroker property given as its argument.

The **GetPropertyAttribute** function returns information about the AddressBroker property given as its argument. For example, you can use this function to determine a property's type (such as read-only) or its default value.

Error handling properties

While you are not required to set these properties, Precisely recommends setting AddressBroker's status reporting properties—STATUS_LOG and STATUS_LEVEL, or THROW_LEVEL.

These properties do not require validation to be used or changed, and should be set first to ensure that no status messages are lost. AddressBroker supports error reporting for all application types; however, the implementation varies by language. See the section titled "Errors, Messages, and Status Logs" in the relevant API chapter in Section 3 of this manual for language-specific details on this topic.

Note: A client cannot override error handling AddressBroker properties when they are set on the server.

Required AddressBroker initialization properties

Precisely requires that you set the initialization properties listed below on the client to specify the subset of all available geo-demographic data, input fields, and output fields that your application uses. When set on the client, these properties override default values set on the server.

INIT_LIST—Delimited list of logical names you are using.

INPUT_FIELD_LIST—List of field names to use from your input records. The field names allowed depend on the INPUT_MODE you are using.

OUTPUT_FIELD_LIST—List of fully specified field names that you retrieve from your output records. Available outputs depend on the modules included with your Precisely license.

Field list properties

The following code fragment shows how to set AddressBroker's field list properties programmatically. These properties may also be set in an initialization file.

Specifying input and output field lists in a C++ application

```
    //constructor
QMSAddressBroker *ab = QMSAddressBroker::CreateClient (
“primary:1234 | secondary:1235”, “socket”, “MyLogon”, “MyPassword”,
“MyInitFile” ) ;
  ...
// input fields are Firmname, AddressLine, Lastline

ab->SetProperty ("INPUT_FIELD_LIST", "FirmName | AddressLine | LastLine"
);

// output field names are Firmname, AddressLine, City, State, ZIP
```

Other recommended properties

Precisely also recommends that you set the following properties:

- MATCH_MODE—Controls the “closeness” of the matched records. Set MATCH_MODE to **AB_MODE_CLOSE** for best results. See “Pre-defined property values” on page 354 for more information.
- KEEP_MULTIMATCH—Specifies whether a single match or multiple matches are returned. The RecordID input and output fields help correlate *input* records with their corresponding *output* record(s).
- KEEP_COUNTS—Specifies whether match criteria counts are kept. To keep counts, set KEEP_MULTIMATCH to **false** and KEEP_COUNTS to **true**. Keeping counts increases processing time.

Logical names—client applications

AddressBroker uses logical names to abstract the details of AddressBroker's reference data file names and locations. The following sections discuss using logical names in a client application. For overview information about logical names, see "Logical names" on page 62.

Logical names and the LOGICAL_NAMES property

If your application is a client, the first thing it needs to know is the list of valid logical names defined on the server. AddressBroker's read-only `LOGICAL_NAMES` property contains a list of logical names available to your application. Any references your application makes to logical names must match the names found in this list. Use a **`GetProperty`** function call with the `LOGICAL_NAMES` property as its argument to retrieve the list. An example using **`GetProperty`** is shown below. Each logical name returned is coded for the following types:

- C for GeoStan Canada
- D for Demographics
- G for GeoStan
- S for Spatial
- Y for GDL Z5 (zip5.gsb)
- Z for GeoStan Z9

Retrieving a list of logical names using the LOGICAL_NAMES property (C++ example)

```
// assume the path assignments from the figure above
    :
ab.GetProperty ( "LOGICAL_NAMES", buffer, buffersize );
printf ( "%s", buffer );
//printf output would look like this:
GEOSTAN:G \tGDT:G \GEOSTAN_Z9:Z \tGDT_Z9:Z \tGEOSTAN_C:C \tCENSUS2K:D
\tCOUNTIES:S \tSTATES:S \tSTATES2:S
```

Logical names and the INIT_LIST property

AddressBroker's `INIT_LIST` property sets the list of logical names (and therefore the data) that an application can access. Assigning logical names to data in the path properties is not enough; you must use the `INIT_LIST` property to define the data sources your application accesses.

The following example assumes the property settings shown in the example in "Logical names and the LOGICAL_NAMES property" on page 77. The logical names are shown in bold.

Note: Precisely only assigns a subset of the logical names to the `INIT_LIST` property. As a consequence, the application can access Precisely Enhanced data, but not GDT data. It can also access two spatial data files, but no demographic data.

Assigning logical names to the INIT_LIST property in a Java client

```
   :
ab = QMSAddressBrokerFactory.make( host + ":" + Integer.toString(port),
"SOCKET", MyLogon, MyPassword );
   :
ab.setProperty( "INIT_LIST", "GEOSTAN | GEOSTAN_Z9 | COUNTIES | STATES |
GEOSTAN_C | STATES2" );
   :
```

Input/Output fields

AddressBroker processes each address record as a collection of fields. Fields identify the specific input and output data associated with an input address record.

Single and multi-valued fields

AddressBroker has two field types: single-valued and multi-valued. Single-valued fields, as the name implies, contain a single value. Examples of single valued fields include `AddressLine` and `City`. To retrieve values from single-valued fields, call the **`GetField`** method.

Depending on the API language you are using, there may be more than one syntax allowed for referencing logical names in **`GetField`**.

To determine if a field requires a logical name, use **`GetFieldAttribute`** with `AB_FIELD_NEEDS_LOGICAL_NAME` as its argument. The value returned in this property also indicates the data type of the required logical name (for example, GeoStan or Spatial+™).

Multi-valued fields, as the name implies, contain multiple values. Some AddressBroker *spatial output fields* may contain multiple values. To retrieve values from multi-valued fields, use repeated calls to **`GetField`**, such as in a `while` loop.

For example, in a hypothetical spatial query you would use `OurSalesTerritory.gsb` to determine the sales territory an address falls within. There is some overlap in these sales territories, consequently it is possible that a given address lies within more than one sales territory (polygon). The names of all the polygons the address falls within (or near) can be retrieved from the multi-valued field `PolygonName`.

Precisely may also get multiple values when doing a closest site spatial query. In this type of query, you want to identify the closest point(s) to an address. In this hypothetical spatial query, use `OurStoreLocations.gsb`, and set AddressBroker properties to return the five closest sites. The names of these sites (points) can be retrieved from the multi-valued field `ClosestSiteName`.

See the **`GetField`** method description in each API chapter in this document for code fragments showing how to retrieve values from multi-valued fields.

Managing fields

AddressBroker fields share many common features. They can all be set using the **`SetField`** function. Retrieve general information about fields using the **`GetFieldAttribute`** function. Retrieve current field values using the **`GetField`** function.

The `SetField` function sets the value of an input field in the current input record. This data is used as input for processing (validation, standardization, and enhancement).

The `GetField` function gets the value (or values if the field is multi-valued) of an output field from the current output record.

Use `ResetField` to reset the pointer to the first value in multi-valued output fields in the current output record.

Use the `GetFieldAttribute` function to query information about an input or output field, such as its type, width, and number of decimal places.

Guidelines for using fields

Use the following guidelines when defining fields:

- **Specify your data**—Before processing your address records, specify only those fields your application uses. The field names that are available to your application depend on the values set in AddressBroker's INIT_LIST and INPUT_MODE properties. Be sure to confirm these properties with `ValidateProperties`.
- **Get the lists of available fields**—After AddressBroker knows which reference files it can access, it can provide lists of all the valid input and output fields that are compatible with its data sources. You can retrieve these lists using `GetProperty` with ALL_INPUT_FIELDS or ALL_OUTPUT_FIELDS as its argument.
- **Select the input fields to use**—After you have the available fields list, select the fields that your application uses and assign AddressBroker's INPUT_FIELD_LIST property, specifying the list of input field names to be set in each AddressBroker record.
- To optimize processing, specify only those fields you want processed and returned, even if your input address records contain additional address information.
- Many input records can be standardized even if they do not include information for every field in INPUT_FIELD_LIST.
- **Select the output fields to use**—The OUTPUT_FIELD_LIST property specifies the list of output fields returned upon completion of processing.

```
ab->SetProperty ("OUTPUT_FIELD_LIST" FirmName | AddressLine | City |
State | ZIP );
```

6 – Server

In this chapter

This chapter discusses the initial steps necessary to get your AddressBroker server up and running. First, this chapter discusses issues specific to running AddressBroker server on Windows and discusses issues specific to running AddressBroker server on UNIX platforms. The remainder of the chapter discusses issues common to running the server on all platforms, including sections on the AddressBroker properties the server requires and error reporting for AddressBroker server.

Installing AddressBroker

You can install the AddressBroker server on the platforms listed in Chapter 3 System Requirements.

In addition to installing the executables, you may also need to install geo-demographic data files. For instructions on installing AddressBroker, please refer to the installation notes. AddressBroker classes are shipped as a Java JAR file and several Windows DLLs. UNIX classes are shipped as static or dynamic libraries.

Backward compatibility

AddressBroker versions 1.5 and later are not backward compatible with versions previous to 1.5. However, starting with AddressBroker version 1.5, all AddressBroker version 1.5 or later clients are operational with all AddressBroker servers at the same version number or higher.

Windows server administration

The following sections discuss installing, starting, and troubleshooting the AddressBroker server, and accessing remote data.

AddressBroker Service Manager

Use AddressBroker Service Manager (ABSM) to install and make registry entries for the AddressBroker Service after you install AddressBroker. You can also edit server initialization files using the ABSM. You do not need to set any registry parameters to run the ABSM.

Step 1: Using ABSM to install the AddressBroker service

Note: If the service is already installed, proceed to step 2, below.

1. To start ABSM, select the Windows Start menu and select Programs --> AddressBroker --> AddressBroker.

The first time you open the ABSM, the AddressBroker Service Manager dialog box appears.

Default settings are provided for all of the fields except the initialization file. The stoplight shows red indicating the service is not fully configured. Throughout the installation and configuration steps, click the stoplight button for help as needed.

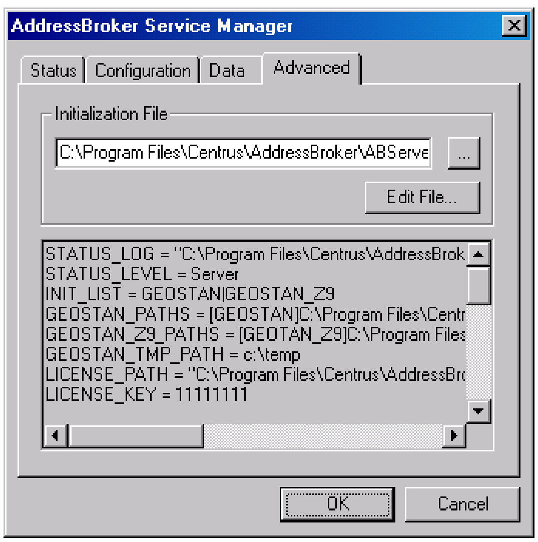
2. Click **Install** to register the AddressBroker server as an NT service.

After selecting **Install**, the button changes to **Start**. To unregister a service, click the **Remove** button. The AddressBroker Service unregisters from the set of NT services, however, no files are removed from your system.

After installing the service, the stoplight turns yellow, indicating you may now start the service.

Step 2: Indicating the server initialization file

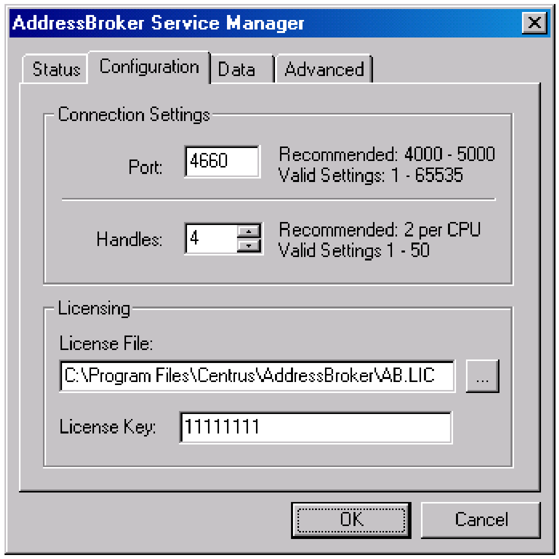
Before running the AddressBroker Service, indicate a server .ini file. For information about creating .ini files, see "Using Initialization Files" on page 58



- In the Initialization File field, type the path and file name of the server .ini file. Click **Browse** to navigate to the file.
- To edit the .ini file, click **Edit File** to open the .ini file in a text editor.

Step 3: Defining the registry settings

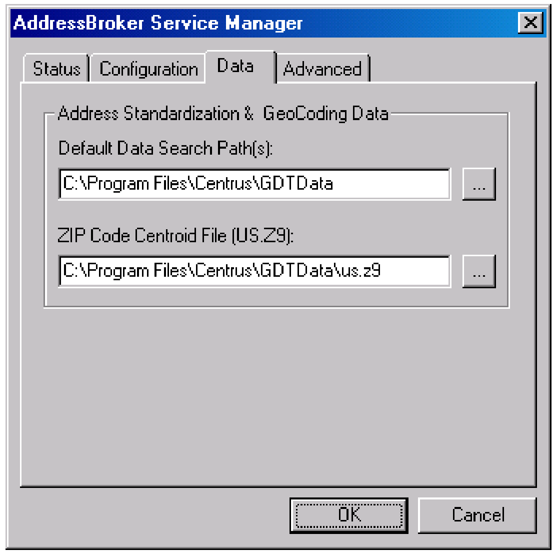
To define the registry settings, select the **Configuration** tab and provide the following information:



Field Name	Description
Port	The port number to run the service. Precisely suggests setting the port to a number between 4000 and 5000.
Number of Handles	The number of handles the server will create for processing. The server is a multi-threaded application. There is one thread per handle. You can run multiple requests simultaneously by running multiple handles. Each handle requires approximately 15 MB of memory (average). If your processing requests include both large and small numbers of address records, increasing the number of AddressBroker instance handles allows the smaller transactions to be processed more quickly. Precisely recommends using 2 to 3 handles per processor for light to normal loads, 5 to 10 handles for heavy processing. Use more handles as system resources or processing requirements merit.
License File	When AddressBroker's server component was installed, the license file was copied to a local directory. Specify the path to the license file which is typically copied to a local directory when AddressBroker's server component is installed.
License Key	The license key (that is, password) provided with your license file.

Step 4: Verifying data paths

Select the **Paths** tab and verify the path to your GeoStan and GeoStan Z9 data sets. You may edit the paths directly here, or from the **Edit File ...** button on the **Advanced** tab.



Note: If you are supporting more than one GEOSTAN or GEOSTAN_Z9 path from the AddressBroker server, you must edit them from the **Edit File...** button on the **Advanced** tab.

Step 5: starting the service

Return to the **Status** tab and click **Start** to start the AddressBroker Service.

If the Service starts correctly, the stoplight turns green. When the Service is running, the **Start Service** button becomes **Restart**, and the **Remove** button becomes **Stop**. Selecting this button stops the AddressBroker server.

To query the status of the service, press the **Stop Light** button. A message displays with information about your AddressBroker Service.

Troubleshooting AddressBroker server

The first step in debugging the AddressBroker server is to check your log file. The log file records error information as reported by the server. The name of the log file is specified in the server .ini file you provided to ABSM. If you did not specify a log file in your .ini file, edit the file to include lines that look something like this:

```
STATUS LOG        = YourFullPathTo\Server.log

STATUS LEVEL       = SERVER
```

Note: AddressBroker automatically rolls over the log file at 2GB.

A common problem preventing the AddressBroker server from starting is that access permissions to necessary geo-demographic data are not set correctly.

AddressBroker also includes a command line interface, called `abserver`, to assist you in debugging the AddressBroker server. If you get an error message when you try to run the AddressBroker Service, follow these steps:

1. From the AddressBroker Service Manager window, verify that you have filled in the required fields for the registry—port number, Number of Handles, Maximum Concurrent Connections, and the full path and filename of your AddressBroker server initialization file. **The command line interface does not execute properly unless these registry values have been entered and the service has been installed**.
2. From the Start menu, select Programs and open a Command Prompt (DOS) window.
3. In the DOS window, navigate to the directory that contains abserver.exe. A standard installation puts this file on your local drive in:
   ..\Program Files \Centrus\AddressBroker\bin directory.
4. `abserver` runs as a console application, not as an NT service. Error messages are printed to the screen. Use Ctrl-C to terminate the application.
5. From the command line prompt, type:

```
abserver -debug
```

A list of one or more status codes and messages displays on the screen. Use this information to debug the AddressBroker server. A typical status message is that the initialization file was not found. Check the path specified and attempt to restart the server. **Debug mode does not execute properly unless the service has been installed and the registry entries have been made**.

The type of status messages `abserver` reports depends on the value assigned to AddressBroker's `STATUS_LEVEL` property. **DEBUG** causes all errors and warnings to be reported. The default value—**SERVER**—causes only critical errors to be reported.

If you received an error running the server through AddressBroker Service Manager, but running `abserver -debug` works error free, you may have not have your network or file permissions set correctly.

UNIX server administration

AddressBroker uses dynamically linked libraries. You need to set your PATH variable to include the AddressBroker lib directory. Depending on the operating system and the shell you are running, the assignment statement looks similar to the following example:

```
SHLIB_PATH = $SHLIB_PATH: /YourFullPathToAddressBroker/libs
LD_LIBRARY_PATH = $LD_LIBRARY_PATH: /YourFullPathToAddressBroker/libs
```

You can set this variable on the command line or in your profile—or the equivalent—file.

The abserver command

To run the AddressBroker server on a UNIX platform, use the abserver.rc script command:

```
ksh abserver.rc start
```

Accessing remote data on UNIX platforms

The user running the **abserver** process may encounter NFS permission difficulties when AddressBroker geo-demographic data is mounted remotely. Ensure that the owner of the server process has read permissions to the files that AddressBroker accesses remotely. If **abserver** does not execute properly, check your file permissions.

Starting the abserver at boot time

To start the **abserver** at boot time, include a script in your /init.d directory. For example:

1. Copy the `abserver.rc` script from the Samples subdirectory to /sbin/init.d/abserver.
2. Edit the script to set the full paths to `abserver.ini` and the abserver executable. These are shown in bold in <Hypertext>“abserver.rc sample file.” You may also want to edit the PORT, NHANDLES, and MAXCONN values.
3. Create a symbolic link from /sbin/rc2.d/S900abserver to /sbin/init.d/abserver.
4. Create a symbolic link from /sbin/rc1.d/K900abserver to /sbin/init.d/abserver.

Assign a number such that the abserver script starts after any required support services (for example, NFS or TCP/IP) have already started. The number in the .rc scripts (for example, 900) must not conflict with any existing script number. The following section shows an example start up script.

abserver.rc sample file

```
#!/bin/sh
##########  CONFIGURATION PARAMETERS  ##########
PORT=1234
FILE=/YourFullPathTo/abserver.ini
NHANDLES=6
MAXCONN=20
PATH=/sbin:/usr/sbin:$PATH
export PATH
# start-up script exit codes
OKAY=0
ERROR=1
##########  main  ##########
case $1 in
   start_msg)
      print "Start Centrus AddressBroker daemon"
      exit $OKAY
      ;;
   stop_msg)
      print "Stop Centrus AddressBroker daemon"
      exit $OKAY
      ;;
   stop)
      abserver -shutdown
      exit $OKAY
      ;;
   start)
      ;;  # fall through
   *)
      print "USAGE: $0 {start_msg | stop_msg | start | stop}" >&2
      exit $ERROR
      ;;
   esac
##########  start  ##########
YourFullPathTo/abserver $PORT $FILE $NHANDLES $MAXCONN
exit $?
```

System resources and AddressBroker UNIX servers

When running an AddressBroker server, be aware of the Interprocess Communication (IPC) resources, (message queues) being used. If you interrupt the process, the server crashes, or you fail to call the appropriate termination method, you may inadvertently continue to consume system resources. To check from the command line call:

```
ipcs
```

which produces output like:

```
IPC status from /dev/kmem as of Tue Oct  6 10:04:14 1998
T      ID      KEY        MODE        OWNER     GROUP
Message Queues:
q       0 0x3c180475 -Rrw--w--w-      root      root
q       1 0x3e180475 --rw-r--r--      root      root
q      52 0x00000000 --rw-------      UserID    qms
q      53 0x00000000 --rw-------      UserID    qms
Shared Memory:
m       0 0x2f140002 --rw-------      root      sys
m       1 0x411802e2 --rw-rw-rw-      root      root
...
```

The lines with your user name listed as owner may include AddressBroker system resources. To remove them manually, use:

```
ipcrm -q <number>
```

where `<number>` is the ID. To clean up the example above, issue this command:

```
ipcrm -q 52 -q 53
```

You need to specify the -q for each ID ("ipcrm -q 52 53" is incorrect).

A simple shell script can automate this process for you. For example:

```
#! /bin/csh
set id= `whoami`
set queues= `ipcs -q | grep $id | awk '{print $2}'`
foreach q ($queues)
-ipcrm -q $q
end
ipcs -q
```

Troubleshooting the AddressBroker server

The first step in debugging the AddressBroker server is to check your log file. The log file records error information as the server reports. The name of the log file is specified in the initialization file you provided to abserver. If you did not specify a log file in your initialization file, edit the file to include lines similar to the following:

STATUS LOG   = YourFullPathTo/Server.log

STATUS LEVEL = DEBUG

A common problem preventing the AddressBroker server from starting is that access permissions to necessary geo-demographic data are not set correctly.

Usage statement for abserver in debug mode

```
Usage:  abserver [-debug] [-t tmpdir] port inifile [nHandles [maxConn]]
        abserver -shutdown
           -debug    Run with debug output to the console,otherwise
                     run in the background (as a UNIX style daemon).
           -t tmpdir Specifies the scratch directory. if the -t flag is not
                    specified on the command line, the $TMPDIR ENV variable
                    is used. If $TMPDIR is unspecified, /tmp is used as the
                     default. Default = /tmp
           port      The decimal port number. Required parameter.
           inifile   The AddressBroker default file. Required parameter.
           nHandles  Number of AddressBroker instances to create.
                     Default = 6.
           maxConn   Maximum number of outstanding open connections
                     attached to the service. These are open to a client
                     but not necessarily currently running a command.
                     Default = 20.
          -shutdown  Halts the abserver. In multi-server situations, the
                     -shutdown parameter halts the last abserver started.
```

The abserver keeps track of the last server started. Calling **abserver** with the **-shutdown** parameter halts the last server started. The user calling:

abserver -shutdown must have the appropriate permissions to kill the abserver process.

When running multiple servers, identify the process IDs and terminate all related processes using the UNIX **ps** and **kill** commands.

Using multiple servers

AddressBroker supports the use of up to one hundred servers for any individual client. Multiple server support ensures continuous support for your client applications. AddressBroker transparently switches between servers if a client has a problem establishing communication with its current server. An AddressBroker client uses its primary server until that server fails, at which point it switches over to a secondary server. It continues to use this secondary server until it—the secondary server—fails. After a failed server is operational, it again becomes available to the client.

To ensure all client requests are serviced identically, make certain that the server initialization file on each host uses the same initialization settings. For more information about configuring AddressBroker for use with multiple servers, see the following sections:

- Java “make” on page 110.
- .NET “Make” on page 159.

- C “QABInit” on page 196.
- C++ “createClient” on page 238.
- ActiveX “InitializeX” on page 294.

7 – Batch Application

In this chapter

The AddressBroker batch application provides geographic standardization, spatial analysis, and geographic determination in one simple application. Using the batch application, you can input a fixed width or delimited file and customize your output through a configuration file.

This chapter contains information on using the AddressBroker batch application, including:

- Formatting your input files
- Creating the configuration file
- Starting the batch application

Formatting your input files

You can format your input files using the Centrus Data Formatter utility. Input field names must correspond to valid AddressBroker input fields found in "Tables of input fields" on page 392. For information on the Centrus Data Formatter utility, see the *GeoStan Geocoding Suite Utilities Reference Manual.*

The input file must have an associated format file (.fmt) of the same name. You can generate the format file using the Centrus Data Formatter utility.

The following is an example input file.

```
//
//File Type
//
TYPE=Fixed
EOL=None

//
//Table Schema
//
AddressLine          Character          60
Lastline             Character          60
junk                 Character          47
Longitude            Character          10
spacer               Character          1
Latitude             Character          10
therest              Character1         95
```

Creating the configuration file

To use the AddressBroker batch application, you need a configuration file (.ini). If you do not specify a configuration file, the application uses the `abbatch.ini` file in the working directory.

Configuration parameters

The following are configuration parameters you can include in your configuration file.

Parameter	Description
ADDR_POINT_INTERP	Address point interpolation uses a patented process that improves upon regular street segment interpolation by inserting point data into the interpolation process. TRUE = Sets find property GS_FIND_ADDR_POINT_INTERP to true in geostan. FALSE = Turns this option off.
ALTERNATE_LOOKUP	To enable firm name matching: 1 = Matches to the address line, if a match is not made, then GeoStan matches to the Firm name line. 2 = Matches to the Firm name line, if a match is not made, then GeoStan matches to address line. 3 = (Default) Matches to the address line.
ALWAYS_FIND_CANDIDATES	Enables AddressBroker to keep multiple candidate records when matching with point-level data for use with centerline matching. Used to return multiple candidate records when street locator matching is enabled. Additional information can be obtained about matching street segments for both a single or multiple match. Not valid when using the reverse geocoding options. TRUE = Keep candidates FALSE = (Default) Do not keep candidates
APN_DATA	Specifies whether Centrus APN data should be loaded. TRUE – Centrus APN data returns available FALSE – Centrus APN data returns not available If you do not set the value, the application uses the server value.
APPROX_PBKEY	When using the Master Location Dataset (MLD), when a match is not made to an MLD record, this feature returns the pbKey of the nearest MLD point location. The search distance ("RevGeoSearchDistance") for the nearest MLD point location can be configured to 0-5280 feet. The default is 150 feet. This type of match returns a pbKey with a leading ‘X’ rather than a ‘P’, for example, X00001XSF1IF. For more information, see “PreciselyID Fallback” on page 20. TRUE – Enables PBKey Fallback. FALSE – (Default) Disables PBKey Fallback.
BATCH_SIZE	Number of records for the application to process in a single request to the AddressBroker server. Default = 100; must be 1 or greater.
BUFFER_RADIUS	Spatial buffer in feet; radius or width. If you do not set the value of the buffer, the application uses the server value set using the AddressBroker property BUFFER_RADIUS.
CENTERLINE_OFFSET	Distance, in feet, to offset the centerline geocode from the street centerline toward the parcel centroid. **Default** is 0 feet, which returns the street centerline geocode. Any value which takes the geocode past the parcel centroid will return the parcel centroid.Range = 0 - 5280.

Parameter	Description
CLOSEST_POINT	Specifies whether matching should be done to the closest feature or point address. TRUE – Matches to the closest point address within the search radius. FALSE – *(default)* Matches to the closest feature including street segments and intersections in addition to address points. **NOTE:** This feature requires that at least one points data set and one streets data set are loaded; otherwise, the match will be made to the closest feature.
DELIMITED_FILE_QUALIFIER	Field qualifier for output file types of DELIMITED. The qualifier is commonly used for delimited files when the actual delimiter character is contained in the delimited field. The possible values are DOUBLEQUOTE (*default*), SINGLEQUOTE, or NONE.
ELEVATION_DATA	Specifies whether Centrus Points parcel elevation data should be loaded. TRUE – Centrus Points parcel elevation data returns available FALSE – Centrus Points parcel elevation data returns not available If you do not set the value, the application uses the server value.
FIRST_LETTER_EXPANDED	Some business locations are identified by address ranges and can be geocoded to the interpolated mid-point of the range. TRUE = Sets find property GS_FIND_ADDRESS_RANGE to true in geostan. FALSE = Turns this option off.
INIT_LIST	List of logical names the application can access at the server. If you do not specify a list of logical names, the application uses the server value set using the AddressBroker property INIT_LIST.
INPUT_FILE	Input file containing the address or spatial inputs for processing. Requires an associated format file, of the same name with a .fmt extension, that describes the input fields and input file format. You MUST specify an input file or the application terminates.
INPUT_MODE	Indicates the valid inputs. Possible values include: NORMAL – *(default)* Uses a single field for lastline information when processing addresses. PARSED_LASTLINE – Uses multiple fields for lastline information when processing addresses. PREDICTIVE_LASTLINE - Uses input fields AddressLine and Latitude and Longitude. For more information about this feature, see “Using predictive lastline” on page 33. REVERSE_APN – Uses input fields of ApnId, CountFips & StateFips for reverse APN lookup. **NOTE:** Reverse APN matching is only available with Centrus Points and Centrus APN data. This feature is not supported using MLD and MLD Extended Attributes data. REVERSE_GEOCODE – Uses input fields Latitude and Longitude for reverse geocoding. REVERSE_PBKEY - Uses input field pbKey for Reverse PBKey Lookup. For more information about this feature, see “Reverse PreciselyID Lookup” on page 21. SPATIAL_ONLY - Uses input fields Latitude and Longitude. Performs spatial lookups only.
INPUT_OUT	Appends input record to the beginning of the output record. Yes – (*default*) Prepends the input record No – Does not prepend the input record

Parameter	Description
LOG_FILE	Name of the log file. If you do not specify a log file, the log output is sent to the console.
MATCH_CODE_EXTENDED	Specifies whether to return the extended match code (3rd hex digit). TRUE = Return extended match code FALSE = **Default.** Extended match code disabled
MATCH_MODE	Sets the leniency used to find a match. RELAX CLOSE EXACT INTERACTIVE CASS CUSTOM **NOTE:** The CASS and CUSTOM match modes are not supported in single-line address matching. If you do not set the value of the match mode, the application uses the server value. For more information on match modes, see “Address match methodology” on page 12.
MIXED_CASE	Determines how the output displays. TRUE – Mixed case FALSE – All upper case If you do not set the value, the application uses the server value.
MUST_MATCH_ADDR_NUM	Candidates must match house number exactly. Usable match modes: Custom Boolean. Default value = True When AddressBroker matches an input address, its default behavior is to match to the address number. This default behavior corresponds to "MUST_MATCH_ADDR_NUM" set to True. If "MUST_MATCH_ADDR_NUM" is set to False, then AddressBroker no longer must match the address number, therefore permitting relaxed address number matching. TRUE = Sets find property GS_FIND_MUST_MATCH_ADDRNUM to true in geostan. FALSE = Turns this option off.
MUST_MATCH_CITY	Candidates must main address exactly. Usable match modes: Custom Boolean. Default value = FALSE
MUST_MATCH_MAINADDR	Candidates must match city. Usable match modes: Custom Boolean. Default value = FALSE
MUST_MATCH_STATE	Candidates must match state. Usable match modes: Custom Boolean. Default value = FALSE
MUST_MATCH_ZIPCODE	Candidates must match ZIP code. Usable match modes: Custom Boolean. Default value = FALSE
OUTPUT_FILE	Output file name. You MUST specify an output file or the application terminates.

Parameter	Description
OUTPUT_FILE_DELIMINATOR	Delimiter for output files where the OUTPUT_TYPE is DELIMITED. Valid values are COMMA (*default)*, SEMICOLON, TAB, SLASH, or OTHER. If you specify OTHER, you must include the ASCII character after OTHER.
OUTPUT_FLN	Indicates if the first line of the output contains field names. YES – Contains field names NO – (*default*) Does not contain field names
OUTPUT_TYPE	Type of output. FIXED (*default)* DELIMITED If you specify FIXED, you must provide a width for the output fields. See “Output fields” on page 98.
REPORT_FILE	Name of the statistical output report, which includes information such as the location code and match code. If you do not specify a report file, the application does not collect the statistics.
SERVER	Server name and port. The default is `localhost:4660`.
STREET_CENTROID	Specifies whether or not to return a street segment geocode as an automatic geocoding fallback. TRUE = Return street segment geocode FALSE = **Default.** Street locator disabled
ZIP_PBKEYS	Specifies whether PBKey ZIP Centroid Locations data should be loaded. TRUE – PBKey ZIP Centroid Locations returns available FALSE – PBKey ZIP Centroid Locations returns not available If you do not set the value, the application uses the server value. For more information, see “PreciselyID ZIP Centroid Locations” on page 17.

Configuring reverse geocoding in batch

To configure reverse geocoding in batch, you will need to install the GSX files and set up the server.ini file. The server.ini file needs to include the required and desired optional properties. Optional properties specific to reverse geocoding include `REVGEO_SEARCH_DISTANCE` and `SQUEEZE_DIST`. For more information, see “Using Initialization Files” on page 58 and “Properties” on page 339. In addition, the `INPUT_FIELD_LIST` in the server .ini file needs to specify the input fields `Latitude` and `Longitude`.

Your input record needs to contain the longitude and latitude input fields. The input points can be specified in either decimal format or millionths of decimal degrees (-105.239771 & -105239771). The format is determined by the server .ini property `AB_COORDINATE_TYPE`.

To configure reverse geocoding in batch, set the configuration parameters in the batch .ini file, as follows:

- *Required*: To enable reverse geocoding, set: `INPUT_MODE=REVERSE_GEOCODE`

- *Optional*: To enable the optional closest point feature, assign: `CLOSEST_POINT=TRUE`
  For more information on this feature, see "Using reverse geocoding to points matching" on page 26.

  Note: The properties set in the batch .ini file override the properties set on the server.

Output fields

You must include a list of output fields and their position in the configuration file. If you do not specify output fields, the application terminates. Valid outputs include those available for GeoStan, GeoStan Canada, Spatial+, and Geographic Determination Library found in "Tables of output fields" on page 399.

Note: Spatial+ and Geographic Determination Library output fields must have an accompanied logical name given in the specification.

For each field, you must specify a position in the output record, in the format *OutputFieldName[LogicalName]occurence = position.*

If the output record type is FIXED (see OUTPUT_TYPE), you must also specify a width. For example, *OutputFieldName[LogicalName]occurence = position, width.* The width should fall within the output field width indicated in "Tables of output fields" on page 399.

Note: Spatial outputs are limited to eight occurrences and you cannot request an occurrence out of sequence. For example, you cannot select the third return without specifying the first and second return.

The following is an example of the text in the configuration file for the output fields.

```
; OUTPUT FIELD LAYOUT - specify the format for each output field, either
position and width (required for FIXED) or just position based on the
OUTPUT_TYPE
; Output Field Name[Logical Name]Occurrence = #, #
; no default
Addressline=1
PolygonName[CA-Rating]1=2
```

Starting the batch application

You start the batch application via a command line interface. You can specify a configuration file; If you do not specify a configuration file, then the AddressBroker batch application uses the `abbatch.ini` in the working directory.

To run the batch application, at the command line enter:

<path to batch application > <path to configuration file>

For example:

```
d:\AddressBroker\AbBatchApp\Abbatch.exe <config file>
```

8 – Java API

In this chapter

This chapter describes the Java API to AddressBroker in detail.

This chapter includes a tutorial using the AddressBroker Java API. The tutorial shows you how to use most of AddressBroker's functionality, yet is general enough that you can modify it for other uses. A complete method reference follows the tutorial. The final section of this chapter discusses error handling.

The naming convention for AddressBroker Java API methods is **`methodName`**.

Restrictions in the Java API

Due to restrictions imposed by Java, the AddressBroker Java API has the following restrictions:

- Using initialization files is not supported.
- Using log files is not supported.
- Using THROW_LEVEL is not supported.

Accessing the AddressBroker Java library

To use the AddressBroker Java API, you must have Java software on your client machine. To install Java, follow these steps:

1. Install the Java Developer's Kit (JDK) 1.7, or later, or the Java Runtime Environment (JRE).
2. Add the ABclient.jar file to the CLASSPATH on your client machine. To update your CLASSPATH:

   On UNIX platforms, modify your CLASSPATH environment variable to include the path to the location of the .jar file. Depending on the shell you are running, the statements you need look similar to the following:

```
CLASSPATH = $CLASSPATH: /<YourFullPathTo>/ABclient.jar
export CLASSPATH
```

Adding a .jar file to your (Windows) CLASSPATH

```
CLASSPATH = %JDK_HOME%\lib\classes.zip;
C:\Centrus\AddressBroker\Win32\lib\ABclient.jar;...
```

To use the AddressBroker client library .jar file, you must import the appropriate classes in your application source code files:

```
import java.io.*;
import java.net.*;
import qms.addressbroker.client.*;
```

You must also use the appropriate factory function call for creating an AddressBroker instance:

```
ab = QMSAddressBrokerFactory.make ( "myhost:4660",
"SOCKET", MyLogon, MyPassword );
```

AddressBroker Java tutorial

This section describes the steps necessary to develop a Java client application using the AddressBroker Java API. The example shows basic Java sample code that performs address record enhancement. It uses the firm name and address fields from the address records as input. This example standardizes the address data and augments it with city, state, and 9-digit ZIP Code information from the GeoStan Enhanced data directory.

Sample Java code (Console.java) is located in the Samples subdirectory.

Step 1: Create and initialize the client object

Java uses package import statements to allow class references without having to specify the fully qualified class name. Instead of using qms.addressbroker.client.AddressBroker, this tutorial uses QMSAddressBroker.

QMSAddressBroker is an interface definition and not a class. You cannot create a concrete QMSAddressBroker instance. Use the QMSAddressBrokerFactory helper class to create an instance for you.

Java initialization example

```
import java.io.*;
import java.net.*;
import qms.addressbroker.client.*;     // Use this to import classes

public class Simpleconsole
{
    public static QMSAddressBroker setupAB()    // a sample startup
function
    {
        QMSAddressBroker   ab = null;

        // Specify the machine name where the server is running
        // (list should be host:port|host:port)
        // We assume that the server is running on port 4660 (0x1234)
        // You may need to change the host:port pair to match
        // Assume the server runs on the local machine
        String list = "localhost:4660";

        // Specify what transport protocol to use.
        // "SOCKET"
        String          transport = "SOCKET";


        try
```

```
            {
                // Create AddressBroker object
                // Set username/password if accounting has been implemented.
              ab = QMSAddressBrokerFactory.make(list, transport, null, null);
            }
            catch (IllegalArgumentException illArg)
            {
                System.out.println("Unsupported protocol: " + transport);
                illArg.printStackTrace();
                return null;
            }
            catch (InstantiationException inst)
            {
              System.out.println("Could not create AddressBroker instance!");
                inst.printStackTrace();
                return null;
            }
```

Production code example

```
Production Version of an AddressBroker Java Client processing:
    QMSAddressBroker   ab = QMSAddressBrokerFactory.make(hostname,
"NOCONNECT", null, null);
    // Set the essential client side properties
    try {
      // Tell AddressBroker what logical Names we are planning on using
      // Here we are using a generic logical name for GeoStan.
      // Add others to the pipe-delimited list for other processing.
      ab.setProperty("INITLIST",
"GEOSTAN|GEOSTAN_Z9|GEOSTAN_Z5|GDTZIP5|COUNTIES|SOILS|PLACE|MUNI");
      // The following line would add a Polygon file with the
      // logical name "Counties"
      //ab.setProperty("INITLIST", "GEOSTAN|GEOSTAN_Z9|Counties");

      // Here we tell AddressBroker what information is going to be
      // provided. The INPUT_MODE property defines a set of Input
      // Fields that are allowed. The INPUT_FIELD_LIST property
      // defines the subset of those fields that are actually used.
      // Here we are providing the a rather minimal address
      ab.setProperty("INPUTMODE", QMSABConst.AB_INPUT_NORMAL );
      ab.setProperty("INPUTFIELDLIST", "firmname|addressline|lastline");

      // This is list of the information we expect about the records
      // we are enhancing.  For our example, we get GeoStan
      // information. If you have added more logical names to the
      // INIT_LIST property, then you need to also add corresponding
      // output fields to this list to define the values you want
      // returned.
      ab.setProperty("OUTPUTFIELDLIST",
                     "firmname|addressline|city|state|zip10|MatchCode"
                   + "|Longitude|Latitude|Location Quality Code"
                   +
"|PolygonName[GDTZIP5]|PolygonName[COUNTIES]|PolygonName[SOILS]"
                   +
"|GDLPolygonName[GDTZIP5]|GDLPolygonName[COUNTIES]|GDLPolygonName[SOILS
]"
                   +
"|PolygonOverlap[GDTZIP5]|PolygonOverlap[COUNTIES]|PolygonOverlap[SOILS
]"
                   + "|PolygonOverlap[MUNI]|PolygonOverlap[PLACE]"
```

```
                    + "|GDLPolygonName[MUNI]|GDLPolygonName[PLACE]"
                    +
"|MUID2[SOILS]|PolygonStatus[SOILS]|ConfidenceSurfaceType[SOILS]"
                    );
     // The following line would add polygon name and status returns for
the
      // Counties layer above.
      //  ab.setProperty("OUTPUTFIELDLIST",
      //          "firmname|addressline|city|state|zip10|MatchCode"
      //       + "|Longitude|Latitude|Location Quality Code"       //
GeoStan
      //       + "|PolygonName[Counties]|PolygonStatus[Counties]"); //
Spatial

      // Set properties that affect the behavior of the server

      // Only want single output record for each input record...
      ab.setProperty("Keep_multimatch", false);
      // Return geocodes in decimal degrees (instead of an integer
      // representing millionths of a degree)
      ab.setProperty("Coordinate Type", QMSABConst.AB_COORD_FLOAT );

    } catch (IllegalArgumentException illArg) {
      // Any of the following occurred:
      // * Attempt to set a non-existent property
      // * Data type mismatch (E.g. set a string property to
      //   an Integer value)
      // * value was null
      illArg.printStackTrace();
      return null;
    }

    // Now we go to the server and make sure everything is valid
    // We have successfully initialized our instance.
    return ab;
  }

  // build a few records for enhancement
  private static void fillRecords(QMSAddressBroker ab)
    throws IllegalArgumentException, AddressBrokerException
  {
    // Fill in a record...
    // a setField call with a bad field name (setField("xxx", ...))
    // or trying to set it to a null value (setField(...,null))
    // will result in an IllegalArgumentException being thrown
    ab.setField("firmname", "Group1 Software");
    ab.setField("addressline", "4750 Walnut #200");
    ab.setField("lastline", "Boulder, CO");
    // setRecord can throw an AddressBrokerException - but only if
    // setField is never called. Obviously not a problem here...
    ab.setRecord();
    // Fill in a second record...
    ab.setField("firmname", "White House");
    ab.setField("addressline", "1600 Pennsylvania");
    ab.setField("lastline", "Washington, DC");
    ab.setRecord();
  }

  private static void myProcessRecords(QMSAddressBroker ab)
        throws IOException, AddressBrokerException
```

```
  {
    ab.processRecords();        // Send it to the server...
    // For each record that comes back...
    String sSoilName;
    String sSoilName2;
    while (ab.getRecord()) {
      // appropriate processing of record here.
      // Print out the basic address
      System.out.println("Firm=" + ab.getField("firmname"));
      System.out.println("Addr=" + ab.getField("addressline"));
      System.out.println("City=" + ab.getField("city"));
      System.out.println("State=" + ab.getField("state"));
      System.out.println("ZIP=" + ab.getField("ZIP10"));
      System.out.println("Match Code=" + ab.getField("MatchCode"));
      System.out.println("Longitude = " + ab.getField("longitude"));
      System.out.println("Latitude = " + ab.getField("latitude"));
      System.out.println("Location Quality Code = " +
ab.getField("LocationQualityCode"));

  private static void LonLatProcessRecords(QMSAddressBroker ab)
        throws IOException, AddressBrokerException
  {
    ab.processRecords();        // Send it to the server...
    // For each record that comes back...
    while (ab.getRecord()) {
      // appropriate processing of record here.
      // Print out the Spatial and GDL Results
      System.out.println("GDT ZIP5=" +
ab.getField("PolygonName[GDTZIP5]"));
      System.out.println("County=" +
ab.getField("PolygonName[COUNTIES]"));
      System.out.println("Soil Name=" +
ab.getField("PolygonName[SOILS]"));
      System.out.println("GDL ZIP5=" +
ab.getField("GDLPolygonName[GDTZIP5]"));
      System.out.println("GDLCounty=" +
ab.getField("GDLPolygonName[COUNTIES]"));
      System.out.println("GDLSoil=" +
ab.getField("GDLPolygonName[SOILS]"));
      System.out.println("GDL GDT ZIP5 Overlap=" +
ab.getField("PolygonOverlap[GDTZIP5]"));
      System.out.println("GDL County Overlap=" +
ab.getField("PolygonOverlap[COUNTIES]"));
      System.out.println("GDL Soil Overlap=" +
ab.getField("PolygonOverlap[SOILS]"));
      System.out.println("GDL Place Name=" +
ab.getField("GDLPolygonName[PLACE]"));
      System.out.println("GDL Muni Name=" +
ab.getField("GDLPolygonName[MUNI]"));
      System.out.println("GDL Place Overlap=" +
ab.getField("PolygonOverlap[PLACE]"));
      System.out.println("GDL Muni Overlap=" +
ab.getField("PolygonOverlap[MUNI]"));

      System.out.println("-----------------------------");
      getPolygonReturns( ab, "SOILS" );

      System.out.println("\n\n");
    }
  }
```

Step 2: Set properties

The client application should set the following properties using the **setProperty** method:

- INIT_LIST—The list of logical names the application uses.

  Logical name and paths are set on the server. The logical names the client uses must match those set on the server. The logical names the client application uses must be defined server.ini file. See “LogicalNames” on page 329 for more information about logical names.

  In the example code shown in “Java setproperty example code” on page 106 the logical names GEOSTAN and GEOSTAN_Z9 refer to a GeoStan data directory and a GeoStan ZIP Code data file, respectively.

- INPUT_FIELD_LIST—The delimited list of field names. The allowable field names in the INPUT_FIELD_LIST are determined by your input data format and the INPUT_MODE property. See “Defining the INPUT_FIELD_LIST” on page 67 for more information about the INPUT_FIELD_LIST.

  Note: The INPUT_FIELD_LIST defined in the client application overrides any settings in the server.ini file.

In the sample code, AddressBroker uses the FirmName, AddressLine, and LastLine field values from each input record.

- OUTPUT_FIELD_LIST—The delimited list of field names to retrieve from the output records. Spatial+, GDL, and Demographics outputs require a logical name paired with the output field name. See “Defining the OUTPUT_FIELD_LIST” on page 67 for more information about the OUTPUT_FIELD_LIST.

  Note: The OUTPUT_FIELD_LIST defined in the client application overrides any settings in the server.ini file.

The sample shows how to enhance the address record with city, state, and ZIP10 information from the GeoStan data file.

You may set other properties in the client. In the example code, KEEP_MULTIMATCH and BUFFER_RADIUS are set. See Chapter 13 Properties for a detailed discussion about other properties.

Java property reference syntax

```
//setting a property using its string name

ab.setProperty ( "MIXED CASE", true );

//setting a property using its property ID

ab.setProperty ( ABConst.AB_MIXED_CASE, true );
```

```
//setting a pre-defined property using its string name

ab.setProperty ( "INPUT_MODE", ABConst.AB_INPUT_PARSED );

//setting a pre-defined property using its property ID

ab.setProperty ( ABConst.AB_INPUT_MODE, ABConst.AB_INPUT_PARSED );
```

Java setproperty example code

```
// Set client side properties
      // These properties are typically a subset of the properties listed
on     // the server.  If no properties are specified, the application
can     // access any of the properties specified in the server.ini file.
        try
        {
            // Tell AddressBroker what logical Names we are using.
          // For this example, we are doing only address standardization
and
            // geocoding so only GeoStan properties are used.

            ab.setProperty("INIT_LIST","Geostan|Geostan_Z9);

          // Here we tell AddressBroker the input record format. Although
we          // do this only once in the example, it is
          // a dynamic property so you could set it at any time, as many
        // times as you want.
          ab.setProperty("INPUTFIELDLIST", "firmname|addressline" +
                            "|lastline");

          // This is list of the output fields listed in the output record.
            ab.setProperty("OUTPUTFIELDLIST",
firmname|addressline|city|state|"+
                              "zip10|match_code|longitude|latitude");

          // Set properties that affect the behavior of the server
          // These properties will override behavior specified in the
          // server.ini file
          // Set the input mode
       ab.setProperty("Input_Mode", 0);

            // Only want single output record for each input record...
            ab.setProperty("Keep_multimatch", false);

            // 200 foot buffer instead of the default of 50
            ab.setProperty("BUFFER RADIUS", 200);
        }
        catch (IllegalArgumentException illArg)
        {
            // Any of the following occurred:
            // * Attempt to set a non-existent property
            // * Data type mismatch (E.g. set a string property to
            //   an Integer value)
            // * value was null
            illArg.printStackTrace();
            return null;
        }
```

Step 3: Validate properties (optional)

Use the validateProperties method to send the property definitions to the server for validation. When `validateProperties` returns **true**, the AddressBroker client object properties are set correctly and are ready for processing. If any property setting is invalid, an error is generated.

validateProperties can be invoked multiple times in your application. For example, you can initially set and validate a group of properties, then allow the end user to dynamically select new values and revalidate the settings.

Java validateProperties example

```
// Check to see that properties are valid.
        try
        {
          ab.validateProperties();
         }
         catch (AddressBrokerException abException)
         {
            abException.printStackTrace();
            return null;
         }
         // We have successfully initialized our instance.
         return ab;
```

Step 4: Enter input records and field values

Next, invoke the setField method to specify the input field values. These input field values are the same fields values specified initially when **setProperty** was invoked with the `INPUT_FIELD_LIST` property (see "Java setproperty example code" on page 106). You must call **setField** for each input field value before calling **setRecord**.

An input value need not be set for every field in a record. In the sample code, an individual record that did not contain FirmName information could still be processed.

Invoking setRecord adds the data for the current record to the input record list and advances the record reference.

Java data input example

```
// Build a few records for enhancement...
private static void fillRecords(QMSAddressBroker ab)
   throws IllegalArgumentExceptions, AddressBrokerException
{
   // Fill in a record...
   // An IllegalArgumentException is thrown when setField is invoked
         // with a bad field name (setField("xxx", ...))
         // or a null value (setField(...,null))
         ab.setField("FirmName", "Centrus");
         ab.setField("AddressLine", "4750 Walnut");
         ab.setField("lastLine", "Boulder, CO");
```

```
            // setRecord can throw and AddressBrokerExecption-but only if
            // setField is never invoked.
            ab.setRecord();
            // Fill in the next record...
            ab.setField("FirmName", "White House");
            ab.setField("AddressLine", "1600 Pennsylvania");
            ab.setField("LastLine", Washington, DC);
            ab.setRecord();
}
```

Step 5: Process records

After all the input data is entered, you are ready to process the records. Use the processRecords method to send all the data to the server for processing. In the sample code, GeoStan data files are used to augment address records.

Note: Invoking this method clears the input record buffer, even if it fails.

Java record processing example

```
ab.processRecords();        // Send it to the server...
```

Step 6: Retrieve address records and field values

Invoke getRecord and `getField` to retrieve the output data. The sample code in Java record and field value retrieval example combines this with a system call to display the output. It also shows an example of how to retrieve values from a multi-valued field.

In your Java applications, loop through Steps 4 through 6 of this tutorial each time you process additional records. You can also repeat Steps 2 and 3 to modify property settings.

Java record and field value retrieval example

```
private static void myProcessRecords(QMSAddressBroker ab)
        throws IOException, AddressBrokerException
    {
        ab.processRecords();        // Send it to the server

        // For each record that comes back...
        while (ab.getRecord())
        {
            // Print out the basic address
          System.out.println("Firm=" + ab.getField("firmname"));
          System.out.println("Addr=" + ab.getField("addressline"));
          System.out.println("City=" + ab.getField("city"));
          System.out.println("State=" + ab.getField("state"));
          System.out.println("ZIP=" + ab.getField("ZIP10"));
          System.out.println("MatchCode=" + ab.getField("matchcode"));
          System.out.println("Longitude=" + ab.getField("longitude"));
          System.out.println("Latitude=" + ab.getField("latitude"));
          System.out.println("\n\n");
        }
    }
```

AddressBroker Java methods

The methods described in this chapter are methods of three public classes/interfaces: `QMSAddressBrokerFactory`, `QMSAddressBroker`, and `AddressBrokerException`. Within each class/interface, methods are listed alphabetically. The method syntax in the Java API is:

- is the name of the class.

Some methods are listed as:

```
methodName (overloaded)
```

This indicates there are two or more methods with the same name whose behavior depends on the parameters it is given. For example, the same method accepts either a Boolean type or a string type.

Quick reference

QMSAddressBrokerFactory Class

make

Creates and initializes instances of `QMSAddressBroker` subclasses. Must be invoked before any other method. With the Java API, you cannot directly instantiate a `QMSAddressBroker` instance. Use the `QMSAddressBrokerFactory` helper class to create an instance.

QMSAddressBroker class

Field/data methods

clear

Clears the input and output record buffers and resets all counter properties to zero.

getField (overloaded)

Retrieves the value(s) of an output field in the current output record. Invoke iteratively for fields that contain multiple values.

getFieldAttribute

Retrieves a field attribute, such as its data type and description.

resetField

Resets the output field reference to the first value of an output field.

setField

Sets an input field value in the current input record.

getRecord

Retrieves the record and advances the output record reference.

resetRecord

Resets the output record reference to the first record of the output record buffer.

setRecord

Adds the data for the current record to the input record buffer and advances the input record reference to the next empty record.

Property methods

getProperty (overloaded)

Retrieves the value of an input or output property.

getPropertyAttribute (overloaded)

Retrieves a property attribute, such as its name, data type, and description.

setProperty (overloaded)

Sets the value of a property.

setSocketReadTimeout

Forces the client-side socket to time out after waiting for a server response.

validateProperties

Validates properties for consistency and completeness. This method must be invoked after **`setProperty`** and before invoking **`setField`**.

Processing methods

processRecords

Processes a set of one or more address records.

lookupRecord

Processes a single incomplete address record.

Termination method

close

Closes any active connections to a server.

AddressBrokerException class

Status code method

getStatusCode

Retrieves the status code of a thrown exception.

QMSAddressBrokerFactory class

Use the **`QMSAddressBrokerFactory`** class to create concrete instances of the various subclasses of **`QMSAddressBroker`**. The factory has only one method, **`make`**.

make

Creates instances of QMSAddressBroker subclasses.

Class

```
QMSAddressBrokerFactory
```

Syntax

```
String make
( String in_hostlist,
  String in_transport,
  String in_user,
  String in_password )
throws IllegalArgumentException,
          InstantiationException
```

Arguments

in_hostlist	A delimited list. *Input*.
in_transport	Case-insensitive string that specifies the network protocol AddressBroker uses.
in_user	A valid user name. *Input*.
in_password	A valid user’s password. *Input*.

Return Values

None.

Prerequisites

None.

Alternates

None.

Notes

The client transparently switches between servers if it has a problem establishing communication with its current server. That is, when the client executes a command that includes a server transaction, it switches servers if there is no response from the current server or a transaction fails.

An AddressBroker client uses the first server specified in *`in_hostlist`* until the server fails, at which point it switches to the next server listed in *`in_hostlist`*. The client continues to use this secondary server until it—the secondary server—fails. After a failed server is operational, it again becomes available to the client. However, the client does not switch back unless its current server fails. When a client searches for a server and encounters the end of *`in_hostlist`*, it continues searching from the beginning of the list.

On a per-transaction basis, the client tries each server in turn until it finds an operational server. If it fails to find a server, the operation fails.

When listing multiple servers, it is extremely important that they all service client requests identically. To ensure predictable results, make sure that the server .ini files on each host use the same initialization settings.

There are two valid protocols for the **`make`** method: SOCKET and NOCONNECT. Both SOCKET and NOCONNECT make standard sockets connections to the Address Broker server. However, the SOCKET protocol actually makes a connection to the server and gets a list of properties as set by the Server INI file. The NOCONNECT protocol does not make that connection. NOCONNECT is appropriate for production environments where all processing is defined programmatically, and not by the end user.

An **`InstantiationException`** is thrown when an AddressBroker instance cannot be created.

An `IllegalArgumentException` is thrown when the value in *`in_transport`* is not a supported protocol.

Example 1

```
// Socket protocol using the computer name
ab  = QMSAddressBrokerFactory.make ( “primary:1234 | secondary:1235”,
“socket”, “MyLogon”, “MyPassword” ) ;
```

Example 2

```
// Socket protocol using a URL
ab  = QMSAddressBrokerFactory.make ( “centrus.com:1234 | centrus-
software.com:1235”, “socket”, “MyLogon”, “MyPassword” ) ;
```

Example 3

```
// Socket protocol using an IP address
ab  = QMSAddressBrokerFactory.make ( “204.180.129.200:1234 |
209.38.36.44:1235”, “socket”, “MyLogon”, “MyPassword” ) ;
```

QMSAddressBroker class

The **`QMSAddressBroker`** interface provides all public methods required by the user. It is not possible to make a concrete **`QMSAddressBroker`** instance. Instead, use the **`QMSAddressBrokerFactory`** class to create an instance of **`QMSAddressBroker`**.

clear

Clears input and output record buffers and resets counter properties.

Class

`QMSAddressBroker`

Syntax

```
boolean clear ( )
```

Parameters

None.

Return Values

true if successful, **false** if unsuccessful.

Prerequisites

None.

Alternates

None.

close

Forces any active connection to a server to close.

Class

`QMSAddressBroker`

Syntax

```
void close ( )
```

Parameters

None.

Return Values

None.

Prerequisites

make

Alternates

None.

Notes

The instance is no longer usable after invoking **close**.

Failure to invoke **close** may prevent your process from exiting when expected due to monitor threads persisting beyond the lifetime of your program's other threads.

getField (overloaded)

Retrieves output field values from the current output record.

Class

`QMSAddressBroker`

Syntax

```
String getField (
                                                         String
in_FieldName )
throws IllegalArgumentException,
       AddressBrokerException
String getField (
                                                         String
in_FieldName,
        String in_LogicalName )
throws IllegalArgumentException,
       AddressBrokerException
```

Parameters

in_FieldName A valid, fully specified field name listed in the `OUTPUT_FIELD_LIST` property (see the examples for this function). The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

in_LogicalName — The logical name required by the value of *in_FieldName*. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

Return Values

Single value fields: returns the field value.

Multi-value fields: returns the current value and advances the reference to the next value in the field.

Returns **null** when no values are found.

Prerequisites

```
getRecord
```

Alternates

None.

Notes

The **getField** method retrieves a field value from the current output record. Invoke **getField** iteratively for multi-valued fields. Use the resetField method to reset the field to its first value. To retrieve single value fields more than once, you must invoke **resetField**.

An **IllegalArgumentException** is thrown when:

- *in_FieldName* is **null** or the empty string (“”).
- *in_FieldName* and/or *in_LogicalName* are invalid.
- *in_FieldName* is not in the OUTPUT_FIELD_LIST property.

An **AddressBrokerException** is thrown when no output records are available.

All Spatial+, GDL, and Demographic fields require logical names. GeoStan and GeoStan Canada fields do not.

Example1

```
//Example using a field that does not require a logical name.
String fieldvalue = ab.getField ("CITY");
```

Example 2

```
//Example using a field with its logical name in brackets.
String fieldvalue = ab.getField ("PolygonName[COUNTIES]");
```

Example 3

```
//Example using a field with its logical name as a separate parameter.
String fieldvalue = ab.getField ("PolygonName", "COUNTIES");
```

See Also

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information about fields.

getFieldAttribute

Retrieves a field attribute.

Class

`QMSAddressBroker`

Syntax

```
String getFieldAttribute (
   String in_FieldName,
   int in_FieldIOType,
   int in_AttributeId )
   throws IllegalArgumentException,
       AddressBrokerException
```

Parameters

in_FieldName	A valid field name listed in the `INPUT_FIELD_LIST` or `OUTPUT_FIELD_LIST` property. The property name is not case sensitive, and spaces and underscores are ignored. Do not associate logical names with field names when using this method. *Input*.
in_FieldIOType	A symbolic constant identifying the field name as an input field (**QMSABConst.AB_FIELD_INPUT**) or an output field (**QMSABConst.AB_FIELD_OUTPUT**). *Input*.
in_AttributeId	A symbolic constant identifying the attribute to retrieve. *Input*.

Return Values

Returns the value of the field's attribute. Integer values are returned as strings.

Prerequisites

`setField`

Alternates

None.

Notes

`getFieldAttribute` retrieves a field attribute's value. These are general attributes, not specific to a record. Valid attribute constants below are all public static members of the **`QMSABConst`** class.

Attribute Values

AB_FIELD_DATA_TYPE	"N" (numeric), "C" (character).
AB_FIELD_DECIMALS	Number of decimal places, if numeric.
AB_FIELD_DESCRIPTION	Short (32-character) description of field.
AB_FIELD_HELP	Long (255-character) field description. This is not implemented for all fields.
AB_FIELD_LENGTH	Field width.
AB_FIELD_NEEDS_LOGICAL_NAME	"0" (zero) = No logical name permitted. "G" = A GeoStan logical name required. "S" = A Spatial+ logical name required. "D" = A DemoLib logical name required. "C" = A GeoStan Canada logical name required. "L" = A GDL logical name required.
AB_FIELD_NUM_VALUES	Maximum number of unique values possible for field. An **IllegalArgumentException** is thrown when: *in_FieldName* is **null** or the empty string (""). *in_FieldName* is invalid. *in_FieldIOType* is not in AB_INPUT_FIELD or AB_OUTPUT_FIELD (global Java constants). *in_FieldIOType* contains an invalid value. *in_AttributeId* contains an invalid value. An **AddressBrokerException** is thrown when: validateProperties is not invoked prior to **getFieldAttribute**. There is a communication problem with the server.

Example

```
{
ab.validateProperties();
String fieldattr = ab.getFieldAttribute
("CITY",QMSABConst.AB_FIELD_INPUT, QMSABConst.AB_FIELD_LENGTH );
fieldattr = ab.getFieldAttribute ( "PolygonName",
QMSABConst.AB_FIELD_OUTPUT,QMSABConst.AB_FIELD_DATA_TYPE );
}
```

See Also

See "INPUT_FIELD LIST and OUTPUT_FIELD_LIST" on page 66 for more information about fields.

getProperty (overloaded)

Retrieves a property value.

Class

```
QMSAddressBroker
```

Syntax

```
Object getProperty (
   String in_PropName )
throws IllegalArgumentException
Object getProperty (
   int in_PropId )
throws IllegalArgumentException
```

Parameters

in_PropName	A valid property name. The property name is not case sensitive. Spaces and underscores are ignored. *Input*.
in_PropID	A valid property symbolic constant. *Input*.

Return Values

Returns the property value. The returned value **`Object`** is of type String, Integer, or Boolean, corresponding to the property's data type. Cast the return value to the appropriate type.

Prerequisites

None.

Alternates

None.

Notes

The **`getProperty`** methods retrieve a property value.

An `IllegalArgumentException` is thrown when:

- *`in_PropName`* is **null** or the empty string ("").
- *`in_PropName`* and/or *`in_PropID`* are invalid.

Example

```
Boolean propvalue = (Boolean)ab.getProperty ("MIXED CASE");
```

```
String propvalue = (String)ab.getProperty  (QMSABConst.AB_INIT_LIST);
```

See Also

See Chapter 13 Properties for more information about properties.

getPropertyAttribute (overloaded)

Retrieves a property attribute.

Class

```
QMSAddressBroker
```

Syntax

```
String getPropertyAttribute (
   String in_PropName,
    int in_AttributeId )
throws IllegalArgumentException
String getPropertyAttribute (
   int in_PropID,
    int in_AttributeId )
throws IllegalArgumentException
```

Parameters

in_PropName	A valid property name. The property name is not case sensitive. Spaces and underscores are ignored. *Input*.
in_PropID	A valid property symbolic constant. *Input*.
in_AttributeId	A symbolic constant of the attribute to retrieve. *Input*.

Return Values

Returns the value of the attribute (see the examples for this function).

Prerequisites

`setProperty` if you want client property information.

Alternates

None.

Notes

An **`IllegalArgumentException`** is thrown when:

- *in_PropName* or *in_PropID* is **null** or the empty string (“”).
- *in_PropName* or *in_PropID* is invalid.
- *in_AttributeId* contains an invalid value.

To receive information about properties set on the server, call **make**. To get server property information, call **getPropertyAttribute** before setting any properties in the client code. To receive information about client properties, call **getPropertyAttribute** after calling **setProperties**.

Attribute Values

AB_PROPERTY_DATA_TYPE	“N” (Integer), “B” (Boolean), or “C” (String)
AB_PROPERTY_DEFAULT_VALUE	Default property value
AB_PROPERTY_DESCRIPTION	Short (100-character) description of property
AB_PROPERTY_ID	Property ID
AB_PROPERTY_LENGTH	Length of property value
AB_PROPERTY_NAME	Property name
AB_PROPERTY_READ_ONLY	“1” property is read-only “0” property is read/write

Example 1

```
//Example using the Property Name
String propattr = ab.getPropertyAttribute ("MIXED CASE",
QMSABConst.AB_PROPERTY_DATA_TYPE);
```

Example 2

```
//Example using the Property ID
String propattr = ab.getPropertyAttribute (QMSABConst.AB_INIT_LIST,
QMSABConst.AB_PROPERTY_LENGTH);
```

See Also

See Chapter 13 Properties for more information about properties.

getRecord

Advances the reference to the next record in the output record buffer.

Class

```
QMSAddressBroker
```

Syntax

```
boolean getRecord ( )
```

Parameters

None.

Return Values

true if successful, **false** if unsuccessful.

Prerequisites

`processRecords`

Alternates

None.

Notes

The first time **`getRecord`** is invoked, it sets a reference in the output record buffer to the first output record. Subsequent calls to **`getRecord`** advance the reference. When no further records are found, **false** is returned.

Use the **`getField`** method to retrieve values from individual record fields. Use the resetRecord method to reset the output record reference to the first output record.

Example

```
while ( ab.getRecord() )
{
for (int i = 0; i < fieldnames.length; ++i)
{
String value = ab.getField(fieldnames[i]);
}
}
```

lookupRecord

Processes a single incomplete U.S. address record or performs a reverse lookup on a Canadian postal code.

Class

`QMSAddressBroker`

Syntax

```
int lookupRecord ( )
throws IOException,
       AddressBrokerException
```

Parameters

None.

Return Values

The `OUTPUT_FIELD_LIST` property defines the fields populated by **`lookupRecord`**, and the return codes listed below describe the search outcome. Individual codes are returned only when the relevant fields are included in `OUTPUT_FIELD_LIST`. A return value of zero (**0**) indicates an internal failure.

Return Codes

AB_LOOKUP_ADDRESS_LINE_INCOMPLETE

For a U.S. address, the firm name or unit number could not be resolved. Multiple incomplete records were returned. The user can be prompted to submit more information. The most useful fields for resolving a match generally include `FirmName`, `HighUnitNumber`, `LowUnitNumber`, `MatchCode`, and `UnitType`.

Other helpful fields include `AddressLine`, `AddressLine2`, `CarrierRoute`, `CountyName`, `FIPSCountyCode`, `GovernmentBuildingIndicator`, `HighEndHouseNumber`, `LACSAddress`, `LastLine`, `LowEndHouseNumber`, `PostfixDirection`, `PrefixDirection`, `RoadClassCode`, `SegmentBlockLeft`, `SegmentBlockRight`, `State`, `UrbanizationName`, `USPSRangeRecordType`, `ZIP`, `ZIPCarrtSort`, `ZIPCityDelivery`, `ZIPClass`, `ZIPFacility`, and `ZIPUnique`.

For a Canadian postal code, the input Postal Code is resolved to a range of possible addresses that contain a single street number. The street number suffix or unit number values will vary over the range.

AB_LOOKUP_LAST_LINE_NOT_FOUND

For a U.S. address, multiple incomplete records were returned; the LastLine was not resolved. Iteratively invoke getRecord to retrieve the possible matches. Only the following output fields are returned: MatchCode, CITY, State, ZIP, and ZIPFacility. For a Canadian postal code, this return code indicates that the input postal code was not found in the CPC data and is invalid.

AB_LOOKUP_MULTIPLE_MATCH

For a U.S. address, the address resolved to a multiple match. Multiple complete address records returned. Iteratively invoke getRecord to retrieve possible matches. For a Canadian postal code, the postal code resolved to a range of possible addresses that vary over the street.

AB_LOOKUP_NOT_FOUND

The address could not resolve to a match or possible match. No records returned. Provide a more complete address. (This return code is not used for Canada.)

AB_LOOKUP_SUCCESS

For a U.S. address, a complete single address was matched and returned. For a Canadian postal code, a single address was matched and returned.

AB_LOOKUP_TOO_MANY_CITIES

No records returned. An incomplete LastLine matched over 100 cities. Provide a more complete address. (This return code is not used for Canada.)

Prerequisites

None.

Alternates

`setRecord`

Notes

The `lookupRecord` method processes a single input record and should be used only when address information is insufficient for standardization. To process single or multiple records containing complete addresses, use processRecords.

Minimally, address information for `lookupRecord` must include a street number, a partial street name, and/or valid LastLine information. For Canada, a valid postal code is required and will return a single address or a range of addresses.

`lookupRecord` is most useful in interactive programs, when an application may have to invoke `lookupRecord` iteratively to find a match for an incomplete address. In client/server and Internet environments, the record is transferred across the network with each call to `lookupRecord`. The method does not return until the record is processed. When `lookupRecord` processes an address record and fails to find an exact match, it does an extensive search to find cities and streets that are possible matches.

The `INPUT_FIELD_LIST` property specifies the list of fields passed to `lookupRecord`. Generally, you provide at least `FirmName, AddressLine, and LastLine` fields as input to `lookupRecord`. For Canada, a valid Canadian Postal Code is the only input, and it is set using the PostalCode input field. Only one Postal Code can be processed at a time.

The `OUTPUT_FIELD_LIST` property specifies the list of possible fields returned.

The `MAXIMUM_LOOKUPS` property limits the number of multiples—possible matches—that are returned by `lookupRecord`. The upper limit of `MAXIMUM_LOOKUPS` is 100. For a Canadian postal code, if the `MAXIMUM_LOOKUPS` is set to 100, the Dressmakers software increases the `MAXIMUM_LOOKUPS` to 200.

Retrieve the list of possible matches using a '`while (getRecord) do getField`' loop. No records are returned when the return value of `lookupRecord` is **AB_LOOKUP_NOT_FOUND** or **AB_LOOKUP_TOO_MANY_CITIES**.

Precisely recommends using `processRecords` instead of `lookupRecord`.

An `IOException` is thrown if the client receives a corrupted message, for example, when there is a failure in the network transport layer.

AddressBroker throws an `AddressBrokerException` when:

- Severe problems occur when processing a user request.
- A time-out occurs.
- Logic errors exist.

Example

In an interactive application, a user submits a partial address to `lookupRecord`. The return code is **AB_LOOKUP_LAST_LINE_NOT_FOUND**. For a U.S. address, this code indicates that the user did not enter enough information for `lookupRecord` to resolve the city, state, or ZIP Code. The application prompts the user to select from the list of possible cities and states returned by `lookupRecord`. The user selects the necessary information and resubmits the address to `lookupRecord`. For a Canadian postal code, this return code indicates that the input postal code was not found in the CPC data and is invalid.

This time the return code is **AB_LOOKUP_ADDRESS_LINE_INCOMPLETE**. The user resolved the last line problem, but the return code indicates the address line could be more specific. For a U.S. address, it is missing information on the firm name or unit number (suite, apartment, etc.). The application can prompt the user to select from the list of possibilities returned by this call to `lookupRecord`. The user enters the additional information and resubmits the address to `lookupRecord`, and **AB_LOOKUP_SUCCESS** is returned. For a Canadian postal code, the **AB_LOOKUP_ADDRESS_LINE_INCOMPLETE** code indicates that the input Postal Code resolved to a range of possible addresses that contain a single street number. The street number suffix or unit number values will vary over the range. For example, a Canadian postal code of T3C 2K7 could resolve to 123 A - 123 G Maple Street (when the street suffix varies) or 123 Maple Street Unit 1-100 (when the unit number changes). A valid postal code for one address submitted to `lookupRecord` returns **AB_LOOKUP_SUCCESS**.

When the next address is entered, `lookupRecord` returns the status code **AB_LOOKUP_MULTIPLE_MATCH.** This indicates multiple complete matches were found. For a U.S. address, the user may then be prompted to select from the list of possible matches. The selected address is resubmitted to `lookupRecord` to ensure that it is entirely correct, and that **AB_LOOKUP_SUCCESS** is returned. For a Canadian postal code, the **AB_LOOKUP_MULTIPLE_MATCH** code indicates a postal code that resolved to a range of possible addresses that vary over the street. For example, a Canadian postal code could resolve to 100-120 Elm, Calgary, AB or 150-165 Maple, Calgary, AB.

processRecords

Processes a set of one or more address records.

Class

```
QMSAddressBroker
```

Syntax

```
void processRecords ( )
throws IOException,
       AddressBrokerException
```

Parameters

None.

Return Values

None.

Prerequisites

`setRecord`

Alternates

None.

Notes

Each record should contain enough address information for standardization. For records containing incomplete addresses, use lookupRecord, which progressively returns address choices for one input record at a time.

The method call does not return until all of the records are processed.

An `IOException` is thrown if the client receives a corrupted message; for example, when there is a failure in the network transport layer.

AddressBroker throws an `AddressBrokerException` when:

- severe problems occur when processing a user request.
- a time-out occurs.
- there are logic errors.

See Also

See Chapter 13 Properties for more information about properties.

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information about fields.

resetField

Resets the output field reference to the first value of a multi-valued output field.

Class

`QMSAddressBroker`

Syntax

```
boolean resetField (
    String in_FieldName,
    String in_LogicalName )
throws IllegalArgumentException
```

Parameters

in_FieldName A valid field name listed in the `OUTPUT_FIELD_LIST` property. Some field names require a logical name. The logical name may be appended to *`in_FieldName`* in brackets, or passed in

	the *`in_LogicalName`* parameter (see the examples for this function). The property name is not case sensitive, and spaces and underscores are ignored. *Input*.
in_LogicalName	The logical name required by the value of *`in_FieldName`*. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

Return Values

true if successful, **false** if unsuccessful.

Prerequisites

`getField`

Alternates

None.

Notes

The output field reference is reset to the first value of the output field.

`resetField` returns **false** when *`in_FieldName`* is not found.

An `IllegalArgumentException` is thrown when:

- *`in_FieldName`* is **null** or the empty string ("").
- A logical name is provided in both *`in_FieldName`* and *`in_LogicalName`*.

If **`getField`** is called with the logical name in brackets, **`resetField`** should be called with the logical name in brackets. Similarly, if the logical name is passed as a separate parameter in **`getField`**, then **`resetField`** must also use separate parameters. This is for consistency purposes only; does not cause an error.

All Spatial+, GDL, and Demographic fields require logical names. GeoStan and GeoStan Canada fields do not.

Example 1

```
// Example using field name with its logical name in brackets.
while ( ab.getField ( "polygonName[COUNTIES]" ) )
{
...
}
ab.resetField ("PolygonName","PolygonName[Counties]");
```

Example 2

```
// Example using field name with its logical name as separate
parameter.
while ( ab.getField ( "polygonName", "COUNTIES" ) )
{
...
}
ab.resetField ("PolygonName", "Counties");
```

See Also

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information about fields.

resetRecord

Resets output record reference to the first record in the output record buffer.

Class

```
QMSAddressBroker
```

Syntax

```
boolean resetRecord ( )
```

Parameters

None.

Return Values

true if successful, **false** if unsuccessful.

Prerequisites

```
getField
```

Alternates

None.

setField

Sets an input field value in the current input record.

Class

```
QMSAddressBroker
```

Syntax

```
void setField (
   String in_FieldName,
   String in_FieldValue )
throws IllegalArgumentException,
       AddressBrokerException
```

Parameters

in_FieldName	A valid field name listed in the `INPUT_FIELD_LIST` property. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

in_FieldValue	The string value to assign to the field. Maximum string length is determined by the AB_FIELD_LENGTH field attribute. *Input*.

Return Values

None.

Prerequisites

`setProperty`

Alternates

None.

Notes

The `RECORD_DELIMITER`, `FIELD_DELIMITER`, and `VALUE_DELIMITER` properties have default values of line feed, tab, and CTRL-A, respectively. If your data contains any of these characters, you *must* reset the appropriate property to a different character. In addition, your data may not contain the NULL character.

An **`IllegalArgumentException`** is thrown when:

- *`in_FieldName`* is **null** or the empty string (“”).
- *`in_FieldName`* is invalid.
- *`in_FieldName`* is not in the `INPUT_FIELD_LIST` property.
- The length of *`in_FieldValue`* is > 256 characters.

An **`AddressBrokerException`** is thrown when:

- *`in_FieldValue`* is **null**.
- Properties were set (via **`setProperty`**) but were not validated (via validateProperties).

Example

```
ab.setField (“AddressLine”, “123 Main”);
ab.setField (“LastLine”, “Anytown, NY”);
```

See Also

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information about fields.

setProperty (overloaded)

Assigns a property value.

Class

```
QMSAddressBroker
```

Syntax

```
void setProperty (
   String in_PropName,
   Boolean in_bPropValue )
throws IllegalArgumentException
void setProperty (
   String in_PropName,
   boolean in_bPropValue )
throws IllegalArgumentException
void setProperty (
   String in_PropName,
   String in_sPropValue )
throws IllegalArgumentException
void setProperty (
   String in_PropName,
   Integer in_iPropValue )
throws IllegalArgumentException
void setProperty (
   String in_PropName,
   int in_iPropValue )
throws IllegalArgumentException
void setProperty (
   int in_PropID,
   Boolean in_bPropValue )
throws IllegalArgumentException
void setProperty (
   int in_PropID,
   boolean in_bPropValue )
throws IllegalArgumentException
void setProperty (
   int in_PropID,
   String in_sPropValue )
throws IllegalArgumentException
void setProperty (
   int in_PropID,
   Integer in_iPropValue )
throws IllegalArgumentException
void setProperty (
   int in_PropID,
   int in_iPropValue )
throws IllegalArgumentException
```

Parameters

in_PropName	A valid property name. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

in_PropID	The valid symbolic constant of the property being set. *Input*.
in_bPropValue	A Boolean object or Boolean value to assign to the property. *Input*.
in_sPropValue	A string value to assign to the property. *Input*.
in_iPropValue	An integer object or integer value to assign to the property. *Input*.

Return Values

None.

Prerequisites

`QMSAddressBrokerFactory.make`

Alternates

None.

Notes

The specific `setProperty` method to use depends on the data type of the property you are setting.

An `IllegalArgumentException` exception is thrown when:

- *`in_PropName`* or *`in_PropID`* are **null** or invalid.
- The property value is **null**.
- The data type of the property does not correspond to the data type of the value.

Example

```
ab.setProperty ("MIXED CASE", true);
ab.setProperty (QMSABConst.AB_INIT_LIST, "GEOSTAN |COUNTIES");
```

See Also

See Chapter 13 Properties for more information about properties.

setRecord

Adds data for the current record to the input record buffer and advances the input record reference to the next empty record in the buffer.

Class

```
QMSAddressBroker
```

Syntax

```
void setRecord ( )
```

Parameters

None.

Return Values

None.

Prerequisites

setField

Alternates

None.

setSocketReadTimeout

Forces the client-side socket to time out after waiting for a server response.

Class

`QMSAddressBroker`

Syntax

```
void setSocketReadTimeout (int seconds)
```

Parameters

int seconds	The number of seconds spent waiting for a server response before a timeout occurs.

Return Values

None.

Prerequisites

None.

Alternates

None.

Notes

The `setSocketReadTimeout` method controls the timeout for establishing the initial socket connection to the server. It forces the client-side socket to time out after a specific number of seconds spent waiting for a server response. If the socket read times out, a failure message is sent to the application.

The application should check the success of the `processRecords` call to verify a good status was returned. The Java client API also throws an `AddressBrokerException` when a problem is discovered. If the application does not set the socket read timeout, or if it makes the call and passes a zero as the parameter, the program continues to wait for a response from the server.

Example

```
broker.setSocketReadTimeout( 5 );
```

validateProperties

Validates properties for consistency and completeness.

Class

```
QMSAddressBroker
```

Syntax

```
void validateProperties ( )
throws AddressBrokerException
```

Parameters

None.

Return Values

true if successful, **false** if unsuccessful.

Prerequisites

`setProperty`

Alternates

None.

Notes

The **`validateProperties`** method verifies the values of initialization and processing control properties to ensure a complete and compatible set of values are available to AddressBroker. Call this method after one or more properties have been set and before calling setField or any processing methods.

When **`validateProperties`** returns **`true`**, it indicates all properties have been successfully validated and that AddressBroker is ready to process records. In some cases, all properties can be validated in a single method call.

See Also

See Chapter 13 Properties for more information about properties.

AddressBrokerException class

AddressBroker methods throw an object of this class to indicate run-time, logical, or processing errors.

getStatusCode

Retrieves the Status Code from a thrown exception.

Class

```
AddressBrokerException
```

Syntax

```
long getStatusCode ( )
```

Parameters

None.

Return Values

Returns the 10-digit integer status code.

Prerequisites

None.

Alternates

None.

Notes

`getStatusCode` is available only while an exception object is accessible (in scope).

See Also

See "GeoStan location codes" on page 433 for a description of status codes. See "AddressBroker Java exceptions" on page 139 for more information on this exception class.

AddressBroker Java exceptions

In the Java API, many AddressBroker methods have no return codes as compared to the C and C++ APIs. Instead, your application must use exception handling. Exceptions are listed in the method syntax statements.

The AddressBroker Java API throws four classes of exceptions:

- `qms.addressbroker.client.AddressBrokerException` – a general run time exception.
- `java.lang.InstantiationException` – instantiation failure.
- `java.lang.IllegalArgumentException` – a parameter to a method is improper.
- `java.io.IOException` – the output stream from a request to the server was corrupted.

AddressBrokerException class

An object of this class is thrown by the methods of the `QMSAddressBroker` class to indicate a run-time, logical, or processing error. This exception class extends the `java.lang.RuntimeException` by adding a status code and message. AddressBrokerException handling example shows an `AddressBrokerException` `try` block example. See "getStatusCode" on page 138 for information about the `getStatusCode` method.

AddressBrokerException handling example

```
...
    try {
         myAddressBrokerInstance.getField("NONSENSE NAME");
    catch( AddressBrokerException abException ) {
         // Unknown field name error
         System.out.println("An exception occurred:\n" + abException);
       System.out.println("ErrorCode = " + abException.getStatusCode());
    }
...
```

IllegalArgumentException class

Parameters passed to methods are checked for correctness. IllegalArgumentException handling example shows an example that checks for an `IllegalArgumentException`.

IllegalArgumentException handling example

```
...
     try {
         myAddressBrokerInstance.getField(null);
     catch( IllegalArgumentException illArgExcept ) {
         // Unknown field name error
         System.out.println(illArgExcept);
    }
...
```

IOException class

AddressBroker throws an exception of this class when the output stream received from a processRecords or a lookupRecord call is corrupted.

9 – .NET API

In this chapter

This chapter describes the .NET API to AddressBroker in detail.

This chapter includes a tutorial using the AddressBroker .NET API. The tutorial shows you how to use most of AddressBroker's functionality, yet is general enough that you can modify it for other uses. A complete method reference follows the tutorial. The final section of this chapter discusses error handling.

The naming convention for AddressBroker .NET API methods is **`MethodName`**.

Accessing the AddressBroker .NET library

To use the AddressBroker .NET API, you must have Microsoft .NET Framework installed on your machine. The .NET Framework is part of the Microsoft Visual Studio .NET installation, or you can download the Microsoft .NET Framework Software Development Kit from http://msdn.microsoft.com.

Note: .NET AddressBroker clients must use version 2.0 or higher of the .NET Framework.

AddressBroker .NET tutorial

This section describes the steps necessary to develop a .NET client application using the AddressBroker .NET API. The example shows basic .NET sample code that performs address record enhancement. It uses the firm name and address fields from the address records as input. This example standardizes the address data and augments it with city, state, and 9-digit ZIP Code information from the GeoStan Precisely Enhanced data directory.

Step 1: Create and initialize the client object

Use the C# `using` statement for the Centrus®. AddressBroker namespace to allow class references without having to specify the fully qualified class name. The AddressBrokerFactory helper class creates an instance of ABClient for you.

.NET initialization example

```
[C#]
using Centrus.AddressBroker;
ABClient ab = null;
try
{
ab = AddressBrokerFactory.Make(AddressBrokerServer + ":" +
AddressBrokerPort, "SOCKET");
}
catch (AddressBrokerException abe)
{
Console.WriteLine("AddressBrokerFactory.Make exception: " +
abe.Message);
}
```

```
catch (ArgumentOutOfRangeException rangeArg)
{
Console.WriteLine("AddressBrokerFactory.Make out of range exception: " +
rangeArg.Message);
}
catch (ArgumentNullException nullArg)
{
Console.WriteLine("AddressBrokerFactory.Make null argument exception: "
+ nullArg.Message);
}
catch (Exception e)
{
Console.WriteLine("AddressBrokerFactory.Make exception (type " +
e.ToString() + "): " + e.Message);
}
```

[Visual Basic]

```
Dim ab As New AddressBrokerFactory()
Dim abclient As New ABClient()

'create the AB Client and connect to an AddressBroker Server
Try
abclient = ab.Make(txtServer.Text & ":" & txtPort.Text, "SOCKET", "", "")
Catch abe As AddressBrokerException
MsgBox(abe.Message, MsgBoxStyle.Critical, "AB AddressBrokerException")
Exit Sub
Catch nullArg As ArgumentNullException
MsgBox(nullArg.Message, MsgBoxStyle.Critical, "AB
ArgumentNullException")
Exit Sub
Catch rangeArg As ArgumentOutOfRangeException
MsgBox(rangeArg.Message, MsgBoxStyle.Critical, "AB
ArgumentOutOfRangeException")
Exit Sub
Catch er1 As Exception
MsgBox(er1.Message, MsgBoxStyle.Critical, "AB Exception: " +
er1.ToString())
Exit Sub
End Try
```

Production code example

```
using System;
using System.IO;
using Centrus.AddressBroker;  //Use this to import the Address Broker
classes

namespace AddressBrokerCSharpConsoleExample
{
    class SimpleConsole
    {
        [STAThread]
        static void Main(string[] args)
        {
            string AddressBrokerServer;        // default is localhost
            string AddressBrokerPort;        // default is 4660

            // Specify the machine name where the server is running
            // (list should be host:port|host:port)
```

```
            Console.Write("Address Broker Server (default: localhost): ");
              AddressBrokerServer = Console.ReadLine();
              if (AddressBrokerServer.Equals(string.Empty) == true)
              {
                  AddressBrokerServer = "localhost";
              }

              Console.Write("Address Broker Port (default: 4660): ");
              AddressBrokerPort = Console.ReadLine();
              if (AddressBrokerPort.Equals(string.Empty) == true)
              {
                  AddressBrokerPort = "4660";
              }

              /// Step #1

              ABClient ab = null;
              Try
               // Using NOCONNECT for production
              {
                 ab = AddressBrokerFactory.Make(AddressBrokerServer + ":"
+ AddressBrokerPort, "NOCONNECT");
              }
              catch (AddressBrokerException abe)
              {
                  Console.WriteLine("AddressBrokerFactory.Make exception:
" + abe.Message);
              }
              catch (ArgumentOutOfRangeException rangeArg)
              {
                Console.WriteLine("AddressBrokerFactory.Make out of range
exception: " + rangeArg.Message);
              }
              catch (ArgumentNullException nullArg)
              {
               Console.WriteLine("AddressBrokerFactory.Make null argument
exception: " + nullArg.Message);
              }
              catch (Exception e)
              {
                  Console.WriteLine("AddressBrokerFactory.Make exception
(type " + e.ToString() + "): " + e.Message);
              }
            /// End Step #1


              /// Step #2

              // Set client side properties
             // These properties are typically a subset of the properties
listed on
           // the server.  If no properties are specified, the application
can
          // access any of the properties specified in the server.ini file.
              try
              {
                  // Tell AddressBroker what logical Names we are using.
                  // For this example, we are doing only address
standardization and
                  // geocoding so only GeoStan properties are used.
```

```
                ab.SetProperty("INIT_LIST", "Geostan|Geostan_Z9");
                // Here we tell AddressBroker the input record format.
Although we
                // do this only once in the example, it is
            // a dynamic property so you could set it at any time, as many
                // times as you want.
                ab.SetProperty("INPUTFIELDLIST",
"firmname|addressline|lastline");
              // This is list of the output fields listed in the output
record.
                ab.SetProperty("OUTPUTFIELDLIST",

"firmname|addressline|city|state|zip10|match_code|location_quality_code
|longitude|latitude");
                // Set properties that affect the behavior of the server
             // These properties will override behavior specified in the
                // server.ini file
                // Set the input mode
                ab.SetProperty("Input_Mode", 0);
              // Only want single output record for each input record...
                ab.SetProperty("Keep_multimatch", false);
                // 200 foot buffer instead of the default of 50
                ab.SetProperty("BUFFER RADIUS", 200);
                //
                Console.WriteLine("Keep Multimatch is: " +
ab.GetProperty(262));
            }
            catch (AddressBrokerException abe)
            {
                // Attempt to set a non-existent property
                // Data type mismatch (E.g. set a string property to
                // an Integer value)
               Console.WriteLine("Set Property failed: " + abe.Message);
            }
            catch (ArgumentOutOfRangeException rangeArg)
            {
                // A property was set to an invalid value or
                // the property name/id was incorrect.
                Console.WriteLine("Argument out of range: " +
rangeArg.Message);
            }
            catch (ArgumentNullException nullArg)
            {
                // A Parameter value was null
              Console.WriteLine("Null argument: " + nullArg.StackTrace);
            }
            catch (Exception e)
            {
              Console.WriteLine("Exception (type " + e.ToString() + "):
" + e.Message);
            }
          /// End Step #2

          /// Step #4

          // Establish the input variables
          string firmname;                   // no default -- required
          string address;                    // no default -- required
          string lastline;                   // no default -- required
```

```
            string longitude;                   // no default -- required
            string latitude;                    // no default -- required
            string selection;             // menu selection

            bool ContinueProcessing = true;
            bool ProcessRecs = true;
            bool ReverseGeocode = false;

            // Simple menu for user to choose either predefined samples
addresses or
            // enter a single address.
            while(ContinueProcessing == true)
            {
                  ProcessRecs = true;
                  Console.WriteLine(" ");
                  Console.WriteLine("Please make a selection (1 or 2, q to
quit program): ");
                  Console.WriteLine("1 - Samples ");
                  Console.WriteLine("2 - Enter an address ");
                  Console.WriteLine("3 - Enter a lon/lat ");
                  Console.WriteLine("q - Quit ");
                  Console.WriteLine(" ");
                  Console.Write("Make selection: ");
                  selection = Console.ReadLine();

                  if (selection == null)
                  {
                        return;
                  }
                  switch(selection)
                  {
                        case "1":
                              /* Sample addresses filled in for
standardization
                                    - An IllegalArgumentException is
thrown when SetField is invoked
                                      with a bad field name
(SetField("xxx", ...))
                                   or a null value (SetField(...,null))
*/
                              try
                              {     // Set data.
                                    ab.SetField("firmname", "Group1
Software");
                                    ab.SetField("addressLine", "4750
Walnut");
                                    ab.SetField("lastline", "Boulder,
CO");

                                    /* SetRecord can throw and
AddressBrokerExecption-but only if
                                       SetField is never invoked. */
                                    ab.SetRecord();

                                    // Fill in the next record...
                                    ab.SetField("firmname", "White
House");
                                    ab.SetField("addressline", "1600
Pennsylvania");
```

```
                                        ab.SetField("lastline",
"Washington, DC");
                                        ab.SetRecord();

                                }
                                catch (AddressBrokerException abe)
                                {
                                        Console.WriteLine("SetField or
SetRecord exception: " + abe.Message);
                                }
                                catch (ArgumentOutOfRangeException
rangeArg)
                                {
                                        // If input value is too long, field
is invalid, or field is readonly.
                                        Console.WriteLine("Argument out of
range: " + rangeArg.Message);
                                }
                                catch (ArgumentNullException nullArg)
                                {
                                        /* Attempt to set a field to null or
the field name parameter
                                           was null. */
                                        Console.WriteLine("Null argument: "
+ nullArg.StackTrace);
                                }
                                catch (Exception e)
                                {
                                        Console.WriteLine("Exception (type
" + e.ToString() + "): " + e.Message);
                                }
                                break;

                        case "2":
                                // Allows user to enter an address for
standardization
                                // Get an address from the command line
                                Console.WriteLine(" ");
                                Console.Write("Firm: ");
                                firmname = Console.ReadLine();

                                Console.Write("Address: ");
                                address = Console.ReadLine();

                                Console.Write("LastLine: ");
                                lastline = Console.ReadLine();

                                try
                                {       // Set data from input lines.
                                        ab.SetField("FIRMNAME", firmname);
                                        ab.SetField("ADDRESSLINE",
address);
                                        ab.SetField("LASTLINE", lastline);
                                        ab.SetRecord();
                                }
                                catch (Exception e)
                                {
                                        Console.WriteLine("SetField
Exception (type " + e.ToString() + "): " + e.Message);
                                }
```

```
                        Console.WriteLine(" ");

                        break;

Exception (type " + e.ToString() + "): " + e.Message);
                        }

                        Console.WriteLine(" ");

                        break;
                    // User enters Q or q to quit the program.
                    case "Q":
                    case "q":
                        return;


                    default :
                        ProcessRecs = false;
                        break;

                } // end of switch (selection)

                // End Step #4


                // Step #5
                // Process Records
                if(ProcessRecs)
                {
                    try
                    {
                        ab.ProcessRecords();
                    }
                    catch (AddressBrokerException abe)
                    {
                        Console.WriteLine("ProcessRecords
exception: " + abe.Message);
                    }
                    catch (IOException ioe)
                    {
                        Console.WriteLine("ProcessRecord
communication exception: " + ioe.Message);
                    }
                    catch (Exception e)
                    {
                        Console.WriteLine("Exception (type " +
e.ToString() + "): " + e.Message);
                    }

                    Console.WriteLine(" ");

                    /// End Step #5
            }

                /// Step #6
                try
                {
                    // For each record that comes back...
                    while (ab.GetRecord() == true)
```

```
                {
                        if (!ReverseGeocode)
                        {
                                // Print out the basic address
information
                                Console.WriteLine(" ");
                                Console.WriteLine("****** RESULTS
****** ");
                                Console.WriteLine(" ");
                                Console.WriteLine("Firm = " +
ab.GetField("firmname"));
                                Console.WriteLine("Address = " +
ab.GetField("addressline"));
                                Console.WriteLine("City = " +
ab.GetField("city"));
                                Console.WriteLine("State = " +
ab.GetField("state"));
                                Console.WriteLine("ZIP = " +
ab.GetField("ZIP10"));
                                Console.WriteLine("MatchCode = " +
ab.GetField("matchcode"));
                                Console.WriteLine("Location Quality
Code = " + ab.GetField("location_quality_code"));
                                Console.WriteLine("Longitude = " +
ab.GetField("longitude"));
                                Console.WriteLine("Latitude = " +
ab.GetField("latitude"));
                                Console.WriteLine(" ");
                        }
                        else
                        {
                                // Print out the basic address
information
                                Console.WriteLine(" ");
                                Console.WriteLine("****** REVERSE
GEOCODING RESULTS ****** ");
                                Console.WriteLine(" ");
                                Console.WriteLine("Address = " +
ab.GetField("addressline"));
                                Console.WriteLine("City = " +
ab.GetField("city"));
                                Console.WriteLine("State = " +
ab.GetField("state"));
                                Console.WriteLine("ZIP = " +
ab.GetField("ZIP10"));
                                Console.WriteLine("MatchCode = " +
ab.GetField("matchcode"));
                                Console.WriteLine("Location Quality
Code = " + ab.GetField("location_quality_code"));
//                              Console.WriteLine("Longitude = " +
ab.GetField("longitude"));
//                              Console.WriteLine("Latitude = " +
ab.GetField("latitude"));
                                Console.WriteLine(" ");
                        }
                }
        }
        catch (AddressBrokerException abe)
        {
```

```
                        Console.WriteLine("GetRecords/GetField
exception: " + abe.Message);
                }
                catch (ArgumentOutOfRangeException rangeArg)
                {
                        // Input field is invalid
                        Console.WriteLine("Argument out of range: " +
rangeArg.Message);
                }
                catch (ArgumentNullException nullArg)
                {
                        // Input field is null.
                        Console.WriteLine("Null argument: " +
nullArg.StackTrace);
                }
                catch (Exception e)
                {
                        Console.WriteLine("GetRecords/GetField exception
(type " + e.ToString() + "): " + e.Message);
                }
                /// End Step #6
            } // end while(ContinueProcessing == true)

            /// Step #7

            ab.Close();

            /// End Step #7

        }
    }
}
```

Step 2: Set properties

The client application should set the following properties using the `SetProperty` method:

- INIT_LIST—The list of logical names the application uses.

  Logical name and paths are set on the server. The logical names the client uses must match those set on the server. The logical names the client application uses must be defined in the server INI file. See “LogicalNames” on page 329 for more information about logical names.

  In the example code shown in “.NET SetProperty example code” on page 152 the logical names GEOSTAN and GEOSTAN_Z9 refer to a GeoStan data directory and a GeoStan ZIP Code data file, respectively.

- INPUT_FIELD_LIST—The delimited list of field names. The allowable field names in the INPUT_FIELD_LIST are determined by your input data format and the INPUT_MODE property. See “Defining the INPUT_FIELD_LIST” on page 67 for more information about the INPUT_FIELD_LIST.

  Note: The INPUT_FIELD_LIST defined in the client application overrides any settings in the server INI file.

In the sample code, AddressBroker uses the `FirmName`, `AddressLine`, and `LastLine` field values from each input record.

- `OUTPUT_FIELD_LIST`—The delimited list of field names to retrieve from the output records. Spatial+, GDL, and Demographics outputs require a logical name paired with the output field name. See "Defining the OUTPUT_FIELD_LIST" on page 67 for more information about the `OUTPUT_FIELD_LIST`.

  Note: The `OUTPUT_FIELD_LIST` defined in the client application overrides any settings in the server INI file.

  The sample shows how to enhance the address record with city, state, and ZIP10 information from the GeoStan data file.

You may set other properties in the client. In the example code, `KEEP_MULTIMATCH` and `BUFFER_RADIUS` are set. See Chapter 13 Properties for a detailed discussion about other properties.

.NET property reference syntax

```
[C#]
// Set client side properties.
ab.SetProperty("INIT_LIST", "Geostan|Geostan_Z9");
// Here we tell AddressBroker the input record format. Although we
// do this only once in the example, it is
// a dynamic property so you could set it at any time, as many
// times as you want.
ab.SetProperty("INPUTFIELDLIST", "firmname|addressline|lastline");
// This is list of the output fields listed in the output record.
ab.SetProperty("OUTPUTFIELDLIST",
"firmname|addressline|city|state|zip10|match_code|location_quality_code
|longitude|latitude");
// Set properties that affect the behavior of the server
// These properties will override behavior specified in the
// server INI file
// Set the input mode
ab.SetProperty("Input_Mode", 0);
// Only want single output record for each input record...
ab.SetProperty("Keep_multimatch", false);
// 200 foot buffer instead of the default of 50
ab.SetProperty("BUFFER RADIUS", 200);
```

```
[Visual Basic]
Try
'Tell AddressBroker what logical Names we are using.
'For this example, we are doing only address standardization and
'geocoding so only GeoStan properties are used.
abclient.SetProperty("INIT_LIST", "Geostan|Geostan_Z9")

'Here we tell AddressBroker the input record format. Although we
'do this only once in the example, it is
'a dynamic property so you could set it at any time, as many
'times as you want.
abclient.SetProperty("INPUTFIELDLIST", "firmname|addressline|lastline")

'This is list of the output fields listed in the output record.
```

```
abclient.SetProperty("OUTPUTFIELDLIST",
"firmname|addressline|lastline|match_code|locationqualitycode|longitude
|latitude")

'Set properties that affect the behavior of the server
'These properties will override behavior specified in the
'server INI file
'Set the input mode
abclient.SetProperty("Input_Mode", "inputnormal")
'Only want single output record for each input record...
abclient.SetProperty("Keep_multimatch", "TRUE")
'200 foot buffer instead of the default of 50
abclient.SetProperty("BUFFER RADIUS", 200)
```

.NET SetProperty example code

```
[C#]
// Set client side properties. These properties are typically a subset of
the
// properties listed on the server.  If no properties are specified, the
// application can access any of the properties specified in the server INI
file.
try
{
// Tell AddressBroker what logical Names we are using.
// For this example, we are doing only address standardization and
// geocoding so only GeoStan properties are used.

ab.SetProperty("INIT_LIST", "Geostan|Geostan_Z9");
// Here we tell AddressBroker the input record format. Although we
// do this only once in the example, it is
// a dynamic property so you could set it at any time, as many
// times as you want.
ab.SetProperty("INPUTFIELDLIST", "firmname|addressline|lastline");
// This is list of the output fields listed in the output record.
ab.SetProperty("OUTPUTFIELDLIST",
"firmname|addressline|city|state|zip10|match_code|location_quality_code
|longitude|latitude");
// Set properties that affect the behavior of the server
// These properties will override behavior specified in the
// server INI file
// Set the input mode
ab.SetProperty("Input_Mode", 0);
// Only want single output record for each input record...
ab.SetProperty("Keep_multimatch", false);
// 200 foot buffer instead of the default of 50
ab.SetProperty("BUFFER RADIUS", 200);
}
catch (AddressBrokerException abe)
{
// Attempt to set a non-existent property
// Data type mismatch (E.g. set a string property to
// an Integer value)
Console.WriteLine("Set Property failed: " + abe.Message);
}
catch (ArgumentOutOfRangeException rangeArg)
{
// A property was set to an invalid value or
// the property name/id was incorrect.
Console.WriteLine("Argument out of range: " + rangeArg.Message);
```

```
}
catch (ArgumentNullException nullArg)
{
// A Parameter value was null
Console.WriteLine("Null argument: " + nullArg.StackTrace);
}
catch (Exception e)
{
Console.WriteLine("Exception (type " + e.ToString() + "): " + e.Message);
}
```

[Visual Basic]

```
Try
'Tell AddressBroker what logical Names we are using.
'For this example, we are doing only address standardization and
'geocoding so only GeoStan properties are used.
abclient.SetProperty("INIT_LIST", "Geostan|Geostan_Z9")

'Here we tell AddressBroker the input record format. Although we
'do this only once in the example, it is
'a dynamic property so you could set it at any time, as many
'times as you want.
abclient.SetProperty("INPUTFIELDLIST", "firmname|addressline|lastline")

'This is list of the output fields listed in the output record.
abclient.SetProperty("OUTPUTFIELDLIST",
"firmname|addressline|lastline|match_code|locationqualitycode|longitude
|latitude")

'Set properties that affect the behavior of the server
'These properties will override behavior specified in the
'server INI file
'Set the input mode
abclient.SetProperty("Input_Mode", "inputnormal")
'Only want single output record for each input record...
abclient.SetProperty("Keep_multimatch", "TRUE")
'200 foot buffer instead of the default of 50
abclient.SetProperty("BUFFER RADIUS", 200)

Catch abe As AddressBrokerException
MsgBox(abe.Message, MsgBoxStyle.Critical, "AB AddressBrokerException")
Exit Sub
Catch nullArg As ArgumentNullException
MsgBox(nullArg.Message, MsgBoxStyle.Critical, "AB
ArgumentNullException")
Exit Sub
Catch rangeArg As ArgumentOutOfRangeException
MsgBox(rangeArg.Message, MsgBoxStyle.Critical, "AB
ArgumentOutOfRangeException")
Exit Sub
Catch er1 As Exception
MsgBox(er1.Message, MsgBoxStyle.Critical, "AB Exception: " &
er1.ToString())
Exit Sub
End Try
```

Step 3: Validate properties (optional)

Use the ValidateProperties method to send the property definitions to the server for validation. When `ValidateProperties` returns **true**, the AddressBroker client object properties are set correctly and are ready for processing. If any property setting is invalid, an error is generated.

`ValidateProperties` can be invoked multiple times in your application. For example, you can initially set and validate a group of properties, then allow the end user to dynamically select new values and revalidate the settings.

.NET ValidateProperties example

```
[C#]
// Check to see that properties are valid.
try
{
ab.ValidateProperties();
}
catch (AddressBrokerException abException)
{
Console.WriteLine("Validate Properties exception: " +
abException.Message);
}
catch (Exception e)
{
Console.WriteLine("Validate Properties exception (type " + e.ToString()
+ "): " + e.Message);
}
```

```
[Visual Basic]
'Check to see that properties are valid.
Try
abclient.ValidateProperties()
Catch abe As AddressBrokerException
MsgBox(abe.Message, MsgBoxStyle.Critical, "AB AddressBrokerException")
Exit Sub
Catch er1 As Exception
MsgBox(er1.Message, MsgBoxStyle.Critical, "AB Exception: " &
er1.ToString())
Exit Sub
End Try
```

Step 4: Enter input records and field values

Next, invoke the SetField method to specify the input field values. These input field values are the same fields values specified initially when `setProperty` was invoked with the `INPUT_FIELD_LIST` property (see “.NET SetProperty example code” on page 152). You must call `SetField` for each input field value before calling `SetRecord`.

An input value need not be set for every field in a record. In the sample code, an individual record that did not contain FirmName information could still be processed.

Invoking SetRecord adds the data for the current record to the input record list and advances the record pointer.

.NET data input example

[C#]

```
try
{
// Build a few records for enhancement...
// Fill in a record...
// An IllegalArgumentException is thrown when SetField is invoked
// with a bad field name (SetField("xxx", ...))
// or a null value (SetField(...,null))
ab.SetField("firmname", "Centrus");
ab.SetField("AddressLine", "4750 Walnut");
ab.SetField("lastLine", "Boulder, CO");
// SetRecord can throw an AddressBrokerExecption - but only if
// SetField is never invoked.
ab.SetRecord();
// Fill in the next record...
ab.SetField("firmname", "White House");
ab.SetField("AddressLine", "1600 Pennsylvania");
ab.SetField("LastLine", "Washington, DC");
ab.SetRecord();
}
catch (AddressBrokerException abe)
{
Console.WriteLine("SetField or SetRecord exception: " + abe.Message);
}
catch (ArgumentOutOfRangeException rangeArg)
{
// Input value is too long, field is invalid, or field is readonly.
Console.WriteLine("Argument out of range: " + rangeArg.Message);
}
catch (ArgumentNullException nullArg)
{
// Attempt to set a field to null or the field name parameter
// was null.
Console.WriteLine("Null argument: " + nullArg.StackTrace);
}
catch (Exception e)
{
Console.WriteLine("Exception (type " + e.ToString() + "): " + e.Message);
}
```

[Visual Basic]

```
'Set input fields -- Submit values for the form
Try
abclient.SetField("firmname", txtFirm.Text)
abclient.SetField("addressline", txtAddress.Text)
abclient.SetField("lastline", txtLastline.Text)
'Set input record
abclient.SetRecord()
Catch abe As AddressBrokerException
```

```
MsgBox(abe.Message, MsgBoxStyle.Critical, "AB AddressBrokerException")
Exit Sub
Catch nullArg As ArgumentNullException
MsgBox(nullArg.Message, MsgBoxStyle.Critical, "AB
ArgumentNullException")
Exit Sub
Catch rangeArg As ArgumentOutOfRangeException
MsgBox(rangeArg.Message, MsgBoxStyle.Critical, "AB
ArgumentOutOfRangeException")
Exit Sub
Catch er1 As Exception
MsgBox(er1.Message, MsgBoxStyle.Critical, "AB Exception: " &
er1.ToString())
Exit Sub
End Try
```

Step 5: Process records

After all the input data is entered, you are ready to process the records. Use the ProcessRecords method to send all the data to the server for processing. In the sample code, GeoStan data files are used to augment address records.

Note: Invoking this method clears the input record buffer, even if it fails.

.NET record processing example

[C#]

```
try
{
ab.ProcessRecords();
}
catch (AddressBrokerException abe)
{
Console.WriteLine("ProcessRecords exception: " + abe.Message);
}
catch (IOException ioe)
{
Console.WriteLine("ProcessRecord communication exception: " +
ioe.Message);
}
catch (Exception e)
{
Console.WriteLine("Exception (type " + e.ToString() + "): " + e.Message);
}
```

[Visual Basic]

```
'Process the record
Try
abclient.ProcessRecords()
Catch ioe As IOException
MsgBox(ioe.Message, MsgBoxStyle.Critical, "AB IOException")
Exit Sub
Catch rangeArg As ArgumentOutOfRangeException
MsgBox(rangeArg.Message, MsgBoxStyle.Critical, "AB
ArgumentOutOfRangeException")
Exit Sub
```

```
Catch er1 As Exception
MsgBox(er1.Message, MsgBoxStyle.Critical, "AB Exception: " &
er1.ToString())
Exit Sub
End Try
```

Step 6: Retrieve address records and field values

Invoke GetRecord and `GetField` to retrieve the output data. The sample code in .NET record and field value retrieval example combines this with a system call to display the output. It also shows an example of how to retrieve values from a multi-valued field.

In your .NET applications, loop through Steps 4 through 6 of this tutorial each time you process additional records. You can also repeat Steps 2 and 3 to modify property settings.

.NET record and field value retrieval example

```
[C#]
try
{
// For each record that comes back...
while (ab.GetRecord() == true)
{
   // Print out the basic address
   Console.WriteLine("Firm = " + ab.GetField("firmname"));
   Console.WriteLine("Addr = " + ab.GetField("addressline"));
   Console.WriteLine("City = " + ab.GetField("city"));
   Console.WriteLine("State = " + ab.GetField("state"));
   Console.WriteLine("ZIP = " + ab.GetField("ZIP10"));
   Console.WriteLine("MatchCode = " + ab.GetField("matchcode"));
   Console.WriteLine("Location Quality Code = " +
ab.GetField("location_quality_code"));
   Console.WriteLine("Longitude = " + ab.GetField("longitude"));
   Console.WriteLine("Latitude = " + ab.GetField("latitude"));
   Console.WriteLine(" ");
}
}
catch (AddressBrokerException abe)
{
Console.WriteLine("GetRecords/GetField exception: " + abe.Message);
}
catch (ArgumentOutOfRangeException rangeArg)
{
// Input field is invalid
Console.WriteLine("Argument out of range: " + rangeArg.Message);
}
catch (ArgumentNullException nullArg)
{
// Input field is null.
Console.WriteLine("Null argument: " + nullArg.StackTrace);
}
catch (Exception e)
{
Console.WriteLine("GetRecords/GetField exception (type " + e.ToString()
+ "): " + e.Message);
}
```

```
[Visual Basic]
Try
While abclient.GetRecord()
    'If this processed, get the output values
    sOutFirm = abclient.GetField("firmname")
    sOutAddress = abclient.GetField("Addressline")
    sOutLastline = abclient.GetField("Lastline")
    sOutMatchCode = abclient.GetField("match_code")
    sOutLatitude = abclient.GetField("Latitude")
    sOutLongitude = abclient.GetField("Longitude")
    sOutLocationCode = abclient.GetField("LocationQualityCode")
    'and display the results
    txtResults.Text = "Address Found:   " & vbCrLf & vbCrLf & "Firm: " &
sOutFirm & vbCrLf & "Address: " & sOutAddress & vbCrLf & "Lastline: " &
sOutLastline & vbCrLf & "Latitude: " & sOutLatitude & vbCrLf & "Longitude:
" & sOutLongitude & vbCrLf & "Match Code: " & sOutMatchCode & vbCrLf &
"LocationCode: " & sOutLocationCode
End While

Catch abe As AddressBrokerException
MsgBox(abe.Message, MsgBoxStyle.Critical, "AB AddressBrokerException")
Exit Sub
Catch nullArg As ArgumentNullException
MsgBox(nullArg.Message, MsgBoxStyle.Critical, "AB
ArgumentNullException")
Exit Sub
Catch rangeArg As ArgumentOutOfRangeException
MsgBox(rangeArg.Message, MsgBoxStyle.Critical, "AB
ArgumentOutOfRangeException")
Exit Sub
Catch er1 As Exception
MsgBox(er1.Message, MsgBoxStyle.Critical, "AB Exception: " &
er1.ToString())
Exit Sub
End Try
```

Step 7: Terminating the program

Invoke the Close method to terminate any active connections to the server.

```
[C#]
ab.Close();
```

```
[Visual Basic]
Try
abclient.Close()
Catch abe As AddressBrokerException
MsgBox(abe.Message, MsgBoxStyle.Critical, "AB AddressBrokerException")
Exit Sub
Catch er1 As Exception
MsgBox(er1.Message, MsgBoxStyle.Critical, "AB Exception: " &
er1.ToString())
Exit Sub
End Try
```

AddressBroker .NET methods

The methods described in this chapter are methods of three public classes/interfaces: `AddressBrokerFactory`, `ABClient`, and `AddressBrokerException`. Within each class/interface, methods are listed alphabetically.

Some methods are listed as:

```
MethodName (overloaded)
```

This indicates there are two or more methods with the same name whose behavior depends on the parameters it is given. For example, the same method accepts either a Boolean (`bool`) type or a string type.

Quick reference

AddressBrokerFactory class

Make

Creates and initializes instances of `ABClient` subclasses. Must be invoked before any other method. With the .NET API, you cannot directly instantiate a `ABClient` instance. Use the `AddressBrokerFactory` helper class to create an instance.

ABClient class

Field/data methods

Clear

Clears the input and output record buffers and resets all counter properties to zero.

GetField (overloaded)

Retrieves the value(s) of an output field in the current output record. Invoke iteratively for fields that contain multiple values.

GetFieldAttribute

Retrieves a field attribute, such as its data type and description.

ResetField

Resets the output field pointer to the first value of an output field.

SetField

Sets an input field value in the current input record.

GetRecord

Retrieves the record and advances the output record pointer.

ResetRecord

Resets the output record pointer to the first record of the output record buffer.

SetRecord

Adds the data for the current record to the input record buffer and advances the input record pointer to the next empty record.

Property methods

GetProperty (overloaded)

Retrieves the value of an input or output property.

GetPropertyAttribute (overloaded)

Retrieves a property attribute, such as its name, data type, and description.

SetProperty (overloaded)

Sets the value of a property.

ValidateProperties

Validates properties for consistency and completeness. This method must be invoked after `SetProperty` and before invoking `SetField`.

Processing methods

ProcessRecords

Processes a set of one or more address records.

LookupRecord

Processes a single incomplete address record.

Termination method

Close

Closes any active connections to a server.

AddressBrokerException class

Status code method

GetStatusCode

Retrieves the status code of a thrown exception.

AddressBrokerFactory class

Use the `AddressBrokerFactory` class to create concrete instances of the various subclasses of `ABClient`. The factory has only one method, `Make`.

Make

Creates instances of ABClient subclasses.

Syntax

```
ABClient AddressBrokerFactory.Make
( string in_hostlist,
  string in_transport,
  string in_user,
  string in_password )
```

Arguments

in_hostlist	A pipe (|)-delimited list of servers and associated parts in the form “host1:port1 | host2:port2 | ...”. *Input*.
in_transport	Case-insensitive string that specifies the network protocol AddressBroker uses.
in_user	A valid user name. *Input*.
in_password	A valid user’s password. *Input*.

Return Values

None.

Prerequisites

None.

Alternates

None.

Notes

The client transparently switches between servers if it has a problem establishing communication with its current server. That is, when the client executes a command that includes a server transaction, it switches servers if there is no response from the current server or a transaction fails.

An AddressBroker client uses the first server specified in *in_hostlist* until the server fails, at which point it switches to the next server listed in *in_hostlist*. The client continues to use this secondary server until it—the secondary server—fails. After a failed server is operational, it again becomes available to the client. However, the client does not switch back unless its current server fails. When a client searches for a server and encounters the end of *in_hostlist*, it continues searching from the beginning of the list.

On a per-transaction basis, the client tries each server in turn until it finds an operational server. If it fails to find a server, the operation fails.

When listing multiple servers, it is extremely important that they all service client requests identically. To ensure predictable results, make sure that the server INI files on each host use the same initialization settings.

There are two valid protocols for the **Make** method: SOCKET and NOCONNECT. Both SOCKET and NOCONNECT make standard sockets connections to the AddressBroker server. However, the SOCKET protocol actually makes a connection to the server and gets a list of properties as set by the Server INI file. The NOCONNECT protocol does not make that connection. NOCONNECT is appropriate for production environments where all processing is defined programmatically, and not by the end user.

An **InstantiationException** is thrown when an AddressBroker instance cannot be created.

An **ArgumentNullException** , **ArgumentOutOfRangeException**, or **AddressBrokerException** may be thrown from this method.

Example 1

```
// Socket protocol using the computer name
ab  = AddressBrokerFactory.Make ( “primary:1234 | secondary:1235”,
“socket”, “MyLogon”, “MyPassword” ) ;
```

Example 2

```
// Socket protocol using an URL
ab  = AddressBrokerFactory.Make ( “centrus.com:1234 | centrus-
software.com:1235”, “socket”, “MyLogon”, “MyPassword” ) ;
```

Example 3

```
// Socket protocol using an IP address
ab  = AddressBrokerFactory.Make ( “204.180.129.200:1234 |
209.38.36.44:1235”, “socket”, “MyLogon”, “MyPassword” ) ;
```

ABClient class

The `ABClient` interface provides all public methods required by the user. It is not possible to make a concrete `ABClient` instance. Instead, use the `AddressBrokerFactory` class to create an instance of `ABClient`.

Clear

Clears input and output record buffers and resets counter properties.

Syntax

```
bool ABClient.Clear ( )
```

Parameters

None.

Return Values

true if successful, **false** if unsuccessful.

Prerequisites

None.

Alternates

None.

Close

Forces any active connection to a server to close.

Syntax

```
void ABClient.Close ( )
```

Parameters

None.

Return Values

None.

Prerequisites

`Make`

Alternates

None.

Notes

The instance is no longer usable after invoking **`Close`**.

Failure to invoke **`Close`** may prevent your process from exiting when expected due to monitor threads persisting beyond the lifetime of your program's other threads.

GetField (overloaded)

Retrieves output field value(s) from the current output record.

Syntax

```
string ABClient.GetField (
   string in_FieldName )
string ABClient.GetField (
   string in_FieldName,
    string in_LogicalName )
```

Parameters

in_FieldName	A valid, fully specified field name listed in the `OUTPUT_FIELD_LIST` property (see the examples for this function). The property name is not case sensitive, and spaces and underscores are ignored. *Input*.
in_LogicalName	The logical name required by the value of *`in_FieldName`*. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

Return Values

Single value fields: returns the field value.

Multi-value fields: returns the current value and advances the pointer to the next value in the field.

Returns **null** when no values are found.

Prerequisites

`GetRecord`

Alternates

None.

Notes

The `GetField` method retrieves a field value from the current output record. Invoke `GetField` iteratively for multi-valued fields. Use the ResetField method to reset the field to its first value. To retrieve single value fields more than once, you must invoke `ResetField`.

An `ArgumentNullException` is thrown when:

- *`in_FieldName`* is **null** or the empty string (“”).

An `ArgumentOutOfRangeException` is thrown when:

- *`in_FieldName`* and/or *`in_LogicalName`* are invalid.
- *`in_FieldName`* is not in the `OUTPUT_FIELD_LIST property.`

An `AddressBrokerException` is thrown when:

- no output records are available.

All Spatial+, GDL, and Demographic fields require logical names. GeoStan and GeoStan Canada fields do not.

Example1

```
//Example using a field that does not require a logical name.
string fieldvalue = ab.GetField ("CITY");
```

Example 2

```
//Example using a field with its logical name in brackets.
string fieldvalue = ab.GetField ("PolygonName[COUNTIES]");
```

Example 3

```
//Example using a field with its logical name as a separate parameter.
string fieldvalue = ab.GetField ("PolygonName", "COUNTIES");
```

See Also

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information about fields.

GetFieldAttribute

Retrieves a field attribute.

Syntax

```
string ABClient.GetFieldAttribute (
    string in_FieldName,
    int in_FieldIOType,
    int in_AttributeId )
```

Parameters

in_FieldName	A valid field name listed in the `INPUT_FIELD_LIST` or `OUTPUT_FIELD_LIST` property. The property name is not case sensitive, and spaces and underscores are ignored. Do not associate logical names with field names when using this method. *Input*.
in_FieldIOType	A symbolic constant identifying the field name as an input field (**ABConst.AB_FIELD_INPUT**) or an output field (**ABConst.AB_FIELD_OUTPUT**). *Input*.
in_AttributeId	A symbolic constant identifying the attribute to retrieve. *Input*.

Return Values

Returns the value of the field's attribute. Integer values are returned as strings.

Prerequisites

`SetField`

`ValidateProperties`

Alternates

None.

Notes

`GetFieldAttribute` retrieves a field attribute's value. These are general attributes, not specific to a record. Valid attribute constants below are all public static members of the `ABConst` class.

Attribute Values

`AB_FIELD_DATA_TYPE`
"N" (numeric), "C" (character).

`AB_FIELD_DECIMALS`
Number of decimal places, if numeric.

`AB_FIELD_DESCRIPTION`
Short (32-character) description of field.

`AB_FIELD_HELP`
Long (255-character) field description. This is not implemented for all fields.

`AB_FIELD_LENGTH`
Field width.

`AB_FIELD_NEEDS_LOGICAL_NAME`

"0" (zero) = No logical name permitted.
"G" = A GeoStan logical name required.
"S" = A Spatial+ logical name required.
"D" = A DemoLib logical name required.
"C" = A GeoStan Canada logical name required.
"L" = A GDL logical name required.

`AB_FIELD_NUM_VALUES`
Maximum number of unique values possible for field.

An **`ArgumentNullException`** is thrown when:

- *`in_FieldName`* is **null** or the empty string ("").

An **`ArgumentOutOfRangeException`** is thrown when:

- *`in_FieldName`* is invalid.
- *`in_FieldIOType`* is not in `AB_INPUT_FIELD` or `AB_OUTPUT_FIELD` (global .NET constants).
- *`in_FieldIOType`* contains an invalid value.
- *`in_AttributeId`* contains an invalid value.

An **`AddressBrokerException`** is thrown when:

- ValidateProperties is not invoked prior to **`GetFieldAttribute`**.
- There is a communication problem with the server.

Example

```
try
{
```

```
   ab.ValidateProperties();
   string fieldattr = ab.GetFieldAttribute
   ("CITY",ABConst.AB_FIELD_INPUT, ABConst.AB_FIELD_LENGTH );
   fieldattr = ab.GetFieldAttribute ( "PolygonName",
ABConst.AB_FIELD_OUTPUT,ABConst.AB_FIELD_DATA_TYPE );
}
```

See Also

See "INPUT_FIELD LIST and OUTPUT_FIELD_LIST" on page 66 for more information about fields.

GetProperty (overloaded)

Retrieves a property value.

Syntax

```
Object ABClient.GetProperty (
   string in_PropName )
Object ABClient.GetProperty (
   int in_PropId )
```

Parameters

in_PropName	A valid property name. The property name is not case sensitive. Spaces and underscores are ignored. *Input*.
in_PropID	A valid property symbolic constant. *Input*.

Return Values

Returns the property value. The returned value **`Object`** is of type String, Integer, or Boolean, corresponding to the property's data type. Cast the return value to the appropriate type.

Prerequisites

None.

Alternates

None.

Notes

The **`GetProperty`** methods retrieve a property value.

An `ArgumentNullException` is thrown when:

- *`in_PropName`* is **null** or the empty string ("").

An `ArgumentOutOfRangeException` is thrown when:

- *`in_PropName`* and/or *`in_PropID`* are invalid.

Example

```
bool propvalue = (bool)ab.GetProperty ("MIXED CASE");
string propvalue = (string)ab.GetProperty  (ABConst.AB_INIT_LIST);
```

See Also

See Chapter 13 Properties for more information about properties.

GetPropertyAttribute (overloaded)

Retrieves a property attribute.

Syntax

```
string ABClient.GetPropertyAttribute (
   string in_PropName,
   int in_AttributeId )
string ABClient.GetPropertyAttribute (
   int in_PropID,
   int in_AttributeId )
```

Parameters

in_PropName	A valid property name. The property name is not case sensitive. Spaces and underscores are ignored. *Input*.
in_PropID	A valid property symbolic constant. *Input*.
in_AttributeId	A symbolic constant of the attribute to retrieve. *Input*.

Return Values

Returns the value of the attribute (see the examples for this function).

Prerequisites

`SetProperty` if you want client property information.

Alternates

None.

Notes

An **ArgumentNullException** is thrown when:

- *in_PropName* or *in_PropID* is **null** or the empty string ("").

An **ArgumentOutOfRangeException** is thrown when:

- *in_PropName* or *in_PropID* is invalid.
- *in_AttributeId* contains an invalid value.

To receive information about properties set on the server, call **Make**. To get server property information, call **GetPropertyAttribute** before setting any properties in the client code. To receive information about client properties, call **GetPropertyAttribute** after calling **SetProperties**.
Valid attribute constants below are all public static members of the **ABConst** class.

Attribute Values

`AB_PROPERTY_DATA_TYPE`
: "N" (Integer), "B" (Boolean), or "C" (String).

`AB_PROPERTY_DEFAULT_VALUE`
: Default property value.

`AB_PROPERTY_DESCRIPTION`
: Short (100-character) description of property.

`AB_PROPERTY_ID`
: Property ID.

`AB_PROPERTY_LENGTH`
: Length of property value.

`AB_PROPERTY_NAME`
: Property name.

`AB_PROPERTY_READ_ONLY`
: "1" property is read-only.
"0" property is read/write.

Example1

```
//Example using the Property Name
string propattr = ab.GetPropertyAttribute ("MIXED CASE",
ABConst.AB_PROPERTY_DATA_TYPE);
```

Example 2

```
//Example using the Property ID
string propattr = ab.GetPropertyAttribute (ABConst.AB_INIT_LIST,
ABConst.AB_PROPERTY_LENGTH);
```

See Also

See Chapter 13 Properties for more information about properties.

GetRecord

Advances the pointer to the next record in the output record buffer.

Syntax

```
bool ABClient.GetRecord ( )
```

Parameters

None.

Return Values

true if successful, **false** if unsuccessful.

Prerequisites

`ProcessRecords`

Alternates

None.

Notes

The first time **`GetRecord`** is invoked, it sets a pointer in the output record buffer to the first output record. Subsequent calls to **`GetRecord`** advance the pointer. When no further records are found, **false** is returned.

Use the **`GetField`** method to retrieve values from individual record fields. Use the ResetRecord method to reset the output record pointer to the first output record.

Possible exceptions thrown in case of error include: **`AddressBrokerException`**, **`IOException`**, and **`SocketException`**.

Example

```
while ( ab.GetRecord() )
{
for (int i = 0; i < fieldnames.length; ++i)
{
string value = ab.GetField(fieldnames[i]);
}
}
```

LookupRecord

Processes a single incomplete U.S. address record or performs a reverse lookup on a Canadian postal code.

Syntax

```
int ABClient.LookupRecord ( )
```

Parameters

None.

Return Values

The `OUTPUT_FIELD_LIST` property defines the fields populated by **`LookupRecord`**, and the return codes listed below describe the search outcome. Individual codes are returned only when the relevant fields are included in `OUTPUT_FIELD_LIST`. A return value of zero (**0**) indicates an internal failure.

Notes

Valid attribute constants below are all public static members of the **`ABConst`** class.

Return Codes

`AB_LOOKUP_ADDRESS_LINE_INCOMPLETE`

For a U.S. address, the firm name or unit number could not be resolved. Multiple incomplete records returned. The user can be prompted to submit more information. The most useful fields for resolving a match generally are `FirmName`, `HighUnitNumber`, `LowUnitNumber`, `MatchCode`, and `UnitType`.

Other helpful fields include `AddressLine`, `AddressLine2`, `CarrierRoute`, `CountyName`, `FIPSCountyCode`, `GovernmentBuildingIndicator`, `HighEndHouseNumber`, `LACSAddress`, `LastLine`, `LowEndHouseNumber`, `PostfixDirection`, `PrefixDirection`, `RoadClassCode`, `SegmentBlockLeft`, `SegmentBlockRight`, `State`, `UrbanizationName`, `USPSRangeRecordType`, `ZIP`, `ZIPCarrtSort`, `ZIPCityDelivery`, `ZIPClass`, `ZIPFacility`, and `ZIPUnique`.

For a Canadian postal code, the input Postal Code is resolved to a range of possible addresses that contain a single street number. The street number suffix or unit number values will vary over the range.

`AB_LOOKUP_LAST_LINE_NOT_FOUND`

For a U.S. address, multiple incomplete records returned; did not resolve LastLine. Iteratively invoke GetRecord to retrieve the possible matches. Only the following output fields are returned: `MatchCode`, `CITY`, `State`, `ZIP`, and `ZIPFacility`. For a Canadian postal code, this return code indicates that the input postal code was not found in the CPC data and is invalid.

`AB_LOOKUP_MULTIPLE_MATCH`

For a U.S. address, the address resolved to multiple match. Multiple complete address records returned. Iteratively invoke GetRecord to retrieve possible matches. For a Canadian postal code, the postal code resolved to a range of possible addresses that vary over the street.

`AB_LOOKUP_NOT_FOUND`

The address could not resolve to match or possible match. No records returned. Provide a more complete address. (This return code is not used for Canada.)

`AB_LOOKUP_SUCCESS`

For a U.S. address, a complete single address was matched and returned. (This return code is not used for Canada.)

`AB_LOOKUP_TOO_MANY_CITIES`

No records returned. An incomplete LastLine matched over 100 cities. Provide a more complete address. (This return code is not used for Canada.)

Prerequisites

None.

Alternates

`SetRecord`

Notes

`LookupRecord` processes a single input record and should be used only when address information is insufficient for standardization. To process single or multiple records containing complete addresses, use ProcessRecords.

Minimally, address information for **`LookupRecord`** must include a street number, a partial street name, and/or valid LastLine information. For Canada, a valid postal code is required and will return a single address or a range of addresses.

`LookupRecord` is most useful in interactive programs, when an application may have to invoke `LookupRecord` iteratively to find a match for an incomplete address. In client/server and Internet environments, the record is transferred across the network with each call to `LookupRecord`. The method does not return until the record is processed. When `LookupRecord` processes an address record and fails to find an exact match, it does an extensive search to find cities and streets that are possible matches.

The `INPUT_FIELD_LIST` property specifies the list of fields passed to `LookupRecord`. Generally, provide at least `FirmName`, `AddressLine`, and `LastLine` fields as input to `LookupRecord`. For Canada, a valid Canadian Postal Code is the only input, and it is set using the PostalCode input field. Only one Postal Code can be processed at a time.

The `OUTPUT_FIELD_LIST` property specifies the list of possible fields returned.

The `MAXIMUM_LOOKUPS` property limits the number of multiples—possible matches—that are returned by `LookupRecord`. The upper limit of `MAXIMUM_LOOKUPS` is 100. For a Canadian postal code, if the `MAXIMUM_LOOKUPS` is set to 100, AddressBroker increases the `MAXIMUM_LOOKUPS` to 200.

Retrieve the list of possible matches using a '`while (GetRecord) do GetField`' loop. No records are returned when the return value of `LookupRecord` is **AB_LOOKUP_NOT_FOUND** or **AB_LOOKUP_TOO_MANY_CITIES**.

Precisely recommends using `ProcessRecords` instead of `LookupRecord`.

An `IOException` is thrown if the client receives a corrupted message, for example, when there is a failure in the network transport layer.

AddressBroker throws an `AddressBrokerException` when:

- Severe problems occur when processing a user request.
- A time-out occurs.
- There are logic errors.

Example

In an interactive application, a user submits a partial address to `LookupRecord`. The return code is **AB_LOOKUP_LAST_LINE_NOT_FOUND**. For a U.S. address, this code indicates that the user did not enter enough information for `LookupRecord` to resolve the city, state, or ZIP Code. The application prompts the user to select from the list of possible cities and states returned by `LookupRecord`. The user selects the necessary information and resubmits the address to `LookupRecord`. For a Canadian postal code, this return code indicates that the input postal code was not found in the CPC data and is invalid.

This time the return code is **AB_LOOKUP_ADDRESS_LINE_INCOMPLETE**. The user resolved the last line problem, but the return code indicates the address line could be more specific. For a U.S. address, it is missing information on the firm name or unit number (suite, apartment, etc.). The application can prompt the user to select from the list of possibilities returned by this call to `LookupRecord`. The user enters the additional information and resubmits the address to `LookupRecord`, and **AB_LOOKUP_SUCCESS** is returned. For a Canadian postal code, the **AB_LOOKUP_ADDRESS_LINE_INCOMPLETE** code indicates that the input Postal Code resolved to a range of possible addresses that contain a single street number. The street number suffix or unit number values will vary over the range. For example, a Canadian postal code of T3C 2K7 could resolve to 123 A - 123 G Maple Street (when the street suffix varies) or 123 Maple Street Unit 1-100 (when the unit number changes). A valid postal code for one address submitted to `lookupRecord` returns **AB_LOOKUP_SUCCESS**.

When the next address is entered, `LookupRecord` returns the status code **AB_LOOKUP_MULTIPLE_MATCH.** This indicates multiple complete matches were found. For a U.S. address, the user may then be prompted to select from the list of possible matches. The selected address is resubmitted to `LookupRecord` to ensure that it is entirely correct, and that **AB_LOOKUP_SUCCESS** is returned. For a Canadian postal code, the **AB_LOOKUP_MULTIPLE_MATCH** code indicates a postal code that resolved to a range of possible addresses that vary over the street. For example, a Canadian postal code could resolve to 100-120 Elm, Calgary, AB or 150-165 Maple, Calgary, AB.

ProcessRecords

Processes a set of one or more address records.

Syntax

```
void ABClient.ProcessRecords ( )
```

Parameters

None.

Return Values

None.

Prerequisites

`SetRecord`

Alternates

None.

Notes

Each record should contain enough address information for standardization. For records containing incomplete addresses, use LookupRecord, which progressively returns address choices for one input record at a time.

The method call does not return until all of the records are processed.

An `IOException` is thrown if the client receives a corrupted message; for example, when there is a failure in the network transport layer.

AddressBroker throws an `AddressBrokerException` when:

- severe problems occur when processing a user request.
- a time-out occurs.
- there are logic errors.

See Also

See Chapter 13 Properties for more information about properties.

See "INPUT_FIELD LIST and OUTPUT_FIELD_LIST" on page 66 for more information about fields.

ResetField

Resets the output field pointer to the first value of a multi-valued output field.

Syntax

```
bool ABClient.ResetField (
   string in_FieldName,
   string in_LogicalName )
```

Parameters

in_FieldName	A valid field name listed in the `OUTPUT_FIELD_LIST` property. Some field names require a logical name. The logical name may be appended to `in_FieldName` in brackets, or passed in the `in_LogicalName` parameter (see the examples for this function). The property name is not case sensitive, and spaces and underscores are ignored. *Input*.
in_LogicalName	The logical name required by the value of `in_FieldName`. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

Return Values

true if successful, **false** if unsuccessful.

Prerequisites

`GetField`

Alternates

None.

Notes

The output field pointer is reset to the first value of the output field.

`ResetField` returns **false** when *`in_FieldName`* is not found.

An `ArgumentNullException` is thrown when:

- *`in_FieldName`* is **null** or the empty string (“”).

An `ArgumentOutOfRangeException` is thrown when:

- A logical name is provided in both *`in_FieldName`* and *`in_LogicalName`*.

If `GetField` is called with the logical name in brackets, `ResetField` should be called with the logical name in brackets. Similarly, if the logical name is passed as a separate parameter in `GetField`, then `ResetField` must also use separate parameters. This is for consistency purposes only; does not cause an error.

All Spatial+, GDL, and Demographic fields require logical names. GeoStan and GeoStan Canada fields do not.

Example 1

```
// Example using field name with its logical name in brackets.
while ( ab.GetField ( "polygonName[COUNTIES]" )==true)
{
...
}
ab.ResetField ("PolygonName","PolygonName[Counties]");
```

Example 2

```
// Example using field name with its logical name as separate
parameter.
while ( ab.GetField ( "polygonName", "COUNTIES" )==true)
{
...
}
ab.ResetField ("PolygonName", "Counties");
```

See Also

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information about fields.

ResetRecord

Resets output record pointer to the first record in the output record buffer.

Syntax

```
bool ABClient.ResetRecord ( )
```

Parameters

None.

Return Values

true if successful, **false** if unsuccessful.

Prerequisites

`GetField`

Alternates

None.

SetField

Sets an input field value in the current input record.

Syntax

```
void ABClient.SetField (
   string in_FieldName,
   string in_FieldValue )
```

Parameters

in_FieldName	A valid field name listed in the `INPUT_FIELD_LIST` property. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.
in_FieldValue	The string value to assign to the field. Maximum string length is determined by the `AB_FIELD_LENGTH` field attribute. *Input*.

Return Values

None.

Prerequisites

`SetProperty`

Alternates

None.

Notes

The `RECORD_DELIMITER`, `FIELD_DELIMITER`, and `VALUE_DELIMITER` properties have default values of line feed, tab, and CTRL-A, respectively. If your data contains any of these characters, you *must* reset the appropriate property to a different character. In addition, your data may not contain the NULL character.

An **`ArgumentNullException`** is thrown when:

- *`in_FieldName`* is **null** or the empty string (“”).
- *`in_FieldValue`* is **null**.

An **`ArgumentOutOfRangeException`** is thrown when:

- *`in_FieldName`* is invalid.
- *`in_FieldName`* is not in the `INPUT_FIELD_LIST` property.
- The length of *`in_FieldValue`* is > 256 characters.

An `AddressBrokerException` is thrown when:

- Properties were set (via `SetProperty`) but were not validated (via ValidateProperties).

Example

```
ab.SetField (“AddressLine”, “123 Main”);
ab.SetField (“LastLine”, “Anytown, NY”);
```

See Also

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information about fields.

SetProperty (overloaded)

Assigns a property value.

Syntax

```
void ABClient.SetProperty (
    string in_PropName,
    bool in_bPropValue )
void ABClient.SetProperty (
    string in_PropName,
    bool in_bPropValue )
void ABClient.SetProperty (
    string in_PropName,
    string in_sPropValue )
void ABClient.SetProperty (
    string in_PropName,
    Integer in_iPropValue )
void ABClient.SetProperty (
    string in_PropName,
    int in_iPropValue )
void ABClient.SetProperty (
    int in_PropID,
    bool in_bPropValue )
void ABClient.SetProperty (
    int in_PropID,
    bool in_bPropValue )
void ABClient.SetProperty (
    int in_PropID,
    string in_sPropValue )
void ABClient.SetProperty (
    int in_PropID,
    Integer in_iPropValue )
void ABClient.SetProperty (
    int in_PropID,
    int in_iPropValue )
```

Parameters

in_PropName	A valid property name. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.
in_PropID	The valid symbolic constant of the property being set. *Input*.
in_bPropValue	A Boolean object or Boolean value to assign to the property. *Input*.
in_sPropValue	A string value to assign to the property. *Input*.
in_iPropValue	An integer object or integer value to assign to the property. *Input*.

Return Values

None.

Prerequisites

`AddressBrokerFactory.Make`

Alternates

None.

Notes

The specific `SetProperty` method to use depends on the data type of the property you are setting.

An `ArgumentNullException` exception is thrown when:

- *in_PropName* or *in_PropID* are **null** or invalid.
- The property value is **null**.

An `AddressBrokerException` is thrown when:

- The data type of the property does not correspond to the data type of the value.

Example

```
ab.SetProperty ("MIXED CASE", true);
ab.SetProperty (ABConst.AB_INIT_LIST, "GEOSTAN |COUNTIES");
```

See Also

See Chapter 13 Properties for more information about properties.

SetRecord

Adds data for the current record to the input record buffer and advances the input record pointer to the next empty record in the buffer.

Syntax

```
void ABClient.SetRecord ( )
```

Parameters

None.

Return Values

true if successful, **false** if unsuccessful.

Prerequisites

`SetField`

Alternates

None.

ValidateProperties

Validates properties for consistency and completeness.

Syntax

```
void ABClient.ValidateProperties ( )
```

Parameters

None.

Return Values

true if successful, **false** if unsuccessful.

Prerequisites

`SetProperty`

Alternates

None.

Notes

The `ValidateProperties` method verifies the values of initialization and processing control properties to ensure a complete and compatible set of values are available to AddressBroker. Call this method after one or more properties have been set and before calling SetField or any processing methods.

When `ValidateProperties` returns **true**, it indicates all properties have been successfully validated and that AddressBroker is ready to process records. In some cases, all properties can be validated in a single method call.

See Also

See Chapter 13 Properties for more information about properties.

AddressBrokerException class

AddressBroker methods throw an object of this class to indicate run-time, logical, or processing errors.

GetStatusCode

Retrieves the Status Code from a thrown exception.

Syntax

```
long AddressBrokerException.GetStatusCode ( )
```

Parameters

None.

Return Values

Returns the 10-digit integer status code.

Prerequisites

None.

Alternates

None.

Notes

`GetStatusCode` is available only while an exception object is accessible (in scope).

See Also

See "GeoStan location codes" on page 433 for a description of status codes. See "AddressBroker .NET exceptions" on page 188 for more information on this exception class.

AddressBroker .NET exceptions

In the .NET API, many AddressBroker methods have no return codes as compared to the C and C++ APIs. Instead, your application must use exception handling. Exceptions are listed in the method syntax statements.

The AddressBroker .NET API throws the following classes of exceptions:

- `Centrus.Addressbroker.AddressBrokerException` – a general run time exception.
- `System.ArgumentNullException` – a parameter to a method is null or the empty string (" ").
- `System.ArgumentOutOfRangeException` – a parameter to a method is invalid or out of range.
- `System.IO.IOException` – the output stream from a request to the server was corrupted.

AddressBrokerException class

An object of this class is thrown by the methods of the `ABClient` class to indicate a run-time, logical, or processing error. This exception class extends the `System.Exception` by adding a status code and message. AddressBrokerException handling example shows an `AddressBrokerException try` block example. See "GetStatusCode" on page 187 for information about the `GetStatusCode` method.

AddressBrokerException handling example

```
...
    try {
         myAddressBrokerInstance.GetField("NONSENSE NAME");
    catch( AddressBrokerException abException ) {
         // Unknown field name error
         Console.WriteLine("An exception occurred:\n" + abException);
        Console.WriteLine("ErrorCode = " + abException.GetStatusCode());
    }
...
```

ArgumentNullException class

Parameters passed to methods are checked for correctness. ArgumentNullException handling example shows an example that checks for an `ArgumentNullException`.

ArgumentNullException handling example

```
...
     try {
         myAddressBrokerInstance.GetField(null);
     catch( ArgumentNullException illArgExcept ) {
         // Unknown field name error
         Console.WriteLine(illArgExcept);
    }
```

. . .

IOException class

AddressBroker throws an exception of this class when the output stream received from a ProcessRecords or a LookupRecord call is corrupted.

10 – C API

In this chapter

This chapter describes the C API to AddressBroker in detail.

This chapter provides a tutorial using the AddressBroker C API. The tutorial shows you how to use most of AddressBroker's functionality, yet is general enough that you can modify it for other uses. A complete function reference follows the tutorial. The final section of this chapter discusses error handling.

The naming convention for AddressBroker C API functions is **`QABFunctionName`**. All C functions use this naming convention.

Accessing the AddressBroker C libraries

To use the AddressBroker library in a client application, you must include the appropriate header file in your application source code files:

```
#include "ABapi.h"      // C API
```

You must also use the appropriate syntax for creating an AddressBroker handle or instance:

```
// C API
```

```
ab = QABInit ( AB_CLIENT, “primary:1234 |
secondary:1235”, “socket”, “MyLogon”, “MyPassword”,
“MyInitFile” );
```

Finally, you must include the AddressBroker import library in the link step of your build.

Windows platforms

Link to the AB.lib import library, which causes your application to use AB.dll. For your application to execute properly, this DLL must be found in your execution path environment variable.

UNIX platforms

Link to libab.sl or libab.so, which causes your application to dynamically bind to the AddressBroker library. For your application to execute properly, this shared library must be found in your shared library path environment variable: `SHLIB_PATH` for HP-UX, or `LD_LIBRARY_PATH` for most other UNIX systems.

Note: To process Canadian addresses, `NCODEDATA` and `LD_LIBRARY_PATH` for Solaris or `SHLIB_PATH` for HP-UX must be set. See the *GeoStan Canada Reference Manual* for more information.

AddressBroker C tutorial

This section describes the steps necessary to develop an AddressBroker application using the C API. The example shows some basic C sample code that performs address record enhancement. It uses the firm name and address fields from Precisely address records as input. This tutorial standardizes the address data and augments it with city, state, and 9-digit ZIP Code information from the GeoStan data directory. Then it retrieves the name and status of the geographic polygon where the address is located using a Spatial+ data file.

Sample C code (Console.c) is located in the Samples subdirectory.

Step 1: Create and initialize the object

To begin, link your application to the AddressBroker import library. Your application must include the “`ABapi.h`” header file, which defines AddressBroker C function prototypes. This header file also includes “`ABtypes.h`,” which defines AddressBroker data types. You do not need to include “`ABtypes.h`” in your source code.

C program initialization

```
/* Centrus AddressBroker includes. */
#include "ABapi.h"
/* A sample MAIN function. */
```

```
main ()
{
    ABId ab;
/* Specify the initialization file */
    char* initfile  = "C:\Program Files\Centrus\abclient.ini";
/* If client, specify... */
   /* ...the machine name where the server is running */
        char* hostname  = "MyServer";
   /* ...the network transport protocol */
        char* transport = "Socket";
   /* ...the logon name where the server is running */
        char* logon     = "MyLogon";
   /* ...the password where the server is running */
        char* password  = "MyPassword";
        unsigned long status_code;
        char          status_msg[2048];

/* If the application is executing as a client: */
       ab = QABInit ( AB_CLIENT, hostname, transport, logon, password,
inifile );
        QABGetStatus ( ab, status_code, status_msg, 2048 );
        if ( status_code )
        {
            printf ( status_msg );
            /* handle status condition  */
            ...
        }
```

Step 2: Set properties

You should assign a minimal set of properties in your client application. For a detailed discussion, see Chapter 5, "Client Applications".

Set logical names and paths on the server. The logical names the client uses must match those set on the server. In the sample code shown in "C QABSetProperty example" on page 193 the logical names GEOSTAN, GEOSTAN_Z9, and COUNTIES refer to a GeoStan data directory, a GEOSTAN ZIP Code file, and a Spatial+ polygon file, respectively. Next, tell AddressBroker to use the FirmName, AddressLine, and LastLine field values from each input record. In this example, the FirmName and AddressLine fields are enhanced with City, State, and ZIP10 information from the GeoStan data file. PolygonName and PolygonStatus are also retrieved from the COUNTIES file.

You can set other properties in the client. In the sample code, KEEP_MULTIMATCH and BUFFER_RADIUS are set. See Chapter 13, "Properties" for a detailed discussion.

C property reference syntax

```
/* setting a property using its string name */

sprintf(buffer, "%d", TRUE );
QABSetPropertyStr ( ab, "MIXED CASE", buffer );

/* setting a property using its property ID */
```

```
sprintf( buffer, "%d", TRUE );
QABSetPropertyID ( ab, AB_MIXED_CASE, buffer );

/* setting a pre-defined property */

sprintf( buffer, "%d", AB_INPUT_PARSED);
QABSetPropertyStr( ab, "INPUT MODE", buffer );
```

C QABSetProperty example

```
/* Tell Centrus AddressBroker what logical names to use. These must match
the logical names set in the server .ini file. */

        QABSetPropertyStr( ab, "INIT_LIST", "GEOSTAN | GEOSTAN_Z9 |
COUNTIES" );
/* Tell Centrus AddressBroker what input to use. We do this only once in
the example; it is a dynamic property, you can set it at any time, as many
times as you want. */
        QABSetPropertyStr( ab, "INPUT_FIELD_LIST",
                        "Firmname| AddressLine| LastLine" );
/* List the output fields we expect returned. */
        QABSetPropertyID( ab, AB_OUTPUT_FIELD_LIST,
"Firmname|AddressLine|City|State|Zip10|PolygonName[COUNTIES]|
PolygonStatus[COUNTIES]" );
// Set some other properties that affect server behavior.
         /* Keep only one output record for each input record.*/
         QABSetPropertyStr( ab, "KEEP_MULTIMATCH", FALSE );
         /* Set a 200 foot buffer instead of using the default. */
         QABSetPropertyStr( ab, "BUFFER_RADIUS", 200 );
```

Step 3: Validate properties (optional)

Use the QABValidateProperties function to send the property definitions to AddressBroker for validation. When `QABValidateProperties` returns **TRUE**, the AddressBroker client object initializes and is ready for processing. If any property setting is invalid, AddressBroker generates an error. Use QABGetStatus to retrieve error messages in the event `QABValidateProperties` does not return successfully.

All AddressBroker properties must be set and validated before data can be input or processed. In client mode, calling QABValidateProperties results in a server transaction.

C QABValidateProperties example

```
/* Check to see that properties are valid. */
if ( !QABValidateProperties ( ab ) )
{
        unsigned long status_code;
        char          status_msg[2048];
        QABGetStatus ( ab, status_code, status_msg, 2048 );
        ...
}
```

`QABValidateProperties` can be called multiple times in your application. For example, you can initially set and validate a group of properties, then select new values and revalidate the settings.

Step 4: Enter input records and field values

Next, use the QABSetField function call to specify the input field values. Note that these are the same fields you specified initially with the INPUT_FIELD_LIST property in the QABSetPropertyID or QABSetPropertyStr function call. (See "C QABSetProperty example" on page 193.)

The QABSetRecord function call adds the data for the current record to the input record list and advances the record pointer.

An input value need not be set for every field in a record. In this example, an individual record that did not contain FirmName information could still be processed.

C data input example

```
/* Enter a few records for processing.
   Fill in a record... */
        QABSetField( ab, "FirmName",    "Centrus");
        QABSetField( ab, "AddressLine", "4750 Walnut");
        QABSetField( ab, "LastLine",    "Boulder, CO");
      /* SetRecord will result in an error if SetField is never called. */
        QABSetRecord( ab );
        /* Fill in the next record... */
        QABSetField( ab, "FirmName",    "White House");
        QABSetField( ab, "AddressLine", "1600 Pennsylvania");
        QABSetField( ab, "LastLine",    "Washington, DC");
        QABSetRecord( ab );
```

Step 5: Process records

After all the input data has been entered, you are ready to process the records. Use the QABProcessRecords function to process records. In client mode, this sends all the data to the server for processing.

Note: This function call clears the input record buffer, even if the call fails.

C record processing example

```
if (!QABProcessRecords ( ab ) )
{
        unsigned long status_code;
        char          status_msg[2048];
        QABGetStatus ( ab, status_code, status_msg, 2048 );
        ...
}
```

Step 6: Retrieve address records and field values

Use the QABGetRecord and QABGetField function calls to retrieve the output data. C data retrieval example uses printf to display the output.

In your C applications, loop through Steps 4 through 6 of this tutorial each time you process additional records. You can also repeat Steps 2 and 3 to modify property settings.

C data retrieval example

```
char  firmname[41];
char  addressline[61];
char  city[29];
char  state[3];
char  zip10[11];
char  polygonname[128];
char  polygonstatus[2];
/* For each record that comes back... */
  while ( QABGetRecord ( ab ) )
   {
     /* Get address data. */
     QABGetField( ab, "FirmName", firmname, 41 );
     QABGetField( ab, "AddressLine", addressline, 61 );
     QABGetField( ab, "City", city, 29 );
     QABGetField( ab, "State", state, 3 );
     QABGetField( ab, "ZIP10", zip, 11 );
     /* Print out the basic address */
     printf( "Firm =  %s\n", firmname );
     printf( "Addr =  %s\n", addressline );
     printf( "City =  %s\n", city );
     printf( "State = %s\n", state );
     printf( "ZIP =   %s\n", zip10 );
     /* Get polygon name and status with a multivalued return */
     while ( QABGetField( ab, "PolygonName[COUNTIES]", polygonname, 128
) )
       {
       /* Print out the polygon name... */
         printf( "Polygon Name = %s\n", polygonname );
       /*...and the polygon status paired with each polygon name found. */
        QABGetField( ab, "PolygonStatus", "COUNTIES", polygonstatus, 2 );
         /* Print out the polygon status. */
         switch ( polygonstatus[0] )
           {
             case 'P':
               printf ("   (address is inside the polygon)\n")
               break;
               case 'I':
              printf ("   (address is inside the polygon and within the
buffer radius)\n")
               break;
               case 'B':
             printf ("   (address is outside the polygon but within the
buffer radius)\n")
               break;
               default:
               /* This should never happen. */
               printf ("   (unknown condition)\n")
               break;
           }
       }
   }

   /* Clean up and quit */
   QABTerm (ab);
```

AddressBroker C functions

This section describes in detail the functions available through the AddressBroker C API.

Quick reference

Initialization functions

QABInit

Creates and initializes an AddressBroker handle. Must be called before any other function.

Property functions

QABGetPropertyID

Retrieves the value of an input or output property.

QABGetPropertyStr

Retrieves the value of an input or output property.

QABGetPropertyAttribute*

Retrieves a property attribute, such as its name, data type, and description.

QABSetPropertyID

Sets the value of a property.

QABSetPropertyStr

Sets the value of a property.

QABValidateProperties

Validates properties for consistency and completeness. This function must be called after `QABSetProperty` and before calls to `QABSetField`.

Field/Data functions

QABClear

Clears the input and output record buffers and resets all counter properties to zero.

QABGetField

Retrieves the value(s) of an output field in the current output record. Call iteratively for fields that contain multiple values.

QABGetFieldAttribute

Retrieves a field attribute, such as its data type and description.

QABResetField

Resets the output field pointer to the first value of an output field.

QABSetField

Sets an input field value in the current input record.

QABGetRecord

Retrieves the record and advances the output record pointer.

QABResetRecord

Resets the output record pointer to the first record of the output record buffer.

QABSetRecord

Adds the data for the current record to the input record buffer and advances the input record pointer to the next empty record.

Processing functions

QABProcessRecords

Processes a set of one or more address records.

QABLookupRecord

Processes a single incomplete address record.

Reporting functions

QABGetStatus

Retrieves status or error codes and messages.

QABSetLogFn

Call back function for handling error messages.

Termination

QABTerm

Destroys a QMSAddressBroker handle.

QABClear

Clears input and output record buffers and resets counter properties.

Syntax

```
Boolean QABClear ( ABId ab )
```

Arguments

ab	The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

None.

Alternates

None.

QABGetField

Retrieves output field value(s) from the current output record.

Syntax

```
Boolean QABGetField
    (ABId ab,
    const char* in_pszFieldName,
    const char* in_pszLogicalName,
    char* out_pszFieldValue,
    UInt32 in_usBufferSize)
```

Arguments

ab — The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.

in_pszFieldName — A valid, fully specified field name listed in the `OUTPUT_FIELD_LIST` property (see the examples for this function). The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

in_pszLogicalName — A valid, fully specified logical name listed in the `OUTPUT_FIELD_LIST` property (see the examples for this function). The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

out_pszFieldValue — Pointer to the field value to retrieve. All values are returned as strings. *Output*.

in_ulBufferSize — The size of the string buffer. *Input.*

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful or no values are found.

Prerequisites

`QABGetRecord`

Alternates

None.

Notes

These functions retrieve a field value from the current output record. Call them iteratively for multi-valued fields. Use the QABResetField function to reset the field to its first value. To retrieve single value fields more than once, you must call **`QABResetField`**.

All Spatial+, GDL, and Demographic fields require logical names. GeoStan and GeoStan Canada fields do not.

Example 1

```
/* Example using a field that does not require a logical name. */
char city[29];

while(QABGetRecord(ab))
{
...
QABGetField ( ab, "City", city, 29 );
...
}
```

Example 2

```
/* Example using a multivalued field with its logical name in the
fieldname
argument. */
char polygonname[128];
while ( QABGetField ( ab, "PolygonName[Counties]", NULL, polygonname,
128) )
{
  ...
}
```

Example 3

```
/* Example using a multivalued field with its logical name as separate
argument. */
char polygonname[128];
while ( QABGetField ( ab, "PolygonName", "Counties", polygonname, 128)
)
{
  ...
}
```

See Also

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information about fields.

QABGetFieldAttribute

Retrieves a field attribute.

Syntax

```
Boolean QABGetFieldAttribute ( ABId ab,
   char* in_pszFieldName,
   unsigned long in_ulFieldIOType,
   unsigned long in_ulAttributeName,
   char* out_pszAttributeValue,
   unsigned long in_ulBufferSize )
```

Arguments

ab	The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.
in_pszFieldName	A valid field name listed in the ALL_INPUT_FIELDS or ALL_OUTPUT_FIELD_LIST property. The property name is not case sensitive, and spaces and underscores are ignored. Do not associate logical names with field names when using this function. *Input*.
in_ulFieldIOType	A symbolic constant identifying the field name as an input field (AB_FIELD_INPUT) or an output field (AB_FIELD_OUTPUT). *Input*.
in_ulAttributeName	A symbolic constant identifying the attribute to retrieve. *Input*.
out_pszAttributeValue	Pointer to the attribute value to retrieve. All values are returned as strings. *Output*.
in_ulBufferSize	The size of the string buffer. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

QABSetField

QABValidateProperties

Alternates

None.

Notes

`QABGetFieldAttribute` retrieves a field attribute's value. These are general attributes, not specific to a record.

Attribute Values

`AB_FIELD_DATA_TYPE (size = 2)`
"N" (numeric), "C" (character).

`AB_FIELD_DECIMALS (size = 12)`
Number of decimal places, if numeric.

`AB_FIELD_DESCRIPTION (size = 33)`
Short (32-character) description of field.

`AB_FIELD_HELP (size = 256)`
Long (255-character) field description. This is not implemented for most fields.

`AB_FIELD_LENGTH (size = 12)`
Field width.

`AB_FIELD_NEEDS_LOGICAL_NAME (size = 2)`

"0" (zero) = No logical name permitted.
"G" = A GeoStan logical name required.
"S" = A Spatial+ logical name required.
"D" = A Demographics Library logical name required.
"C" = A GeoStan Canada logical name required.
"L" = A GDL logical name required.

`AB_FIELD_NUM_VALUES (size = 12)`
Maximum number of unique values possible for field.

Example

```
char length[13];
char datatype[2];
int len;

QABValidateProperties(ab);
QABGetFieldAttribute ( ab, "City", AB_FIELD_INPUT, AB_FIELD_LENGTH,
length,13 );
len = atoi (length);

QABGetFieldAttribute ( ab, "PolygonName", AB_FIELD_OUTPUT,
AB_FIELD_DATA_TYPE, datatype, 2 );

printf("City field length: %i\n", len);
printf("PolygonName datatype:%s\n", datatype);
```

See Also

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information about fields.

QABGetPropertyID

Retrieves a property value.

Syntax

```
Boolean QABGetPropertyID ( ABId ab,
   unsigned long in_usPropID,
   char* out_pszPropValue,
   unsigned long in_usBufferSize )
```

Arguments

ab — The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.

in_usPropID — A valid property symbolic constant. *Input*.

out_pszPropValue — Pointer to the property value to retrieve. All values are returned as
strings. *Output*.

in_usBufferSize — The size of the string buffer. *Input.*

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

None.

Alternates

QABInit

Example

```
char        buffer [ AB_MAX_FIELD_VALUE ];
char*       szInitlist;
Boolean     bMixedCase;
int         len;

QABGetPropertyID( ab, AB_MIXED_CASE, buffer, AB_MAX_FIELD_VALUE );
```

```
bMixedCase = atoi(buffer);

QABGetPropertyID ( ab, AB_INIT_LIST, szInitlist, len );
...
free ( szInitlist );
```

See Also

See Chapter 13, "Properties" for more information about properties.

QABGetPropertyStr

Retrieves a property value.

Syntax

```
Boolean QABGetPropertyStr ( ABId ab,
   char* in_pszPropName,
   char* out_pszPropValue,
   unsigned long in_usBufferSize )
```

Arguments

ab	The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.
in_pszPropName	A valid property name. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.
out_pszPropValue	Pointer to the property value to retrieve. All values are returned as strings. *Output*.
in_usBufferSize	The size of the string buffer. *Input.*

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

None.

Alternates

None.

Notes

The **QABGetPropertyID** function provides slightly better performance.

Example

```
char        buffer [ AB_MAX_FIELD_VALUE ];
char*       szInitlist;
Boolean     bMixedCase;
int         len;

QABGetPropertyAttributeStr( ab, "INIT_LIST", AB_PROPERTY_LENGTH,
buffer, AB_MAX_FIELD_VALUE );
// len will include space for the trailing nul
len = atoi (buffer);
szInitlist = malloc ( len );
```

See Also

See Chapter 13, "Properties" for more information about properties.

QABGetPropertyAttribute*

Retrieves a property attribute.

Syntax

```
Boolean QABGetPropertyAttributeStr ( ABId ab,
   char* in_pszPropName,
   unsigned long in_usAttributeName,
   char* out_pszAttributeValue,
   unsigned long in_usBufferSize )
Boolean QABGetPropertyAttributeID ( ABId ab,
   unsigned long in_usPropID,
   unsigned long in_usAttributeName,
   char* out_pszAttributeValue,
   unsigned long in_usBufferSize )
```

Arguments

ab
The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.

in_pszPropName
A valid property name. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

in_usAttributeName
A symbolic constant of the attribute to retrieve. *Input*.

out_pszAttributeValue
Pointer to the attribute value (string) to be loaded. *Output*.

in_usBufferSize	The size of the string buffer. *Input.*
in_usPropID	A valid property symbolic constant. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

`QABSetPropertyID`

`QABSetPropertyStr` for client property information

Alternates

None.

Notes

The `QABGetPropertyAttributeID` function provides slightly better performance. To receive information about properties set on the server, call the function before setting any properties in the client code. To receive information about client properties, call `getPropertyAttribute` after calling `QABSetPropertyID` or `QABSetPropertyStr`.

Attribute Values

`AB_PROPERTY_DATA_TYPE (size = 2)`
"N" (integer), "B" (Boolean), or "C" (string).

`AB_PROPERTY_DEFAULT_VALUE`
Default property value. The size of `AB_PROPERTY_DEFAULT_VALUE` is determined by the value assigned to `AB_PROPERTY_LENGTH`.

`AB_PROPERTY_DESCRIPTION (size = 101)`
Short (100-character) description of property.

`AB_PROPERTY_ID (size = 12)`
Property ID.

`AB_PROPERTY_LENGTH (size = 12)`
Length of property value.

`AB_PROPERTY_NAME (size = 33)`
Property name.

`AB_PROPERTY_READ_ONLY(size = 2)`

"1" property is read-only.
"0" property is read/write.

Example 1

```
char datatype[2];
char length[13];

QABGetPropertyAttributeStr ( ab, "MIXED CASE", AB_PROPERTY_DATA_TYPE,
datatype, 2);
QABGetPropertyAttributeStr ( ab, "INIT_LIST", AB_PROPERTY_LENGTH,
length, 13);
```

Example 2

```
char datatype[2];
char length[13];

QABGetPropertyAttributeID ( ab, AB_MIXED_CASE, AB_PROPERTY_DATA_TYPE,
datatype, 2);
QABGetPropertyAttributeID ( ab, AB_INIT_LIST, AB_PROPERTY_LENGTH,
length, 13);
```

See Also

See Chapter 13, "Properties" for more information about properties.

QABGetRecord

Advances the pointer to the next record in the output record buffer.

Syntax

```
Boolean QABGetRecord ( ABId ab )
```

Arguments

ab	The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful or no further records found.

Prerequisites

`QABProcessRecords`

Alternates

None.

Notes

The first call to **`QABGetRecord`** sets a pointer in the output record buffer to the first output record. Subsequent calls to **`QABGetRecord`** advance the pointer. When no further records are found, **FALSE** is returned. Use the QABGetField functions to retrieve values from individual record fields. Use the QABResetField function to reset the output record pointer to the first output record.

Example

```
char addrln[61];
while ( QABGetRecord (ab ))
{
/* get field value*/
QABGetField ( ab, "AddressLine", addrln, 61 )
...
}
...
```

QABGetStatus

Returns status or error codes and messages.

Syntax

```
Boolean QABGetStatus ( ABId ab,
   unsigned long* out_ulStatus,
   char* out_pszStatusMsg,
   unsigned long in_ulBufferSize )
```

Arguments

ab	The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.
out_ulStatus	Status or error code returned. *Output*.
out_pszStatusMsg	Status or error message returned. *Output*.
in_ulBufferSize	The size of the string buffer. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

`QABInit`

Alternates

None.

Notes

Generally, a 2048-character buffer is sufficient, although the actual message size varies.

QABInit

Creates instances of **QMSAddressBroker** subclasses.

Syntax

```
ABId QABInit (
   unsigned long* in_usConnect,
   char* in_pszHostName,
   char* in_pszTransport,
   char* in_pszUser,
   char* in_pszPassword,
   char* in_pszInitFileName )
```

Arguments

in_usConnect Specifies connection type (AB_CLIENT). *Input*.

in_pszHostName A delimited list. *Input*.

in_pszTransport Case-insensitive string that specifies the network protocol AddressBroker uses. *Input*.

in_pszUser A valid user name. *Input*.

in_pszPassword A valid user's password. *Input*.

in_pszInitFileName
Specify an initialization file name or **NULL**. *Input*.

Return Values

A handle to a broker instance if successfully created, **NULL** if unsuccessful.

Prerequisites

None.

Alternates

None.

Notes

The created object is initialized and default properties are set.

The client transparently switches between servers if it has a problem establishing communication with its current server. That is, when the client executes a command that includes a server transaction, it switches servers if there is no response from the current server or a transaction fails.

An AddressBroker client uses the first server specified in *in_pszHostName* until the server fails, at which point it switches to the next server listed in *in_pszHostName*. The client continues to use this secondary server until it—the secondary server—fails. After a failed server is operational, it again becomes available to the client. However, the client does not switch back unless its current server fails. When a client searches for a server and encounters the end of *in_pszHostName* it continues searching from the beginning of the list.

On a per-transaction basis, the client tries each server in turn until it finds an operational server. If it fails to find a server, the operation fails.

When listing multiple servers, it is extremely important that they all service client requests identically. To ensure predictable results, make sure that the server .ini files on each host use the same initialization settings.

in_pszInitFileName optionally specifies an input file containing property settings and keyword commands.

Values set in the input file override any default property settings. Subsequent calls to the QABSetPropertyID and QABSetPropertyStr functions override property values found in the file.

Example 1

```
// Socket protocol using machine name
ab = QABInit ( AB_CLIENT, “primary:1234 | secondary:1235”, “socket”,
“MyLogon”, “MyPassword”, “MyInitFile” );
```

Example 2

```
// Socket protocol using URL
ab = QABInit ( AB_CLIENT, “centrus.com:1234 | centrus-
software.com:1235”, “socket”, “MyLogon”, “MyPassword”, “MyInitFile” )
;
```

Example 3

```
// Socket protocol using IP address
ab = QABInit ( AB_CLIENT, “204.180.129.200:1234 | 209.38.36.44:1235”,
“socket”, “MyLogon”, “MyPassword”, “MyInitFile” ) ;
```

QABLookupRecord

Processes a single incomplete U.S. address record or performs a reverse lookup on a Canadian postal code.

Syntax

```
int QABLookupRecord ( ABId ab )
```

Arguments

ab	The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.

Return Values

The `OUTPUT_FIELD_LIST` property defines the fields populated by **`QABLookupRecord`**, and the return codes listed below describe the search outcome. Codes are returned only when the relevant fields are included in `OUTPUT_FIELD_LIST`. A return value of zero (**0**) indicates an internal failure.

Return Codes

AB_LOOKUP_ADDRESS_LINE_INCOMPLETE

For a U.S. address, the firm name or unit number could not be resolved. Multiple incomplete records were returned. The user can be prompted to submit more information. The most useful fields for resolving a match generally are `FirmName`, `HighUnitNumber`, `LowUnitNumber`, `MatchCode`, and `UnitType`.

Other helpful fields include `AddressLine`, `AddressLine2`, `CarrierRoute`, `CountyName`, `FIPSCountyCode`, `GovernmentBuildingIndicator`, `HighEndHouseNumber`, `LACSAddress`, `LastLine`, `LowEndHouseNumber`, `PostfixDirection`, `PrefixDirection`, `RoadClassCode`, `SegmentBlockLeft`, `SegmentBlockRight`, `State`, `UrbanizationName`, `USPSRangeRecordType`, `ZIP`, `ZIPCarrtSort`, `ZIPCityDelivery`, `ZIPClass`, `ZIPFacility`, and `ZIPUnique`.

For a Canadian postal code, the input Postal Code is resolved to a range of possible addresses that contain a single street number. The street number suffix or unit number values will vary over the range.

AB_LOOKUP_LAST_LINE_NOT_FOUND

For a U.S. address, multiple incomplete records returned. Did not resolve LastLine. Use iterative calls to QABGetRecord to retrieve possible matches. Only the following output fields are returned: `MatchCode`, `CITY`, `State`, `ZIP`, and `ZIPFacility`. For a Canadian postal code, this return code indicates that the input postal code was not found in the CPC data and is invalid.

AB_LOOKUP_MULTIPLE_MATCH
For a U.S. address, the address resolved to a multiple match. Multiple complete address records were matched and returned. Use iterative calls to QABGetRecord to retrieve possible matches. For a Canadian postal code, the postal code resolved to a range of possible addresses that vary over the street.

AB_LOOKUP_NOT_FOUND
The address could not resolve to a match or possible match. No records returned. Provide a more complete address. (This return code is not used for Canada.)

AB_LOOKUP_SUCCESS
For a U.S. address, a complete single address was matched and returned. For a Canadian postal code, a single address was matched and returned.

AB_LOOKUP_TOO_MANY_CITIES
No records returned. An incomplete LastLine matched over 100 cities. Provide a more complete city name. (This return code is not used for Canada.)

Prerequisites

None.

Alternates

`QABSetRecord`

Notes

`QABLookupRecord` processes a single input record and should be used only when address information is insufficient for standardization. To process single or multiple records containing complete addresses, use QABProcessRecords.

Minimally, address information for **`QABLookupRecord`** must include a street number, a partial street name, and/or valid LastLine information. For Canada, a valid postal code is required and will return a single address or a range of addresses.

`QABLookupRecord` is most useful in interactive programs, when an application may have to make several calls to **`QABLookupRecord`** in order to find a match for an incomplete address. In client/server and Internet environments, the record is transferred across the network with each call to `QABLookupRecord`. The function call does not return until the record is processed. When `QABLookupRecord` processes an address record and fails to find an exact match, it does an extensive search to find cities and streets that are possible matches.

The `INPUT_FIELD_LIST` property specifies the list of fields passed to `QABLookupRecord`. Generally, provide at least `FirmName`, `AddressLine` and `LastLine` fields as input to `QABLookupRecord`. For Canada, a valid Canadian Postal Code is the only input, and it is set using the PostalCode input field. Only one Postal Code can be processed at a time.

The `OUTPUT_FIELD_LIST` property specifies the list of possible fields returned.

The `MAXIMUM_LOOKUPS` property limits the number of multiples—possible matches—that are returned by `QABLookupRecord`. The upper limit of `MAXIMUM_LOOKUPS` is 100. For a Canadian postal code, if the `MAXIMUM_LOOKUPS` is set to 100, the AddressBroker software increases the `MAXIMUM_LOOKUPS` to 200.

Retrieve the list of possible matches using a ‘`while (QABGetRecord) do QABGetField`’ loop. No records are returned when the return value of `QABLookupRecord` is **AB_LOOKUP_NOT_FOUND** or **AB_LOOKUP_TOO_MANY_CITIES**.

Precisely recommends using `QABProcessRecords` instead of `QABLookupRecord`.

Example

In an interactive application, a user submits a partial address to `QABLookupRecord`. The return code is **AB_LOOKUP_LAST_LINE_NOT_FOUND**. For a U.S. address, this code indicates that the user did not enter enough information for `QABLookupRecord` to resolve the city, state, or ZIP Code. The application can then prompt the user to select from the list of possible cities and states returned by `QABLookupRecord`. The user selects the necessary information and resubmits the address to `QABLookupRecord`. For a Canadian postal code, this return code indicates that the input postal code was not found in the CPC data and is invalid.

This time the return code is **AB_LOOKUP_ADDRESS_LINE_INCOMPLETE**. The user resolved the last line problem, but the return code indicates the address line could be more specific. For a U.S. address, it is missing information on the firm name or unit number (suite, apartment, etc.). The application can prompt the user to select from the list of possibilities returned by this call to `QABLookupRecord`. The user enters the additional information and resubmits the address to `QABLookupRecord`, and **AB_LOOKUP_SUCCESS** is returned. For a Canadian postal code, the **AB_LOOKUP_ADDRESS_LINE_INCOMPLETE** code indicates that the input Postal Code resolved to a range of possible addresses that contain a single street number. The street number suffix or unit number values will vary over the range. For example, a Canadian postal code of T3C 2K7 could resolve to 123 A - 123 G Maple Street (when the street suffix varies) or 123 Maple Street Unit 1-100 (when the unit number changes). A valid postal code for one address submitted to `lookupRecord` returns **AB_LOOKUP_SUCCESS**.

When the next address is entered, `QABLookupRecord` returns the status code **AB_LOOKUP_MULTIPLE_MATCH.** This indicates multiple complete matches were found. For a U.S. address, the user may then be prompted to select from the list of possible matches. The selected address is resubmitted to `QABLookupRecord` to ensure that it is entirely correct and that **AB_LOOKUP_SUCCESS** is returned. For a Canadian postal code, the **AB_LOOKUP_MULTIPLE_MATCH** code indicates a postal code that resolved to a range of possible addresses that vary over the street. For example, a Canadian postal code could resolve to 100-120 Elm, Calgary, AB or 150-165 Maple, Calgary, AB.

QABProcessRecords

Processes a set of one or more address records.

Syntax

`Boolean QABProcessRecords ( ABId ab)`

Arguments

ab	The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.

Return Values

Returns **TRUE (1)** if successful, **FALSE (0)** if unsuccessful.

Prerequisites

None.

Alternates

`QABSetRecord`

Notes

Each record should contain enough address information for standardization. For records containing incomplete addresses, use QABLookupRecord, which progressively returns address choices for one input record at a time.

The function call does not return until all of the records are processed.

The `MATCH_MODE` property controls the “closeness” of the matched records. Set `MATCH_MODE` to `AB_MODE_CLOSE` for best results. See “Pre-defined property values” on page 354 for more information.

The KEEP_MULTIMATCH property specifies whether a single match or multiple matches are returned. The RecordID input and output fields help correlate input records with their corresponding output record(s).

The KEEP_COUNTS property specifies whether match criteria counts are kept. To keep counts, set KEEP_MULTIMATCH to FALSE and set KEEP_COUNTS to TRUE. Keeping counts increases processing time.

The INPUT_FIELD_LIST property specifies the list of record fields that is given to **QABProcessRecords**. The OUTPUT_FIELD_LIST property specifies the list of field names that **QABProcessRecords** can return.

See Also

See Chapter 13, "Properties" for more information about properties.

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information about fields.

QABResetField

Resets the output field pointer to the first value of an output field.

Syntax

```
QABResetField ( ABId ab,
char* in_pszFieldName )
```

Arguments

ab	The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.
in_pszFieldName	A valid field name listed in the OUTPUT_FIELD_LIST property. Some field names require a logical name. The logical name must be appended to *in_pszFieldName* in brackets. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

QABGetField

Alternates

None.

Notes

`QABResetField` returns **FALSE** when, for any reason, *in_pszFieldName* is not found.

All Spatial+, GDL, and Demographic fields require logical names. GeoStan and GeoStan Canada fields do not.

Example

```
/* Example showing logical name appended in brackets. */
while ( QABGetField ( ab, "PolygonName[COUNTIES]", polygonname, 128 ))
{
    ...
}
QABResetField ( ab, "PolygonName[COUNTIES]" );
```

See Also

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information about fields.

QABResetRecord

Resets output record pointer to the first record in the output record buffer.

Syntax

```
Boolean QABResetRecord ( ABId ab )
```

Arguments

ab	The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

`QABGetField`

Alternates

None.

QABSetField

Sets an input field value in the current input record.

Syntax

```
Boolean QABSetField ( ABId ab,
   char* in_pszFieldName,
   char* in_pszFieldValue )
```

Arguments

ab	The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.
in_pszFieldName	A valid field name listed in the `INPUT_FIELD_LIST` property. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.
in_pszFieldValue	The string value to assign to the field. Maximum string length is 256 characters. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

`QABSetProperty`

Alternates

None.

Notes

The `RECORD_DELIMITER`, `FIELD_DELIMITER`, and `VALUE_DELIMITER` properties have default values of line feed, tab, and CTRL-A, respectively. If your data contains any of these characters, you *must* reset the appropriate property to a different character. In addition, your data may not contain the null character.

This function sets the fields of the current input record only. **`QABSetRecord`** must be called *after* each input record.

Example

```
/*Assumes a record consists of addressline and lastline
  only. */
char addressline [61];
char lastline[61];
while ( more_data )
{
QABSetField ( ab, "AddressLine", addressline );
QABSetField ( ab, "LastLine", lastline );
QABSetRecord ( ab );
}
```

See Also

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information about fields.

QABSetLogFn

Call back function for handling error messages.

Syntax

```
Boolean QABSetLogFn (
   void ( * in_pLogFn )(
   QMSABStatusType type,
   const char * message ))
```

Arguments

in_pLogFn	A user-provided routine to handle error messages. *Input*.

Return Values

TRUE if successful. **FALSE** if the log file was not set.

Prerequisites

None.

Alternates

None.

Notes

`QABSetLogFn` takes a user-provided function as its argument. This function in turn takes two arguments—an enumeration of **`QMSABStatusType`** and a message buffer (see Example below).

Status Types

`ABSTATUS_NONE`	Report no status messages.
`ABSTATUS_FATAL`	Report warning, errors, and fatal errors.
`ABSTATUS_ERROR`	Report warnings and errors only.
`ABSTATUS_WARN`	Report warning messages only.

`ABSTATUS_INFO`

Report status messages.

`ABSTATUS_DEBUG`

Report status messages, development only.

Example

```
/* Here is the error handling routine the user provides us. */
void MyErrorHandler ( QMSABStatusType type, const char * message)
{
...
printf("%s\n", message);
}
QABSetLogFn (MyErrorHandler);
```

QABSetPropertyID

Assigns a property value.

Syntax

```
Boolean QABSetPropertyID ( ABId ab,
   unsigned long in_usPropID,
    const char* in_pszPropValue )
```

Arguments

ab	The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.
in_usPropID	The valid symbolic constant of the property being set. *Input*.
in_pszPropValue	A string value to assign to the property. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

`QABInit`

Alternates

None.

Notes

The enumerated constants `AB_*` available in `abtypes.h` may be passed, as a string, as input to **`QABSetPropertyStr`** by dropping the `AB_`. For example, `AB_INPUT_PARSED` may be passed as "`INPUT_PARSED`".

Example

```
QABSetPropertyID( ab, "MIXED CASE", "TRUE" );
QABSetPropertyID( ab, "INIT_LIST", "GEOSTAN | GEOSTAN_Z9" );
QABSetPropertyID( ab, "INPUT_MODE", "INPUT_PARSED" );
```

See Also

See Chapter 13, "Properties" for more information about properties.

QABSetPropertyStr

Assigns a property value.

Syntax

```
Boolean QABSetPropertyStr ( ABId ab,
   const char* in_pszPropName,
   const char* in_pszPropValue )
```

Arguments

ab	The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.
in_pszPropName	A valid property name. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.
in_pszPropValue	A string value to assign to the property. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

`QABInit`

Alternates

None.

Notes

The enumerated constants `AB_*` available in `abtypes.h` may be passed, as a string, as input to **`QABSetPropertyStr`** by dropping the `AB_`. For example, `AB_INPUT_PARSED` may be passed as “`INPUT_PARSED`”.

Example

```
QABSetPropertyStr ( ab, "MIXED CASE", "TRUE" );
QABSetPropertyStr ( ab, "INIT_LIST", "GEOSTAN |
GEOSTAN_Z9" );
QABSetPropertyStr ( ab, "INPUT_MODE", "INPUT_PARSED" );
```

See Also

See Chapter 13, "Properties" for more information about properties.

QABSetRecord

Adds data for the current record to the input record buffer and advances the input record pointer to the next empty record in the buffer.

Syntax

```
Boolean QABSetRecord ( ABId ab )
```

Arguments

ab	The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if there is no current record.

Prerequisites

`QABSetField`

Alternates

None.

QABTerm

Destroys QMSAddressBroker instance.

Syntax

```
Boolean QABTerm ( ABId ab )
```

Arguments

ab	The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

`QABInit`

Alternates

None.

Notes

On UNIX machines, failure to call this command may result in continued system resource consumption. See “System resources and AddressBroker UNIX servers” on page 89 for more information.

QABValidateProperties

Validates properties for consistency and completeness.

Syntax

```
Boolean QABValidateProperties ( ABId ab )
```

Arguments

ab	The ID returned by a call to QABInit for the current AddressBroker handle. *Input*.

Return Values

TRUE (1) if all properties are valid, **FALSE (0)** if one or more properties fail to validate.

Prerequisites

`QABSetProperty`

Alternates

None.

Notes

The `QABValidateProperties` function verifies the values of initialization and processing control properties to ensure a complete and compatible set of values are available to AddressBroker. Call this function after one or more properties have been set and before calling QABSetField or any processing functions.

When `QABValidateProperties` returns **TRUE**, it indicates all properties have been successfully validated and that AddressBroker is ready to process records. In some cases, all properties can be validated in a single function call.

See Also

See Chapter 13, "Properties" for more information about properties.

Errors, messages, and status logs

The AddressBroker C API supports two independent methods of error handling:

- The use of a log file—assigned to AddressBroker's STATUS_LOG property—used in conjunction with a reporting threshold, assigned to AddressBroker's STATUS_LEVEL property.
- The THROW_LEVEL property can be used to cause your application to abort upon error.

These two error handling methods are discussed in this section. See "GeoStan location codes" on page 433 for a discussion about the codes themselves.

Using STATUS_LOG and STATUS_LEVEL

The STATUS_LOG and STATUS_LEVEL properties do not require validation to be used or changed. The STATUS_LOG property is set by the server administrator, rather than by the client programmer.

STATUS_LOG holds the output destination of all reported messages and contains general server events. Set AddressBroker's STATUS_LOG property to one of the following:

- The path and file name for a status log to save status messages.

- The value CONSOLE to display status messages to a console window.

Set AddressBroker's STATUS_LEVEL property to the appropriate level of message reporting you require:

- NONE—No messages. The least verbose.
- FATAL—Fatal errors, errors, and warnings.
- ERROR—Errors and warnings only.
- WARN—Warnings only.
- INFO—All information messages.
- DEBUG—Debug messages; for development only.
- SERVER—Server level only messages. Default.

The LOG_ROLLOVER property sets age and size criteria for the status and request log files for the periodic rollover of file names. This property ensures that the log file does not become too large or too old to be useful. The log files that are rolled over include the STATUS_LOG and REQUEST_LOG.

Using REQUEST_LOG

The REQUEST_LOG property specifies a log file that contains a final summary of each request (client interaction with the server). For each request, the following information will be supplied:

- Request type.
- Request ID.
- Creation time.
- Client IP.
- Logical names used by the client.
- Username.
- Server handle number that processed the request.
- Number of records processed.
- Elapsed seconds on request queue, elapsed seconds being processed.
- Total seconds in the server.

```
The REQUEST_LOG property is set by the server administrator, rather than
by the client programmer. Following is a sample REQUEST_LOG file:

Request type: Initialize. Request# 1. Create time: Wed May 26 13:54:50
2004. Client IP: 175.18.2.76.
Logical Names: GEOSTAN|GEOSTAN_Z9|CENSUS2K. User Name: .
Handle# 0. Num Records: 0. Elapsed seconds on queue: 0. Elapsed seconds
in processing: 0.
Total seconds in server: 0.
```

Using THROW_LEVEL

AddressBroker's THROW_LEVEL property determines the level at which your application is notified of an error or status condition. Even though the C programming language does not support the use of try-throw-catch routines (for which this property was included), you can still make use of it in your application as described here.

In the C API, THROW_LEVEL is automatically reset to NONE when an object initializes, thereby disabling this mechanism of error handling.

Note: Setting THROW_LEVEL to any other value causes your application to abort if a status condition meets or exceeds its value. THROW_LEVEL does not require validation to be used or changed.

Legal values for THROW_LEVEL are as follows:

- FATAL—fatal errors, errors, and warnings.
- ERROR—errors and warnings only. Default.
- WARN—warnings only.
- INFO—all information messages.
- NONE—no messages.
- DEBUG—debug messages; for development only.

11 – C++ API

In this chapter

This chapter describes the C++ API to AddressBroker in detail. For general information on AddressBroker, see Chapters 1, 2, and 4 of this manual.

This chapter provides a tutorial using the AddressBroker C++ API. The tutorial shows you how to use most of AddressBroker's functionality, yet is general enough that you can modify it for other uses. A complete member function reference follows the tutorial. The final section of this chapter discusses error handling.

The naming convention for AddressBroker C++ API functions is **`FunctionName`**. All C++ functions use this naming convention.

Accessing the AddressBroker C++ libraries

To use the AddressBroker library in a client application, you must include the appropriate header file in your application source code files:

```
#include "ABbase.h"    // C++ API
```

You must also use the appropriate syntax for creating an AddressBroker handle or instance:

```
// C++ API
QMSAddressBroker *ab = QMSAddressBroker::CreateClient ("hostname:4660",
"SOCKET", MyLogon, MyPassword,
"myinit.ini");
```

Finally, you must include the AddressBroker import library in the link step of your build.

Windows platforms

Link to the `AB.lib` import library, which causes your application to use `AB.dll`. For your application to execute properly, this DLL must be found in your execution `PATH` environment variable.

UNIX platforms

Link to `libab.sl` or `libab.so`, which causes your application to dynamically bind to the AddressBroker library. For your application to execute properly, this shared library must be found in your shared library path environment variable: `SHLIB_PATH` for HP-UX, or `LD_LIBRARY_PATH` for most other UNIX systems.

Note: To process Canadian addresses, `NCODEDATA` and `LD_LIBRARY_PATH` for Solaris, or `SHLIB_PATH` for HP-UX must be set. See the *GeoStan Canada Reference Manual* for more information.

AddressBroker C++ tutorial

This section describes the steps necessary to develop an AddressBroker application using the C++ API. The example shows basic C++ sample code that does address record enhancement. The sample uses the `FirmName`, `AddressLine`, and `LastLine` fields from Precisely address records as input. The tutorial standardizes the address data and augments it with city, state, and 9-digit ZIP Code information from the GeoStan data directory. Then, it retrieves the name and status of the geographic polygon where the address is located using a Spatial+ data file.

Sample C++ code (`console.cpp`) is located in the Samples subdirectory.

Step 1: Create and initialize the object

To begin, link your application to the AddressBroker import library. Your application must include the “`ABbase.h`” header file, which defines AddressBroker C++ class definitions. This header file also includes “`ABtypes.h`”, which defines AddressBroker data types. You do not need to include “`ABtypes.h`” in your source code.

The "`console.cpp`" includes its header file, "`console.hpp`", which includes the AddressBroker client header file:

```
#include "abbase.h"
```

You can initialize your application from a client initialization file, or you can initialize the client programmatically without the use of an initialization file. To use the supplied initialization file, “`abconsole.ini`", uncomment the line that establishes that name in the application:

```
//initfile = filename;
```

This action causes the initialization of the AddressBroker client object to use `abconsole.ini` to establish client properties, such as `INIT_LIST`, `INPUT_MODE`, `INPUT_FIELD_LIST`, `OUTPUT_FIELD_LIST`, and others. If you leave it commented, the application will instead invoke a function called **`ABConsoleTest::InitProperties()`** to perform the initialization programmatically after the AddressBroker client object has been created:

```
if( !initfile)
{
    // set the necessary properties.
    InitProperties();
}
```

The hostname and port are hard coded to "`localhost:4660`". You can override this in two ways. The first way is to change the following line of code:

```
char host[128] = "localhost:4660";
```

The second way to override it is to invoke the program and supply the name of the host and port on the command line. The program reads the command line and uses your supplied parameters:

```
if(argc > 1) {
    strcpy( host, argv[1] );
}
```

The hostname is the name of the machine which is running the AddressBroker server, and the port is the port that the server is listening on for new client connections. If your AddressBroker server requires a logon user ID and password, you can set them in the program by changing the following lines:

```
char* logon     = NULL;
char* password  = NULL;
```

You now have enough information to create the AddressBroker client object, which is accomplished with the following line:

```
broker = QMSAddressBroker::CreateClient( hostname, socket", logon,
password, initfile );
```

The program traps possible errors generated by a failed client initialization with the following code:

```
broker->GetStatus( status_code, status_msg, sizeof( status_msg ) );
if( status_code )
{
  throw( status_msg );
}
```

Step 2: Set properties

You assign a minimal set of properties in your client application. For a detailed discussion about client applications, see Chapter 5, "Client Applications".

Logical names and paths are set on the server. The logical names the client uses must match those set on the server. In the sample code shown in "C++ SetProperty example" on page 234, the logical names `GEOSTAN`, `GEOSTAN_Z9`, and `COUNTIES` refer to a GeoStan data directory, a GEOSTAN ZIP Code file, and a Spatial+ polygon file. Next, tell AddressBroker to use the `FirmName`, `AddressLine`, and `LastLine` field values from each input record. In this example, the `FirmName` and `AddressLine` fields are enhanced with City, State, and ZIP10 information from the GEOSTAN data file. `PolygonName` and `PolygonStatus` are also retrieved from the COUNTIES file.

You can set other properties in the client. In the sample code, `KEEP_MULTIMATCH` and `BUFFER_RADIUS` are set. See Chapter 13, "Properties" for a detailed discussion.

C++ property reference syntax

```
//setting a property using its string name
broker->SetProperty("INIT_LIST", "GEOSTAN|GEOSTAN_Z9" );
```

```
//setting a property using its property ID
broker->SetProperty(AB_INIT_LIST, "GEOSTAN|GEOSTAN_Z9" );

//setting a pre-defined property
broker->SetProperty("INPUT_MODE", AB_INPUT_NORMAL);
```

C++ SetProperty example

If you choose to initialize the application properties programmatically (rather than through an initialization file as described earlier in this example), you make calls to the **`SetProperty`** method of the **`QMSAddressBroker`** object.

The `INIT_LIST` property provides the logical names that AddressBroker will use. In the following example, a generic logical name for GeoStan is used. Add others to the pipe-delimited list for other processing:

```
broker->SetProperty("INIT_LIST", "GEOSTAN|GEOSTAN_Z9" );
```

The `INPUT_FIELD_LIST` provides the specific input fields to use. This is done only once in the example; it is a dynamic property and you can set it at any time and as many times as you want.

```
broker->SetProperty("INPUT_FIELD_LIST",
"FirmName|AddressLine|LastLine");
```

The `OUTPUT_FIELD_LIST` defines the output fields to be returned:

```
broker->SetProperty("OUTPUT_FIELD_LIST",
"FirmName|AddressLine|City|State|Zip10|MatchCode"
"|Longitude|Latitude|Location Quality Code" );
```

You can set other properties that affect server behavior, such as instructing the server to keep only one output record for each input record and to use a coordinate type when returning geocodes:

```
broker->SetProperty( "KEEP_MULTIMATCH", (Boolean)false );
broker->SetProperty("Coordinate Type", AB_COORD_FLOAT );
```

Step 3: Validate properties (optional)

Use the ValidateProperties function to send the property definitions to AddressBroker for validation. When **`ValidateProperties`** returns **TRUE**, the AddressBroker client object properties are set correctly and are ready for processing. If any property setting is invalid, an error is generated. You can use GetStatus to retrieve error messages in the event **`ValidateProperties`** does not return successfully.

All AddressBroker properties must be set and validated before data can be input or processed. In client mode, calling this function results in a server transaction.

C++ ValidateProperties example

```
if( !broker->ValidateProperties() )
{
  UInt32 status_code;
  char status_msg[2048];
  broker->GetStatus( status_code, status_msg, sizeof(
 status_msg ) );
  throw( status_msg );
}
```

ValidateProperties can be called multiple times in your application. For example, you can initially set and validate a group of properties, then allow the end user to dynamically select new values and revalidate the settings.

Step 4: Enter input records and field values

Use the SetField function call to specify the input field values. Note that these are the same fields you specified initially with the `SetProperty` function call (see the code example above).

The SetRecord function call adds the data for the current record to the input record list and advances the record pointer.

You do not need to set an input value for every field in a record. In our example, an individual record that did not contain `FirmName` information could still be processed.

C++ Data input example

```
// build a few records for enhancement
// Fill in a record...
broker->SetField("FirmName", "Precisely");
broker->SetField("AddressLine", "4750 Walnut #200");
broker->SetField("LastLine", "Boulder, CO");

// SetRecord can fail (but only if SetField is never called)
broker->SetRecord();

// Fill in a second record...
broker->SetField("FirmName", "White House");
broker->SetField("AddressLine", "1600 Pennsylvania");
broker->SetField("LastLine", "Washington, DC");
broker->SetRecord();
```

Step 5: Process records

After all the input data has been entered, you are ready to process the records. Use the ProcessRecords function to process records. In client mode, this sends all the data to the server for processing. A return value of true indicates success and false indicates failure. The example below deals with a failed call to **`ProcessRecords`**.

Note: This function call clears the input record buffer, even if the call fails.

C++ record processing example

```
if( !broker->ProcessRecords() )
{
  UInt32 status_code;
  char status_msg[2048];
  broker->GetStatus( status_code, status_msg, sizeof(
status_msg ) );
  throw( status_msg );
}
else
{
   ... ProcessRecords was successful. Return values may
       now be retrieved...
}
```

Step 6: Retrieve address records and field values

If **ProcessRecords** was successful (Step 5), use the **GetRecord** and **GetField** function calls to retrieve the output data.

In your C++ applications, loop through Steps 4 - 6 of this tutorial each time you process additional records. You can also repeat Steps 2 and 3 to modify property settings.

C++ data retrieval example

```
// field sizes are documented in manual
char firmname[41];
char addressline[61];
char city[29];
char state[3];
char zip10[11];
char matchCode[5];
char longitude[12];
char latitude[11];
char locCode[5];

// for each record that comes back
while( broker->GetRecord() )
{
  // get address data
  broker->GetField( "FirmName", firmname, sizeof( firmname ) );
  broker->GetField( "AddressLine", addressline, sizeof( addressline ) );
  broker->GetField( "City", city, sizeof( city ) );
  broker->GetField( "State", state, sizeof( state ) );
  broker->GetField( "ZIP10", zip10, sizeof( zip10 ) );
  broker->GetField( "MatchCode", matchCode, sizeof( matchCode ) );
  broker->GetField( "Longitude", longitude, sizeof( longitude ) );
  broker->GetField( "Latitude", latitude, sizeof( latitude ) );
  broker->GetField( "LocationQualityCode", locCode, sizeof( locCode ) );

  // print out the basic address
  cout << "Firm = " << firmname << endl;
  cout << "Address = " << addressline << endl;
  cout << "City = " << city << endl;
  cout << "State = " << state << endl;
  cout << "ZIP = " << zip10 << endl;
```

```
    cout << "Match Code = " << matchCode << endl;
    cout << "Longitude = " << longitude << endl;
    cout << "Latitude = " << latitude << endl;
    cout << "Location Quality Code = " << locCode << endl;

    //GetPolygonReturns( "COUNTIES" );
    cout << endl;
  }
```

AddressBroker C++ member functions

This section describes in detail the member functions available through the AddressBroker C++ API.

Some functions are listed as `FunctionName (overloaded)`. This indicates there are two or more functions with the same name whose behavior depends on the argument types it is given. For example, the same function name accepts either a Boolean type or a string type.

QMSAddressBroker classes

The `QMSAddressBroker` base class is never instantiated directly. Use one of the constructor methods to instantiate a client object.

The `QMSABStatus` class member functions manipulate the AddressBroker Exception Status object. To use this object, write your application to catch an exception object of this class.

The `QMSABLogFile` class lets you configure messaging to console or file.

Quick reference

QMSAddressBroker class member functions

Initialization member functions

createClient

Create and initialize instances of `QMSAddressBroker` subclasses. You must create an instance before calling any other AddressBroker function.

Property member functions

GetProperty (overloaded)

Retrieves the value of an input or output property.

GetPropertyAttribute (overloaded)

Retrieves a property attribute, such as its name, data type, and description.

SetProperty (overloaded)

Sets the value of a property.

ValidateProperties

Validates properties for consistency and completeness.This function must be called after `SetProperty` and before calls to `SetField`.

Field/data member functions

Clear

Clears the input and output record buffers and resets all counter properties to zero.

GetField (overloaded)

Retrieves the value(s) of an output field in the current output record. Call iteratively for fields that contain multiple values.

GetFieldAttribute

Retrieves a field attribute, such as its data type and description.

ResetField

Resets the output field pointer to the first value of an output field.

SetField

Sets an input field value in the current input record.

GetRecord

Retrieves the record and advances the output record pointer.

ResetRecord

Resets the output record pointer to the first record of the output record buffer.

SetRecord

Adds the data for the current record to the input record buffer and advances the input record pointer to the next empty record.

Processing member functions

ProcessRecords

Processes a set of one or more address records.

LookupRecord

Processes a single incomplete address record.

Reporting member functions

GetStatus

Retrieves status or error codes and messages.

Termination member functions

destroy

Destroys a `QMSAddressBroker` instance.

QMSABStatus class member functions

constructor (overloaded)

Creates and initializes instances of QMSABStatus class.

Message

Mechanism to retrieve a status message.

Status

Mechanism to retrieve a status type.

QMSABLogFile class member functions

constructor (overloaded)

Creates and initializes instances of QMSABLogFile class.

info, vinfo

Posts an info status message.

warn, vwarn

Posts a warning status message.

error, verror

Posts an error status message.

fatal, vfatal

Posts a fatal status message.

debug, vdebug

Posts a debug status message.

showStatus

Displays a status message.

SetLogFilePath

Sets the log file path.

GetLogFilePath

Retrieves the log file path.

EnableTermIO

Flag enabling terminal IO.

DisableTermIO

Flag disabling terminal IO.

UsingTermIO

Retrieves the status of terminal IO.

EnableEventLog

Flag enabling use of eventlog/syslog.

DisableEventLog

Flag disabling use of eventlog/syslog.

UsingEventLog

Retrieves the status of the use of eventlog/syslog.

SetLogProgramName

Retrieves the status of use of eventlog/syslog.

QMSAddressBroker class

Most of the functionality in AddressBroker's C++ API is encapsulated in the `QMSAddressBroker` class. The `QMSAddressBroker` base class is never instantiated directly. Use one of the constructor methods to instantiate a client object.

createClient

Create and initialize an instance of the `QMSAddressBroker` object.

Syntax

```
static QMSAddressBroker::CreateClient
( char* in_pszHostlist,
  char* in_pszTransport,
  char* in_pszUser,
  char* in_pszPassword,
  const char* in_pszInitFileName )
```

Arguments

in_pszHostlist	A delimited list of hosts where AddressBroker servers are running. *Input*.
in_pszTransport	Case-insensitive string that specifies the network protocol AddressBroker uses. Set this parameter to “socket”. *Input*.
in_pszUser	A valid user name. *Input*.
in_pszPassword	A valid user's password. *Input*.

in_pszInitFileName

Specifies the optional file name for property settings. *Input*.

Return Values

Returns a pointer to a new instance of the AddressBroker object if successfully created, **NULL** if unsuccessful.

Prerequisites

None.

Alternates

None.

Notes

The created object is initialized and default properties are set.

The client transparently switches between servers if it has a problem establishing communication with its current server. That is, when the client executes a command that includes a server transaction, it switches servers if there is no response from the current server or a transaction fails.

An AddressBroker client uses the first server specified in *in_pszHostlist* until the server fails, at which point it switches over to the next server listed in *in_pszHostlist*. The client continues to use this secondary server until it—the secondary server—fails. Once a failed server is operational, it again becomes available to the client. However, the client does not switch back unless its current server fails. When a client searches for a server and encounters the end of *in_pszHostlist*, it continues searching from the beginning of the list.

On a per-transaction basis, the client tries each server in turn until it finds an operational server. If it fails to find a server, the operation fails.

When listing multiple servers, it is extremely important that they all service client requests identically. To ensure predictable results, make sure that the server initialization files on each host use the same initialization settings.

in_pszInitFileName optionally specifies an input file containing property settings and keyword commands.

Values set in the input file override any default property settings. Subsequent calls to the `SetProperty` function override property values found in the file.

Example 1

```
// Socket protocol using machine name
QMSAddressBroker *ab = QMSAddressBroker::CreateClient (
"primary:1234 | secondary:1235", "socket", "MyLogon", "MyPassword",
"MyInitFile" ) ;
```

Example 2

```
// Socket protocol using URL
QMSAddressBroker *ab = QMSAddressBroker::CreateClient (
"centrus.com:1234 | centrus-software.com:1235", "socket", "MyLogon",
"MyPassword", "MyInitFile" ) ;
```

Example 3

```
// Socket protocol using IP address
QMSAddressBroker *ab = QMSAddressBroker::CreateClient (
"204.180.129.200:1234 | 209.38.36.44:1235", "socket",
"MyLogon", "MyPassword", "MyInitFile" ) ;
```

Backward Compatibility

In earlier releases of the AddressBroker product, users created objects of type **`QMSAddressBrokerLocal( )`** and **`QMSAddressBrokerClient( )`**. Backward compatibility of these objects is supported; to access new features, however, you must use the factory methods described in this section.

destroy

Destroys **`QMSAddressBroker`** instance.

Syntax

```
void QMSAddressBroker::Destroy(QMSAddressBroker * ab)
```

Arguments

ab	A pointer to an AddressBroker object.

Return Values

None.

Prerequisites

`QMSAddressBroker::create`

Alternates

None.

Example (Windows)

```
QMSAddressBroker *ab = QMSAddressBroker::CreateClient
(“primary:1234 | secondary:1235”, “socket”, “MyLogon”,
“MyPassword”, “MyInitFile” ) ;
...
QMSAddressBroker::Destroy(ab);
```

Clear

Clears input and output record buffers and resets counter properties.

Syntax

```
virtual Boolean QMSAddressBroker::Clear ( )
```

Arguments

None.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

None.

Alternates

None.

GetField (overloaded)

Retrieves output field value(s) from the current output record.

Syntax

```
virtual Boolean QMSAddressBroker::GetField (
   const char* in_pszFieldName,
   char* out_pszFieldValue,
   const unsigned long in_ulBufferSize )
virtual Boolean QMSAddressBroker::GetField (
   const char* in_pszFieldName,
   const char* in_pszLogicalName,
   char* out_pszFieldValue,
   const unsigned long in_ulBufferSize )
```

Arguments

in_pszFieldName
: A valid, fully specified field name listed in the OUTPUT_FIELD_LIST property. The property name is not case sensitive, and spaces and underscores are ignored (see the examples for this function). *Input*.

in_pszLogicalName
The logical name required by the value of *in_pszFieldName*. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

out_pszFieldValue
Pointer to the field value to be loaded. All values are returned as strings.
Output.

in_ulBufferSize The size of the string buffer. *Input*.

Return Values

Integer—**TRUE(1)** if a value for the field is found, **FALSE (0)** if unsuccessful or no values found.

Prerequisites

getRecord

Alternates

None.

Notes

The **GetField** function retrieves a field value from the current output record. Call **GetField** iteratively for multi-valued fields. Use the ResetField function to reset the field to its first value. To retrieve single value fields more than once, you must call **ResetField**.

All Spatial+, GDL, and Demographic fields require logical names. GeoStan and GeoStan Canada fields do not.

Example 1

```
/*Example using a field that does not require a logical name.*/
char city[29];
ab->GetField ( "City", city, 29 );
```

Example 2

```
/* Example using a multivalued field with its logical name in
brackets.*/
char polygonname[128];
while ( ab->GetField ( "PolygonName[COUNTIES]", polygonname, 128) )
{
...
}
```

Example 3

```
/* Example using a multivalued field with its
 logical name as separate argument.*/
while ( ab->GetField ( "PolygonName", "COUNTIES", polygonname, 128) )
{
...
}
```

See Also

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information on fields.

GetFieldAttribute

Retrieves information about AddressBroker fields.

Syntax

```
virtual Boolean QMSAddressBroker::GetFieldAttribute (
   const char* in_pszFieldName,
   const unsigned long in_ulFieldIOType,
   const unsigned long in_ulAttributeName,
   char* out_pszAttributeValue,
   const unsigned long in_ulBufferSize )
```

Arguments

in_pszFieldName A valid field name listed in the ALL_INPUT_FIELDS or ALL_OUTPUT_FIELD_LIST property. The property name is not case sensitive, and spaces and underscores are ignored. Do not associate logical names with field names when using this function. *Input*.

in_ulFieldIOType Indicates whether field name is an input field (AB_FIELD_INPUT) or an output field (AB_FIELD_OUTPUT). *Input*.

in_ulAttributeName
The symbolic constant for the attribute value to retrieve. *Input*.

out_pszAttributeValue
Pointer to the attribute value to be loaded. All values are returned as strings. *Output*.

in_ulBufferSize The size of the string buffer. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

`SetField`

Alternates

None.

Notes

`GetFieldAttribute` retrieves a field attribute's value. These are general attributes, not specific to a record.

`GetFieldAttribute` should only be called after **`ValidateProperties`**.

Attribute Values

`AB_FIELD_DATA_TYPE (size = 2)`
"N" (numeric), "C" (character).

`AB_FIELD_DECIMALS (size = 12)`
Number of decimal places, if numeric.

`AB_FIELD_DESCRIPTION (size = 33)`
Short (32-character) description of field.

`AB_FIELD_HELP (size = 256)`
Long (255-character) field description. This is not implemented for most fields.

`AB_FIELD_LENGTH (size = 12)`
Field width.

`AB_FIELD_NEEDS_LOGICAL_NAME (size = 2)`

"0" (zero) = No logical name permitted.
"G" = A GeoStan logical name required.
"S" = A Spatial+ logical name required.
"D" = A Demographics Library logical name required.
"C" = A GeoStan Canada logical name required.
"L" = A GDL logical name required.

`AB_FIELD_NUM_VALUES (size = 12)`
Maximum number of unique values possible for field.

Example

```
ab->ValidateProperties();
char length[13];
```

```
ab->GetFieldAttribute ( "City", AB_FIELD_INPUT, AB_FIELD_LENGTH,
length, 13);
int len = atoi (length);
char datatype[2];
ab->GetFieldAttribute ( "PolygonName", AB_FIELD_OUTPUT,
AB_FIELD_DATA_TYPE,
datatype, 2 );
```

See Also

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information on fields.

GetProperty (overloaded)

Retrieves a property value.

Syntax

```
virtual Boolean QMSAddressBroker::GetProperty (
   const char* in_pszPropName,
   Boolean & out_pbPropValue )
virtual Boolean QMSAddressBroker::GetProperty (
   const char* in_pszPropName,
   char* out_pszPropValue,
   const unsigned long in_ulBufferSize )
virtual Boolean QMSAddressBroker::GetProperty (
   const char* in_pszPropName,
   unsigned long & out_pulPropValue )
virtual Boolean QMSAddressBroker::GetProperty (
   const unsigned long in_ulPropID,
   Boolean & out_pbPropValue )
virtual Boolean QMSAddressBroker::GetProperty (
   const unsigned long in_ulPropID,
   char* out_pszPropValue,
   const unsigned long in_ulBufferSize )
virtual Boolean QMSAddressBroker::GetProperty (
   const unsigned long in_ulPropID,
   unsigned long & out_pulPropValue )
```

Arguments

in_pszPropName A valid property name. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

in_ulPropID A valid property symbolic constant. *Input*.

out_pbPropValue The location to store the returned Boolean data. *Output*.

out_pszPropValue

Pointer to the property value retrieved. All values are returned as strings.
Output.

out_pulPropValue
The location to store the returned long data. *Output*.

in_ulBufferSize The size of the string buffer. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

None.

Alternates

None.

Notes

The `GetProperty` functions that accept a property ID provide slightly better performance.

Example

```
char        buffer [ AB_MAX_FIELD_VALUE ];
char*       szInitlist;
Boolean     bMixedCase;
int         len;

ab->GetProperty ( AB_MIXED_CASE, buffer, AB_MAX_FIELD_VALUE );
bMixedCase = atoi(buffer);

ab->GetPropertyAttribute( "INIT_LIST", AB_PROPERTY_LENGTH,
buffer, AB_MAX_FIELD_VALUE);
  /* len will include space for the trailing null */
  len = atoi (buffer);
  szInitlist = new char[len];
  ab->GetProperty( AB_INIT_LIST, szInitlist, len );
 .
 .
 .
  delete szInitlist;
```

See Also

See Chapter 13, "Properties" for more information on properties.

GetPropertyAttribute (overloaded)

Retrieves a property attribute.

Syntax

```
virtual Boolean QMSAddressBroker::GetPropertyAttribute (
    const char* in_pszPropName,
    const unsigned long in_ulAttributeName,
    char* out_pszAttributeValue,
    const unsigned long in_ulBufferSize )
virtual Boolean QMSAddressBroker::GetPropertyAttribute (
    const unsigned long in_ulPropID,
    const unsigned long in_ulAttributeName,
    char* out_pszAttributeValue,
    const unsigned long in_ulBufferSize )
```

Arguments

in_pszPropName A valid property name. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

in_ulPropID A valid property symbolic constant. *Input*.

in_ulAttributeName
The symbolic constant for the attribute value to retrieve. *Input*.

out_pszAttributeValue
Pointer to the attribute value (string) to be loaded. *Output*.

in_ulBufferSize The size of the string buffer. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

None.

Alternates

None.

Notes

The `GetPropertyAttribute` function that accept a property ID provide slightly better performance.

Attribute Values

`AB_PROPERTY_DATA_TYPE (size = 2)`
"N" (integer), "B" (Boolean), or "C" (string).

`AB_PROPERTY_DEFAULT_VALUE`
Default property value. The size of `AB_PROPERTY_DEFAULT_VALUE` is determined by the value assigned to `AB_PROPERTY_LENGTH`.

`AB_PROPERTY_DESCRIPTION (size = 101)`
Short (100-character) description of property.

`AB_PROPERTY_ID (size = 12)`
Property ID.

`AB_PROPERTY_LENGTH (size = 12)`
Length of property value.

`AB_PROPERTY_NAME (size = 33)`
Property name.

`AB_PROPERTY_READ_ONLY (size = 2)`
"1" property is read-only.
"0" property is read/write.

Example 1

```
char datatype[2];
char length[13];
ab->GetPropertyAttribute ( "MIXED CASE", AB_PROPERTY_DATA_TYPE,
datatype,
2);
ab->GetPropertyAttribute ( "INIT_LIST", AB_PROPERTY_LENGTH, length,
13);
```

Example 2

```
char datatype[2];
char length[13];

ab->GetPropertyAttribute ( AB_MIXED_CASE, AB_PROPERTY_DATA_TYPE,
datatype,
2);
ab->GetPropertyAttribute ( AB_INIT_LIST, AB_PROPERTY_LENGTH, length,
13);
```

See Also

See Chapter 13, "Properties" for more information on properties.

GetRecord

Advances the pointer to the next record in the output record buffer.

Syntax

```
virtual Boolean QMSAddressBroker::GetRecord ( )
```

Arguments

None.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

`ProcessRecords`

Alternates

None.

Notes

The first call to `GetRecord` sets a pointer to the first output record. Subsequent calls advance the pointer. When no more data is found, the return value **FALSE** is returned.

Use the `GetField` member function to retrieve record field values. Use the ResetRecord member function to reset the record pointer to the first record.

Example

```
char addrln[61];
while ( ab->GetRecord ())
{
ab->GetField ( "AddressLine", addrln, 61 )
...
}
```

GetStatus

Returns status or error codes and messages.

Syntax

```
virtual Boolean QMSAddressBroker::GetStatus (
   unsigned long & out_ulStatus,
   char* out_pszStatusMsg,
   const unsigned long in_ulBufferSize )
```

Arguments

out_ulStatus Status or error code returned. *Output*.

out_pszStatusMsg Status or error message returned. *Output*.

in_ulBufferSize The size of the string buffer. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

None.

Alternates

None.

Notes

Generally, a 2048-character buffer is sufficient, although the actual message size varies.

LookupRecord

Processes a single incomplete U.S. address record or performs a reverse lookup on a Canadian postal code.

Syntax

```
int QMSAddressBroker::LookupRecord ( )
```

Arguments

None.

Return Values

The `OUTPUT_FIELD_LIST` property defines the fields populated by **`LookupRecord`**, and the return codes listed below describe the search outcome. Codes are returned only when the relevant fields are included in `OUTPUT_FIELD_LIST`. A return value of zero (**0)** indicates an internal failure.

Return Codes

AB_LOOKUP_ADDRESS_LINE_INCOMPLETE

For a U.S. address, the FirmName or UnitNumber could not be resolved. Multiple incomplete records returned. The user can be prompted to submit more information. The most useful fields for resolving a match generally are `FirmName`, `HighUnitNumber`, `LowUnitNumber`, `MatchCode`, and `UnitType`.

Other helpful fields include `AddressLine`, `AddressLine2`, `CarrierRoute`, `CountyName`, `FIPSCountyCode`, `GovernmentBuildingIndicator`, `HighEndHouseNumber`, `LACSAddress`, `LastLine`, `LowEndHouseNumber`, `PostfixDirection`, `PrefixDirection`, `RoadClassCode`, `SegmentBlockLeft`, `SegmentBlockRight`, `State`, `UrbanizationName`, `USPSRangeRecordType`, `ZIP`, `ZIPCarrtSort`, `ZIPCityDelivery`, `ZIPClass`, `ZIPFacility`, and `ZIPUnique`.

For a Canadian postal code, the input Postal Code is resolved to a range of possible addresses that contain a single street number. The street umber suffix or unit number values will vary over the range.

AB_LOOKUP_LAST_LINE_NOT_FOUND

For a U.S. address, multiple incomplete records were returned; the LastLine was not resolved. Only the following output fields are returned: `MatchCode`, `CITY`, `State`, `ZIP` and `ZIPFacility`. For a Canadian postal code, this return code indicates that the input postal code was not found in the CPC data and is invalid.

AB_LOOKUP_MULTIPLE_MATCH

For a U.S. address, the address resolved to a multiple match. Multiple complete address records returned. Use iterative calls to GetRecord to retrieve possible matches. For a Canadian postal code, the postal code resolved to a range of possible addresses that vary over the street.

AB_LOOKUP_NOT_FOUND

No records returned, no address matched. Provide a more complete address. (This return code is not used for Canada.)

AB_LOOKUP_SUCCESS

For a U.S. address, a single complete address was matched and returned. For a Canadian postal code, a single address was matched and returned.

AB_LOOKUP_TOO_MANY_CITIES

No records returned. An incomplete LastLine matched over 100 cities. Provide a more complete address. (This return code is not used for Canada.)

Prerequisites

None.

Alternates

`SetRecord`

Notes

`LookupRecord` processes a single input record and should be used only when address information is insufficient for standardization. To process single or multiple records containing complete addresses, use ProcessRecords.

Minimally, address information for `LookupRecord` must include a street number, a partial street name, and/or valid LastLine information. For Canada, a valid postal code is required and will return a single address or a range of addresses.

`LookupRecord` is most useful in interactive programs, when an application may have to make several calls to `LookupRecord` in order to find a match for an incomplete address. In client/server and Internet environments, the record is transferred across the network with each call to `LookupRecord`. The function call does not return until the record is processed. When `LookupRecord` processes an address record and fails to find an exact match, it does an extensive search to find cities and streets that are possible matches.

The INPUT_FIELD_LIST property specifies the list of fields passed to **LookupRecord**. Generally, provide at least FirmName, AddressLine and LastLine fields as input to **LookupRecord**. For Canada, a valid Canadian Postal Code is the only input, and it is set using the PostalCode input field. Only one Postal Code can be processed at a time.

The OUTPUT_FIELD_LIST property specifies the list of possible fields returned.

The MAXIMUM_LOOKUPS property limits the number of multiples—possible matches—that are returned by **LookupRecord**. The upper limit of MAXIMUM_LOOKUPS is 100. For a Canadian postal code, if the MAXIMUM_LOOKUPS is set to 100, the AddressBroker software increases the MAXIMUM_LOOKUPS to 200.

Retrieve the list of possible matches using a 'while (GetRecord) do GetField' loop. No records are returned when the return value of **LookupRecord** is **AB_LOOKUP_NOT_FOUND** or **AB_LOOKUP_TOO_MANY_CITIES**.

Precisely recommends using **ProcessRecords** instead of **LookupRecord**.

Example

In an interactive application, a user submits a partial address to **LookupRecord**. The return code is **AB_LOOKUP_LAST_LINE_NOT_FOUND**. For a U.S. address, this code indicates the user did not enter enough information for **LookupRecord** to resolve the city, state or ZIP Code. The application prompts the user to select from the list of possible cities and states returned by **LookupRecord**. The user selects the necessary information and resubmits the address to **LookupRecord**. For a Canadian postal code, this return code indicates that the input postal code was not found in the CPC data and is invalid.

This time the return code is **AB_LOOKUP_ADDRESS_LINE_INCOMPLETE**. The user resolved the last line problem, but the return code indicates the address line could be more specific. For a U.S. address, it is missing information on the firm name or unit number (suite, apartment, etc.). The application can prompt the user to select from the list of possibles returned by this call to **LookupRecord**. The user enters the additional information and resubmits the address to **LookupRecord**, and **AB_LOOKUP_SUCCESS** is returned. For a Canadian postal code, the **AB_LOOKUP_ADDRESS_LINE_INCOMPLETE** code indicates that the input Postal Code resolved to a range of possible addresses that contain a single street number. The street number suffix or unit number values will vary over the range. For example, a Canadian postal code of T3C 2K7 could resolve to 123 A - 123 G Maple Street (when the street suffix varies) or 123 Maple Street Unit 1-100 (when the unit number changes). A valid postal code for one address submitted to **lookupRecord** returns **AB_LOOKUP_SUCCESS**.

When the next address is entered, **LookupRecord** returns the status code **AB_LOOKUP_MULTIPLE_MATCH.** This indicates multiple complete matches were found. For a U.S. address, the user may then be prompted to select from the list of possible matches. The selected address is resubmitted to **LookupRecord** to ensure that it is entirely correct, and that **AB_LOOKUP_SUCCESS** is returned. For a Canadian postal code, the **AB_LOOKUP_MULTIPLE_MATCH** code indicates a postal code that resolved to a range of possible addresses that vary over the street. For example, a Canadian postal code could resolve to 100-120 Elm, Calgary, AB or 150-165 Maple, Calgary, AB.

ProcessRecords

Processes a set of one or more input records.

Syntax

```
virtual Boolean QMSAddressBroker::ProcessRecords ( )
```

Return Values

Returns **TRUE (1)** if successful, **FALSE (0)** if unsuccessful.

Prerequisites

None.

Alternates

`SetRecord`

Notes

Each record should contain enough address information for standardization. For records containing incomplete addresses, use LookupRecord, which progressively returns address choices for one input record at a time.

The function call does not return until all of the records are processed.

See Also

See Chapter 13, "Properties" for more information on properties.

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information on fields.

ResetField

Resets the output pointer to the first value of an output field.

Syntax

```
virtual Boolean QMSAddressBroker::ResetField (
    const char* in_pszFieldName,
    const char* in_pszLogicalName )
```

Arguments

in_pszFieldName

A valid field name listed in the `OUTPUT_FIELD_LIST` property. Some field names require a logical name. The logical name may be appended to *in_FieldName* in brackets, or passed in the *in_LogicalName* parameter (see the examples for this function). The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

in_pszLogicalName

The logical name required by the value of *in_pszFieldName*. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

`GetField`

Alternates

None.

Notes

`ResetField` returns **FALSE** when, for any reason, *in_pszFieldName* is not found.

If **`GetField`** is called with the logical name in brackets, then **`ResetField`** must be called with the logical name in brackets. Similarly, if the logical name is passed as a separate parameter in **`GetField`**, then it should be a separate parameter in **`ResetField`**.

All Spatial+, GDL, and Demographic fields require logical names. GeoStan and GeoStan Canada fields do not.

Example 1

```
// Example using field name with its logical name in brackets.
while ( ab->GetField ( "PolygonName[COUNTIES]", polygonname, 128 ) )
{
    ...
}
ab->ResetField ( "PolygonName[COUNTIES]" );
```

Example 2

```
// Example using field name with its logical name as a separate
argument.
while ( ab->GetField ( "PolygonName", "COUNTIES", polygonname, 128 ) )
{
    ...
}
ab->ResetField ( "PolygonName", "COUNTIES" );
```

See Also

See "INPUT_FIELD LIST and OUTPUT_FIELD_LIST" on page 66 for more information on fields.

ResetRecord

Resets output record pointer to the first record in the output record buffer.

Syntax

```
virtual Boolean QMSAddressBroker::ResetRecord ( )
```

Arguments

None.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

`GetField`

Alternates

None.

SetField

Sets an input field value in the current input record.

Syntax

```
virtual Boolean QMSAddressBroker::SetField (
   const char* in_pszFieldName,
   const char* in_pszFieldValue )
```

Arguments

in_pszFieldName	A valid field name listed in the `INPUT_FIELD_LIST` property. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.
in_pszFieldValue	The string value to assign to the field. Maximum string length is 256 characters. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

`SetProperty`

Alternates

None.

Notes

Reserved characters: The `RECORD_DELIMITER`, `FIELD_DELIMITER`, and `VALUE_DELIMITER` properties have default values of line feed, tab, and CTRL-A, respectively. If your data contains any of these characters, you **must** reset the appropriate property to a different character. In addition, your data may not contain the null character.

This function sets the fields of the current input record only. **`SetRecord`** must be called *after* each input record.

Example

```
/*Assumes a record consists of addressline and lastline
  only. */
char addressline [61];
char lastline[61];
while ( more_data )
{
```

```
        ab->SetField ( "AddressLine", addressline );
        ab->SetField ( "LastLine", lastline );
        ab->SetRecord ();
    }
```

See Also

See "INPUT_FIELD LIST and OUTPUT_FIELD_LIST" on page 66 for more information on fields.

SetProperty (overloaded)

Assign a property value.

Syntax

```
virtual Boolean QMSAddressBroker::SetProperty (
    const char* in_pszPropName,
     const Boolean in_bPropValue )
virtual Boolean QMSAddressBroker::SetProperty (
    const char* in_pszPropName,
     const char* in_pszPropValue )
virtual Boolean QMSAddressBroker::SetProperty (
    const char* in_pszPropName,
     const unsigned long in_ulPropValue )
virtual Boolean QMSAddressBroker::SetProperty (
    const unsigned long in_ulPropID,
     const Boolean in_bPropValue )
virtual Boolean QMSAddressBroker::SetProperty (
    const unsigned long in_ulPropID,
     const char* in_pszPropValue )
virtual Boolean QMSAddressBroker::SetProperty (
    const unsigned long in_ulPropID,
     const unsigned long in_ulPropValue )
```

Arguments

in_pszPropName	A valid property name. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.
in_ulPropID	The valid symbolic constant of the property being set. *Input*.
in_bPropValue	A Boolean value to assign to the property. *Input*.
in_pszPropValue	A string value to assign to the property. The enumerated constants `AB_*` available in `abtypes.h` may be passed as a string, as input to **`SetProperty`**, by dropping the `AB_` from the beginning. For example, `AB_INPUT_PARSED` may be passed as "`INPUT_PARSED`". *Input*.
in_ulPropValue	The long value to assign to the property. *Input*.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

`QMSAddressBroker::create`

Alternates

None.

Notes

The `SetProperty` functions set input properties. The specific `SetProperty` function to use depends on the data type of the property you are setting. The `SetProperty` functions that accept a property ID provide slightly better performance.

Example 1

```
ab->SetProperty ( "MIXED CASE", (Boolean) TRUE );
```

Example 2

```
ab->SetProperty ( AB_MIXED_CASE, (Boolean) TRUE );
```

Example 3

```
ab->SetProperty ( "INIT_LIST", "GEOSTAN | GEOSTAN_Z9" );
```

Example 4

```
ab->SetProperty ( AB_INIT_LIST, "GEOSTAN | GEOSTAN_Z9" );
```

Example 5

```
ab->SetProperty (AB_INPUT_MODE, “INPUT_PARSED”);
```

See Also

See Chapter 13, "Properties" for more information on properties.

SetRecord

Adds data for the current record to the input record buffer and advances the input record pointer to the next empty record in the buffer.

Syntax

```
virtual Boolean QMSAddressBroker::SetRecord ( )
```

Arguments

None.

Return Values

TRUE (1) if successful, **FALSE (0)** if unsuccessful.

Prerequisites

`SetField`

Alternates

None.

Notes

Each call to the `SetRecord` member function sets the data for the current record and advances the record pointer to the next empty record.

ValidateProperties

Validates properties for consistency and completeness.

Syntax

```
virtual Boolean QMSAddressBroker::ValidateProperties ( )
```

Arguments

None.

Return Values

TRUE (1) if all properties are valid, **FALSE (0)** if one or more properties fail to validate.

Prerequisites

`SetProperty`

Alternates

None.

Notes

The `ValidateProperties` function verifies the values of initialization and processing control properties to ensure a complete and compatible set of values are available to AddressBroker. Call this function after one or more properties have been set and before calling SetField or any processing functions.

When `ValidateProperties` returns **TRUE**, it indicates all properties have been successfully validated and that AddressBroker is ready to process records. In some cases, all properties can be validated in a single function call.

See Also

See Chapter 13, "Properties" for more information on properties.

QMSABStatus class

Use the **QMSABStatus** class to handle the AddressBroker Exception Status object.

constructor (overloaded)

Creates and initializes an instance of the QMSABStatus object.

Syntax

```
QMSABStatus ( )
QMSABStatus ( QMSABStatus &statusObj )
QMSABStatus ( const char* message, ... )
QMSABStatus ( QMSABStatusType type,
const char* message, ... )
QMSABStatus ( QMSABStatusType type,
const char* message,
va_list args )
```

Arguments

&statusObj	A **QMSABStatus** object. *Input*.
message	A string value containing a status message. The maximum length of the message is equal to `MAX_MESSAGE_LENGTH (1024)`. *Input*.

type	A valid **QMSABStatusType**, as listed here. *Input*.
	`ABSTATUS_NONE`
	`ABSTATUS_FATAL`
	`ABSTATUS_ERROR`
	`ABSTATUS_WARN`
	`ABSTATUS_INFO`
	`ABSTATUS_DEBUG`

args	A variable argument list (standard C). *Input*.

Return Values

Returns a pointer to a new instance of the status object if successfully created, **NULL** if unsuccessful.

Prerequisites

None.

Alternates

None.

Message

Mechanism to retrieve a status message.

Syntax

```
const char* Message( void )
```

Arguments

None.

Return Value

Returns a pointer to a string containing a status message.

Prerequisites

None.

Alternates

None.

Status

Mechanism to retrieve a status type.

Syntax

```
QMSABStatusType Status( void )
```

Arguments

None.

Return Value

Returns **`QMSABStatusType`**.

Prerequisites

None.

Alternates

None.

QMSABLogFile class

Use the **`QMSABLogFile`** class to direct logging to console or file.

constructor (overloaded)

Creates and initializes an instance of the **`QMSABLogFile`** object.

Syntax

```
QMSABLogFile ( )
QMSABLogFile( Boolean useTerm,
const char* logFile,
Boolean useEventLog,
const char* progName )
```

Arguments

useTerm	A Boolean indicating the use of the terminal for status messages. *Input*.
logFile	A string containing the name of a log file. *Input*.
useEventLog	A Boolean specifying *logFile* to be eventlog (on Windows platforms) or syslog (on UNIX platforms). *Input*.
progName	The name of the program sending the status messages. Default is `abserver`. *Input*.

Return Value

Returns a pointer to a new instance of the log file object if successfully created, **`NULL`** if unsuccessful.

Prerequisites

None.

Alternates

None.

GetLogFilePath

Retrieves the log file path.

Syntax

```
const char* GetLogFilePath ( void )
```

Arguments

None.

Return Value

A string containing the path to a log file.

Prerequisites

None.

Alternates

None.

SetLogFilePath

Sets the log file path.

Syntax

```
void SetLogFilePath ( const char* path )
```

Arguments

path	A string containing the path to a log file. *Input*.

Return Value

None.

Prerequisites

None.

Alternates

None.

SetLogProgramName

Sets the name of the program sending status messages.

Syntax

```
void SetLogProgramName ( const char* progName )
```

Arguments

progName	The name of the program sending the status messages. Default is `abserver`. *Input*.

Return Value

None.

Prerequisites

None.

Alternates

None.

DisableEventLog

Flag disabling use of eventlog (windows) or syslog (unix).

Syntax

```
void DisableEventLog( void )
```

Arguments

None.

Return Value

None.

Prerequisites

None.

Alternates

None.

Notes

Default is **off**.

EnableEventLog

Flag enabling use of eventlog (windows) or syslog (unix).

Syntax

```
void EnableEventLog( void )
```

Arguments

None.

Return Value

None.

Prerequisites

None.

Alternates

None.

Notes

Default is **off**.

UsingEventLog

Retrieves the status of use of eventlog (windows) or syslog (unix).

Syntax

```
Boolean UsingEventLog( void )
```

Arguments

None.

Return Values

TRUE (1) indicates eventlog (Windows) or syslog (UNIX) is disabled. **FALSE (0)** indicates use of these logs is enabled.

Prerequisites

None.

Alternates

None.

DisableTermIO

Flag disabling terminal io.

Syntax

```
void DisableTermIO( void )
```

Arguments

None.

Return Value

None.

Prerequisites

None.

Alternates

None.

Notes

Default is **off**.

EnableTermIO

Flag enabling terminal io.

Syntax

```
void EnableTermIO( void )
```

Arguments

None.

Return Value

None.

Prerequisites

None.

Alternates

None.

Notes

Default is **off**.

UsingTermIO

Retrieves the status of terminal io.

Syntax

```
Boolean UsingTermIO( void )
```

Arguments

None.

Return Values

TRUE (1) indicates terminal /IO is disabled. **FALSE (0)** indicates terminal I/O is enabled.

Use **`debug`**, **`error`**, **`fatal`**, **`info`**, and **`warn`** to post messages to the log file.

Prerequisites

None.

Alternates

None.

debug, vdebug

Posts a debug status message.

Syntax

```
void debug ( const char* format, ... )
void vdebug ( const char* format, va_list args )
```

Arguments

format	A format string status message. Maximum length of a message is `MAX_MESSAGE_LENGTH (1024)`. *Input*.
args	A variable argument list (standard C). *Input*.

Return Value

None.

Prerequisites

None.

Alternates

None.

error, verror

Posts an error status message.

Syntax

```
void error ( const char* format, ... )
void verror ( const char* format, va_list args )
```

Arguments

format	A format string status message. Maximum length of a message is `MAX_MESSAGE_LENGTH (1024)`. *Input*.
args	A variable argument list (standard C). *Input*.

Return Value

None.

Prerequisites

None.

Alternates

None.

info, vinfo

Posts an info status message.

Syntax

```
void info ( const char* format, ... )
void vinfo ( const char* format, va_list args )
```

Arguments

format	A format string status message. Maximum length of a message is `MAX_MESSAGE_LENGTH (1024)`. *Input*.
args	A variable argument list (standard C). *Input*.

Return Value

None.

Prerequisites

None.

Alternates

None.

fatal, vfatal

Posts a fatal status message.

Syntax

```
void fatal ( const char* format, ... )
void vfatal ( const char* format, va_list args )
```

Arguments

format	A format string status message. Maximum length of a message is MAX_MESSAGE_LENGTH (1024). *Input*.
args	A variable argument list (standard C). *Input*.

Return Value

None.

Prerequisites

None.

Alternates

None.

warn, vwarn

Posts a warning status message.

Syntax

```
void warn ( const char* format, ... )
void vwarn ( const char* format, va_list args )
```

Arguments

format
A format string status message. Maximum length of a message is `MAX_MESSAGE_LENGTH (1024)`. *Input*.

args
A variable argument list (standard C). *Input*.

Return Value

None.

Prerequisites

None.

Alternates

None.

showStatus

Displays a status message.

Syntax

```
void showStatus ( QMSABStatus &status )
```

Arguments

status	An object of type **`QMSABStatus`**. *Input*.

Return Value

None.

Prerequisites

None.

Alternates

None.

Errors, messages, and status logs

The AddressBroker C++ API supports two independent methods of error handling:

- the use of a log file—assigned to AddressBroker's STATUS_LOG property—used in conjunction with a reporting threshold, assigned to AddressBroker's STATUS_LEVEL property.
- a try-throw-catch routine used in conjunction with the THROW_LEVEL property.

These two error handling methods are discussed in this section. See "GeoStan location codes" on page 433 for a discussion about the codes themselves.

Using STATUS_LEVEL and STATUS_LOG

These two properties do not require validation to be used or changed.

STATUS_LOG holds the output destination of all reported messages. Set AddressBroker's STATUS_LOG property to either:

- the path and file name for a status log to save status messages.
- the value CONSOLE to display status messages to a console window.

Set AddressBroker's STATUS_LEVEL property to the appropriate level of message reporting you require:

- FATAL—fatal errors, errors, and warnings.
- ERROR—errors and warnings only.
- WARN—warnings only.
- INFO—all information messages.
- NONE—no messages.
- DEBUG—debug messages; for development only.
- SERVER to report server level only messages. Default.

Using THROW_LEVEL

AddressBroker's THROW_LEVEL property determines the level at which your application is notified of an error or status condition. Use THROW_LEVEL in combination with a try-throw-catch routine to manage errors and status conditions. THROW_LEVEL does not require validation to be used or changed. Legal values for THROW_LEVEL are:

- FATAL—fatal errors, errors, and warnings.
- ERROR—errors and warnings only. Default.
- WARN—warnings only.
- INFO—all information messages.

- NONE—no messages.
- DEBUG—debug messages; for development only.

12 – ActiveX Interface

In this chapter

This chapter describes in detail the ActiveX interface to AddressBroker. For general information on AddressBroker, please refer to Chapters 1, 2, and 4 of this manual.

This chapter provides a tutorial using the AddressBroker ActiveX component in a Visual Basic coding environment. This tutorial shows you how to use most of AddressBroker's functionality, yet is general enough that you can port it to other coding environments, including ASP and other Web development environments. A complete function and property reference follows the tutorial. The final section of this chapter discusses error handling.

The naming convention for AddressBroker ActiveX functions is `FunctionNameX`. All ActiveX functions use this naming convention.

The AddressBroker ActiveX component is an ActiveX object for use primarily as an automation tool. It can be used in Integrated Development Environments (IDE) such as Visual Basic, Delphi, and Power Builder. It can also be used with Web scripting languages such as VBScript. JavaScript and JScript are not supported.

IDEs and enumerated types

Some AddressBroker properties have enumerated types. Please be aware that some Integrated Development Environments do not support the use of enumerated types. Please see `ABXConstants.asp`, shipped on the installation CDs for an example work-around.

AddressBroker properties vs. ActiveX properties

Both AddressBroker and ActiveX make use of a "property" concept.

- The AddressBroker ActiveX properties have a 1:1 naming correspondence with AddressBroker properties—unless otherwise noted.
- The ActiveX component includes a small set of non-AddressBroker properties—that is, properties specific to the AddressBroker ActiveX interface. These properties are all discussed as "ActiveX only" properties. The information conveyed in this small set of properties is equivalent to information passed via function parameters in AddressBroker's other APIs.
- Please note that when using functions to manipulate properties programmatically, you can only set AddressBroker properties. Using an "ActiveX only" property as an argument to one of the functions result in an error.

Accessing the AddressBroker ActiveX library

If you chose the default installation, there are no additional steps required to access the AddressBroker ActiveX library. If you did not choose the default installation, make sure the library is in the same directory as the ActiveX component.

The calling language determines the method by which the component is accessed. For instance, in Visual Basic, you must make a reference to the component. In C++, you must add a `#include` statement for `qmsabactx.dll` to your code.

AddressBroker ActiveX tutorial

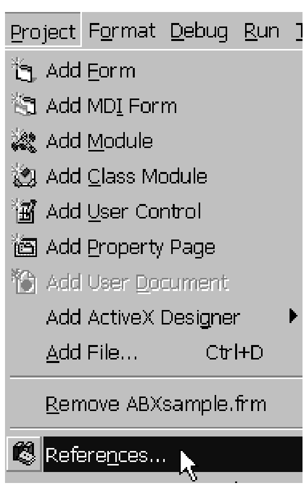
This chapter describes the steps necessary to develop a client application using the AddressBroker ActiveX component in Visual Basic 5. The example shows some Visual Basic sample code that does address standardization and enhancement. The sample includes a form with a "Process Addresses" button and an output text box. When the Process Addresses button is pressed, addresses are processed using `Firm_Name` and `Address` fields as input.

This sample application standardizes the address data and augments it with city, state, and 9-digit ZIP Code information using the GeoStan data directory. Then it retrieves the name and status of the geographic polygon in which the address is found using a Spatial+ data file.

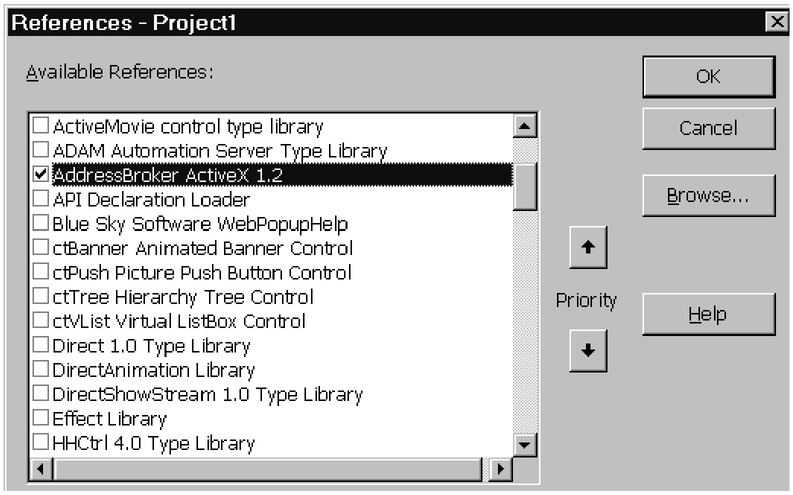
Sample Visual Basic code (`ABXSample.frm`) for a client application is shipped with AddressBroker. It is located in the `Samples\ActX\VB` subdirectory. For detailed descriptions of ActiveX functions and properties, refer to the "AddressBroker ActiveX functions" on page 294. For detailed descriptions of AddressBroker properties and fields, refer to the specific sections on these topics.

Step 1: Create and initialize the object

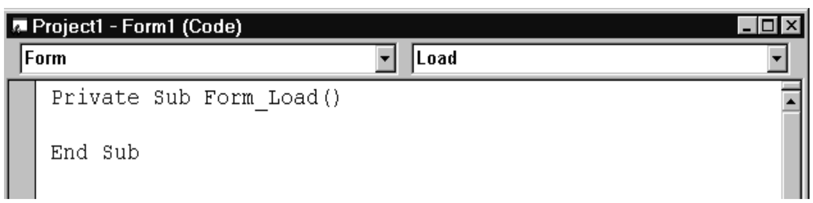
To begin, enable the AddressBroker ActiveX type library. From the **Project** menu, select the **References**.



Select **AddressBroker ActiveX 1.2**. Click **OK**.



Next a variable of the proper type must be defined in the global area. Double-click on the Form1 window to open the code window.

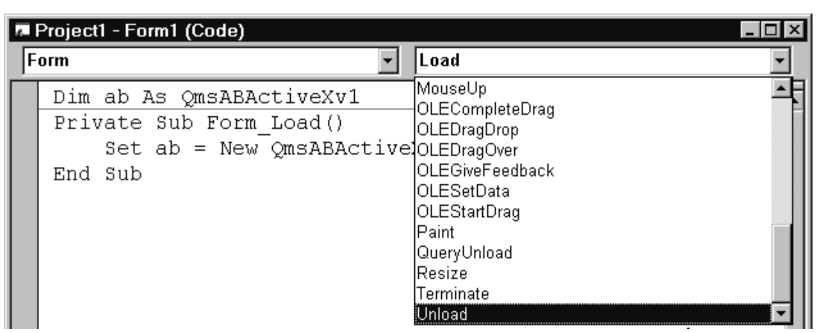


The component must be created before it is used. When the form using the component is loaded, (`Form_Load` event) create the component. To do this:

1. Select **Load** from the event drop-down menu.
2. Add code as needed so the code window reads:

```
Dim ab As QmsABActiveXv1
Private Sub Form_Load ()
Set ab = New QmsABActiveXv1
End Sub
```

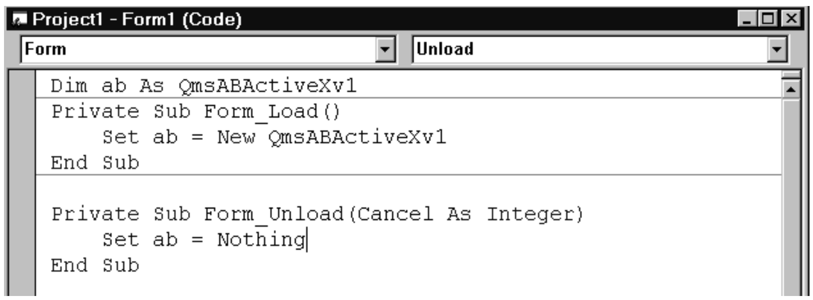
The component must be deleted when it is finished being used. When the form using the component is unloaded, (`Form_Unload`), delete the component.



To do this step:

3. Select Unload from the event drop-down menu as shown in screen shot, above.
4. Add code as needed so the code window reads:

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

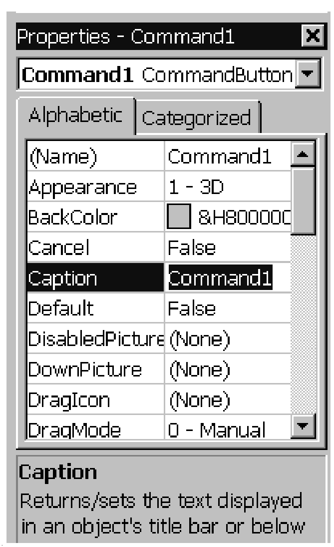


Next, add some functionality to the form. To do this:

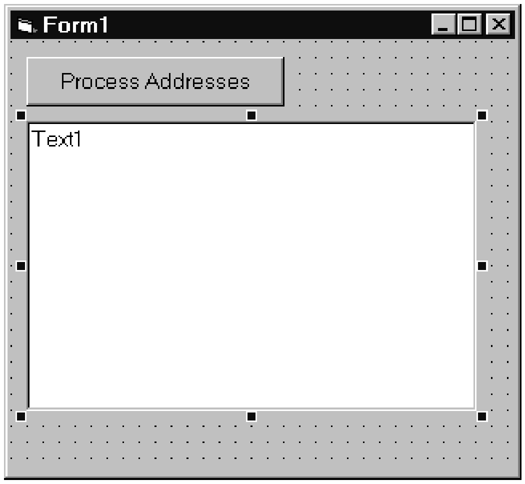
5. Select the Form1 window.
6. Add a command button to the form.

   This is accomplished by double-clicking on the command button icon shown on the left side of the VB screen.
7. Set the caption property for the command button.

   To do this, use the properties dialog found on the right hand side of the VB screen. Change the text from “Command1” to “Process Addresses”.

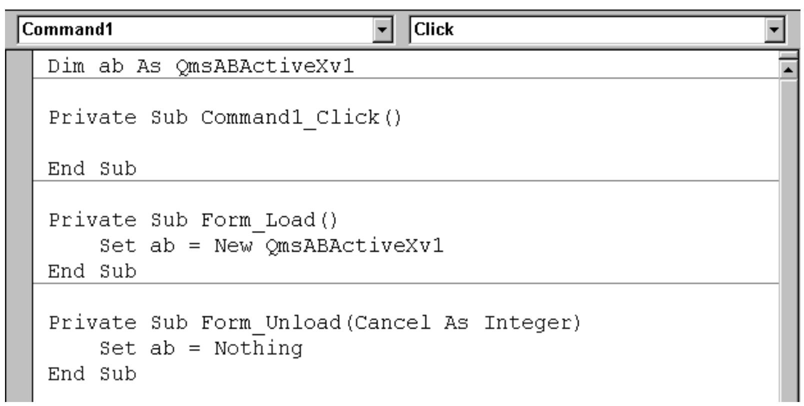


8. Add a text box to the form. You can change the name of the text box if you wish. This example keeps the default name “Text1”.



9. Use the property dialog again, this time to set the multiline property for the text box to “true”.

10. Double-click on the “Process Addresses” button to access the code window for the Command1_Click event. This code window is shown below.



11. The variables used in the Command1_Click subroutine must be defined and the component initialized. As part of the initialization process, you must set the ActiveX properties required for the type of application you are building.

For clients, you are required to set the “ActiveX only” properties HostList, `TransportProtocol`, `UserName`, `Password`, and—optionally—`InitializationFileName`. Precisely also recommends setting the `LogFileName` property, unless you plan to use the default value, `ab.log`.

Once the required ActiveX properties have been set, make a call to InitializeX. The following code is for a client application. Enter it in the Command1_Click subroutine:

Dim ab as QmsABActiveXv1

```
Private Sub Form_Load()
   Set ab = New QmsABActiveXv1
End Sub

  Private Sub Command1_Click()
   Dim return_value As Integer
   Dim msg_code
   Dim msg
   Dim valid_properties As Integer
   Dim valid_state As Integer
   Dim firmname
   Dim addressline
   Dim city
   Dim state
   Dim zip10
   Dim polygonname
   Dim polygonstatus
   Dim endl As String * 2
   'Define an end of line string for use in outputting results.
   endl = chr$(13) + chr$(10)
   'Set the Active X properties for a client application.
   'We plan to use the LogFileName default; no need to set here.
   'Set the list of host:port pairs to use.  Only use one server in this
example.
   ab.HostList = "localhost:4660"
   ab.TransportProtocol = "socket"
   ab.Username = "MyUserName"
   ab.Password = "MyPassword"
   'Default is false, for clients.
'Optionally specify an initialization file
   'ab.InitializationFileName = "ABActiveX.ini"
   return_value = ab.InitializeX()
   'Check the return value for success or failure.
```

Step 2: Set properties

You should assign a minimal set of AddressBroker properties in your application. In the client/server application shown here, logical names and paths are set on the server. The input and output field name properties and the initialization list property are set on the client. These topics are discussed in detail in the chapters devoted to client applications earlier in the manual.

The client refers to the logical names to access geo-demographic data on the server. The logical names the client uses must match those set on the server. In the sample code shown in AddressBroker ActiveX setting properties example, the logical names `GEOSTAN`, `GEOSTAN_Z9`, and `COUNTIES` refer to a GeoStan data directory, a GEOSTAN ZIP Code file, and a Spatial+ polygon file, respectively. Next, the examples tells AddressBroker to use the `FirmName`, `AddressLine`, and `LastLine` field values from each input record. In this example, `FirmName` and `AddressLine` fields are enhanced with City, State, and ZIP10 information from the GeoStan data file. `PolygonName` and `PolygonStatus` are also retrieved from the COUNTIES file.

You may set other properties on the client. In the sample code, `KEEP_MULTIMATCH` and `BUFFER_RADIUS` are set. See Chapter 13, "Properties" for a detailed discussion.

To accomplish this step, add this additional code to the Command_Click1 subroutine:

ActiveX property reference syntax

```
' This example shows how to set properties in VB using methods
' VB requires you to specify the return value, even if unused.

result = ab.SetPropertyX("INIT_LIST", "GEOSTAN|GEOSTAN_Z9|Counties")
result = ab.SetPropertyXBool("MIXED CASE", True)
' Set enumerated values using the Property ID or the equivalent value
result = ab.SetPropertyXLong("INPUT MODE", 0)
result = ab.SetPropertyXLong("INPUT MODE", ABX_INPUT_NORMAL)

' This example shows how to set properties in VB using ActiveX properties

ab.InitList = "GEOSTAN|GEOSTAN_Z9|Counties"
ab.MixedCase = True
' Set enumerated values using the Property ID or the equivalent value
ab.InputMode = ABX_INPUT_NORMAL
ab.InputMode = 0
```

AddressBroker ActiveX setting properties example

```
'Identify the Logical names to use. These must match the names used on
the server.
   ab.InitList = "GEOSTAN|GEOSTAN_Z9|Counties"
   'Identify the inputs. Although we do this only once in this example
  'it is a dynamic property, so you can set it at any time, as many times
as you want
   ab.InputFieldList = "FirmName|AddressLine|LastLine"
   'Identify the output we expect returned
   ab.OutputFieldList =
"FirmName|AddressLine|City|State|Zip10|PolygonName[COUNTIES]|PolygonSta
tus[COUNTIES]"
   'Set some other properties that affect behavior
   'Only keep one output record for each input record.
   ab.KeepMultimatch = False
   'Set a 200 foot buffer instead of using the default
   ab.BufferRadius = 200
```

Step 3: Validate properties (optional)

Use the ValidatePropertiesX function to send the property definitions to AddressBroker for validation. When **`ValidatePropertiesX`** returns **TRUE**, the AddressBroker client object is initialized and ready for processing. If any property setting is invalid, an error is generated. You can use GetStatusX to retrieve error messages in the event **`ValidatePropertiesX`** does not return successfully.

All AddressBroker properties must be set and validated before data can be input or processed. In client mode, calling this function results in a server transaction. To accomplish this step, add this code to the Command_Click1 subroutine:

AddressBroker ActiveX ValidatePropertiesX example

```
‘Check to see that properties are valid
   valid_properties = ab.ValidatePropertiesX()
   If valid_properties = 0 Then
     return_value = ab.GetStatusX(msg_code, msg)
    Text1.Text = "Validate Properties failed msg = " + endl + msg + endl
   End If
```

ValidatePropertiesX can be called multiple times in your application. For example, you can initially set and validate a group of properties, then allow the end user to dynamically select new values and revalidate the settings.

Step 4: Enter input records and field values

Next call SetFieldX function to specify the input field values. Note that these are the same fields you specified initially by setting the `InputFieldList` property (see “AddressBroker ActiveX setting properties example” on page 290).

The SetRecordX function adds the data for the current record to the input record list and advances the record pointer. An input value need not be set for every field in a record. For instance, in our example, an individual record that did not contain FirmName information could still be processed.

To accomplish this step, add this code to the Command_Click1 subroutine:

AddressBroker ActiveX input data example

```
‘Enter a few records for processing
   If valid_properties <> 0 Then
      ‘Fill in a record…
      return_value = ab.SetFieldX("FirmName", "Precisely")
      return_value = ab.SetFieldX("AddressLine", "4750 Walnut")
      return_value = ab.SetFieldX("LastLine", "Boulder, CO")
      return_value = ab.SetRecordX()
      ‘Fill in the next record…
      return_value = ab.SetFieldX("FirmName", "White House")
      return_value = ab.SetFieldX("AddressLine", "1600 Pennsylvania")
      return_value = ab.SetFieldX("LastLine", "Washington, DC")
      return_value = ab.SetRecordX()
    End If
```

Step 5: Process records

Once all the input data has been entered, you are ready to process the records. Use the `ProcessRecordsX` function to process your address records. This sends all the data to the server for processing.

Note: Calling this function clears the input record buffer, even if the call fails.

To accomplish this step, add this code to the Command_Click1 subroutine:

AddressBroker ActiveX record processing example

```
If valid_properties <> 0 Then
      valid_state = ab.ProcessRecordsX()

      If valid_state = 0 Then
        return_value = ab.GetStatusX(msg_code, msg)
        Text1.Text = "Process Records failed msg = " + endl + msg + endl
      End If
    End If
```

Step 6: Retrieve address records and field values

Use the GetRecordX and GetFieldX* functions to retrieve the output data. “AddressBroker ActiveX data retrieval example” adds the output data to a text string that displays in Form1’s text box (Text1).

To accomplish this step, add this code to the Command_Click1 subroutine:

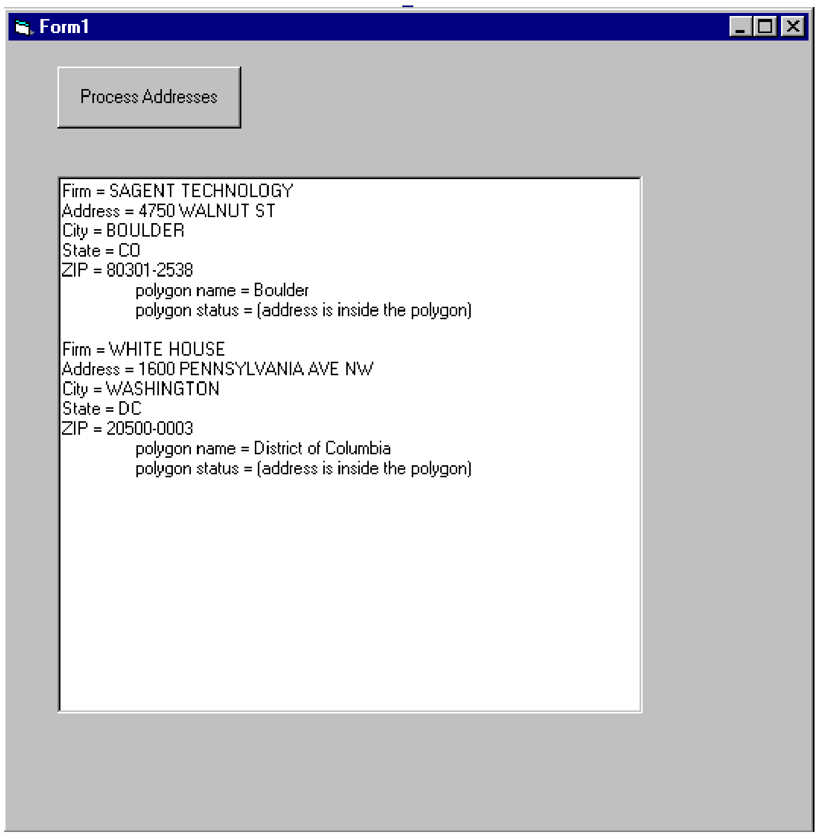
AddressBroker ActiveX data retrieval example

```
    If valid_state <> 0 Then

      Text1.Text = ""

      Do While ab.GetRecordX()
        ‘get address data
        return_value = ab.GetFieldX("FirmName", 41, firmname)
        return_value = ab.GetFieldX("AddressLine", 61, addressline)
        return_value = ab.GetFieldX("City", 29, city)
        return_value = ab.GetFieldX("State", 3, state)
        return_value = ab.GetFieldX("ZIP10", 11, zip10)

        ‘output the basic address
        Text1.Text = Text1.Text + "Firm = " + firmname + endl
        Text1.Text = Text1.Text + "Address = " + addressline + endl
        Text1.Text = Text1.Text + "City = " + city + endl
        Text1.Text = Text1.Text + "State = " + state + endl
        Text1.Text = Text1.Text + "ZIP = " + zip10 + endl

         ‘ Get polygon namd and status with a multivalued return
        Do While ab.GetFieldX("PolygonName[Counties]", 128, polygonname)
          ‘Print out the polygon name
         Text1.Text = Text1.Text + "     polygon name = " + polygonname
+ endl
          ‘…and the polygon status (paired with each polygon name found)
          return_value = ab.GetFieldX("PolygonStatus", 2, polygonstatus)
          If polygonstatus = "P" Then
            Text1.Text = Text1.Text + "     polygon status = (address is
inside the polygon)" + endl + endl
          ElseIf polygonstatus = "I" Then
            Text1.Text = Text1.Text + "     polygon status = (address is
inside the polygon and within the buffer radius)" + endl + endl
          ElseIf polygonstatus = "B" Then
            Text1.Text = Text1.Text + "     polygon status = (address is
outside the polygon but within the buffer radius)" + endl + endl
          Else
```

```
                Text1.Text = Text1.Text + "     polygon status = (unknown
condition)" + endl + endl
            End If
        Loop

      Loop
    End If
```

Once you have entered all the code in the code window, compile and run your program. Press the “Process Addresses” button on Form1.

The output appears in the Form1 text box, as shown below. The addresses have been standardized and enhanced with spatial information.



AddressBroker ActiveX functions

This section describes in detail the functions available through the AddressBroker ActiveX component.

Quick reference

QmsActiveXv1 class functions

Initialization functions

InitializeX

Initializes the ActiveX component. Call this function before calling any other AddressBroker ActiveX function.

Property functions

Note: These functions manipulate AddressBroker properties only. Using these functions with an ActiveX only property results in error. See "AddressBroker ActiveX properties" on page 318.

GetPropertyX*

Retrieves the value of an input or output property.

GetPropertyAttributeX

Retrieves a property attribute, such as its name, data type, and description.

SetPropertyX*

Sets the value of a property.

ValidatePropertiesX

Validates properties for consistency and completeness. This function must be called after properties are set and before calls to `SetFieldX`.

Field/data functions

ClearX

Clears the input and output record buffers and resets all counter properties to zero.

GetFieldX*

Retrieves the value(s) of an output field in the current output record. Call iteratively for fields that contain multiple values.

GetFieldAttributeX

Retrieves a field attribute, such as its data type and description.

ResetFieldX

Resets the output field pointer to the first value of an output field.

SetFieldX

Sets an input field value in the current input record.

GetRecordX

Retrieves the record and advances the output record pointer.

ResetRecordX

Resets the output record pointer to the first record of the output record buffer.

SetRecordX

Adds the data for the current record to the input record buffer and advances the input record pointer to the next empty record.

Processing functions

LookupRecordX

Processes a single incomplete address record.

ProcessRecordsX

Processes a set of one or more address records.

Reporting functions

GetStatusX

Retrieves status and error codes and messages.

QMSActiveXv1 class

The AddressBroker ActiveX component has one class. All of the functions described in this section are members of this class.

The class name is `QmsABActiveXv1`. The two interfaces supported by the class are: `IQmsABActiveXv1` and `IAddressBrokerX2`. The `IAddressBrokerX2` interface is the default interface for the class.

ClearX

Clears input and output record buffers and resets counter properties.

Syntax

```
Integer ClearX ( )
```

Arguments

None.

Return Values

Returns **1** if successful, **0** if unsuccessful.

Prerequisites

None.

Alternates

None.

GetFieldX*

Retrieves output field value(s) from the current output record.

Syntax

```
Integer GetFieldX (
   String in_FieldName,
   Integer in_MaxStringSize,
   Variant out_FieldValue )
Integer GetFieldXUseLogical (
   String in_FieldName,
   String in_LogicalName,
```

```
Integer in_MaxStringSize,
Variant out_FieldValue )
```

Arguments

in_FieldName

A valid, fully specified field name listed in the OUTPUT_FIELD_LIST property (see Example). The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

in_LogicalName

The logical name required by the value of *in_FieldName*. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

in_MaxStringSize

The maximum size of the output string. *Input*.

out_FieldValue

Pointer to the field value to be loaded. All values are returned as strings. *Output*.

Return Values

Returns **1** if successful, **0** if unsuccessful.

Prerequisites

SetFieldX

Alternates

None.

Notes

The **GetFieldX*** functions retrieve a field value from the current output record. Call **GetFieldX*** iteratively for multi-valued fields. Use the ResetFieldX function to reset the field to its first value. To retrieve single value fields more than once, you must call **ResetFieldX**.

All Spatial+, GDL, and Demographic fields require logical names. GeoStan and GeoStan Canada fields do not.

VB Example 1

```
'Example using a field that does not require a logical name.
Dim return_value As Integer
Dim City
```

```
...
return_value = ab.GetFieldX ( "City", 29, city )
```

VB Example 2

```
'Example using a multivalued field with its logical name in brackets.
Dim PolygonName
...
Do while ab.GetFieldX ( "PolygonName[Counties]", 128, PolygonName )
    ...
```

VB Example 3

```
'Example using a multivalued field
'with its logical name as separate argument.
Dim PolygonName
...
Do while ab.GetFieldXUseLogical ( "PolygonName", "Counties", 128,
PolygonName )
    ...
```

See Also

See "INPUT_FIELD LIST and OUTPUT_FIELD_LIST" on page 66 for more information on fields.

GetFieldAttributeX

Retrieves information about AddressBroker fields.

Syntax

```
Integer GetFieldAttributeX (
   String in_FieldName,
   Integer in_FieldIOType,
   Integer in_AttributeID,
   Variant out_AttributeValue )
```

Arguments

in_FieldName

A valid field name listed in the `AllInputFields` or `AllOutputFieldList` property. The property name is not case sensitive, and spaces and underscores are ignored. Do not associate logical names with field names when using this function. *Input*.

in_FieldIOType

Indicates whether field name is an input field—**ABX_FIELD_INPUT** (**1**)—or an output field—**ABX_FIELD_OUTPUT** (**2**). *Input*.

in_AttributeID

The symbolic constant for the attribute value to retrieve. *Input*.

out_AttributeValue

Pointer to the attribute value to be loaded. All values are returned as strings. *Output*.

Return Values

Returns **1** if successful, **0** if unsuccessful.

Prerequisites

`SetFieldX`

Alternates

None.

Notes

`GetFieldAttributeX` retrieves a field attribute's value. These are general attributes, not specific to a record. Attribute values are listed opposite.

Attribute Values

Note: You must work with the numeric values provided in parentheses if you are using a coding environment that does not support enumerated types.

ABX_FIELD_DATA_TYPE (**0**)

"N" (numeric), "C" (character).

ABX_FIELD_DECIMALS (**2**)

Number of decimal places, if numeric.

ABX_FIELD_DESCRIPTION (**5**)

Short (32-character) description of field.

ABX_FIELD_HELP (**6**)

Long (255-character) field description. This is not implemented for most fields.

ABX_FIELD_LENGTH (**1**)

Field width.

ABX_FIELD_NEEDS_LOGICAL_NAME (**5**)

“0” (zero) = No logical name permitted.
“G” = A GeoStan logical name required.
“S” = A Spatial+ logical name required.
“D” = A Demographics Library logical name required.
“C” = A GeoStan Canada logical name required.
“L” = A GDL logical name required.

ABX_FIELD_NUM_VALUES (**3**)
Maximum number of unique values possible for field.

VB Example

```
Dim return_value As Integer
Dim length As String
Dim len As Integer
...
return_value = ab.GetFieldAttributeX ( "City", ABX_FIELD_INPUT,
ABX_FIELD_LENGTH, length )
len = num$ (length)
```

See Also

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information on fields.

GetPropertyX*

Retrieves a property value.

Syntax

```
Integer GetPropertyX (
   String in_PropName,
   Variant out_PropValue )
Integer GetPropertyXBool (
   String in_PropName,
   Variant out_PropValue )
Integer GetPropertyXLong (
   String in_PropName,
   Variant out_PropValue )
```

Arguments

in_PropName

A valid property name. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

out_PropValue

The value of the property given in `in_PropName`. *Output*.

Return Values

Returns **1** if successful, **0** if unsuccessful.

Prerequisites

None.

Alternates

None.

Notes

In addition to using **`GetPropertyX*`** to retrieve AddressBroker property values, many AddressBroker property values can be retrieved via the ActiveX property of the same name using the `value = PropertyName` syntax.

Note: This function manipulates AddressBroker properties only. Using these functions with an ActiveX only property results in error. See "AddressBroker ActiveX properties" on page 318.

VB Example

```
Dim mixedcase
```

```
Dim initlist
Dim return_value As Integer
...
return_value = ab.GetPropertyXBool ( "MIXEDCASE", mixedcase)
return_value = ab.GetPropertyX ( "INITLIST", initlist)
MsgBox initlist
'MsgBox output would show something like
' GEOSTAN \t GEOSTAN_Z9 \t COUNTIES

' This example shows how to get properties in VB using ActiveX
properties

value = ab.InitList
MsgBox InitList
'MsgBox output would show something like
' SAGNET \t GEOSTAN_Z9 \t COUNTIES
```

GetPropertyAttributeX

Retrieves a property attribute.

Syntax

```
Integer GetPropertyAttributeX (
   String in_PropName,
   Integer in_AttributeID,
   Variant out_AttributeValue )
```

Arguments

in_PropName — A valid property name. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

in_AttributeID — The symbolic constant for the attribute value to retrieve. *Input*.

out_AttributeValue — The attribute value to be loaded. *Output*.

Return Values

Returns **1** if successful, **0** if unsuccessful.

Prerequisites

None.

Alternates

None.

Notes

This function manipulates AddressBroker properties only. Using these functions with an ActiveX only property results in error. See “AddressBroker ActiveX properties” on page 318.

Attribute Values

Note: You must work with the numeric values provided in parentheses if you are using a coding environment that does not support enumerated types.

`ABX_PROPERTY_DATA_TYPE ( 5 )`
: “N” (integer), “B” (Boolean), or “C” (string).

`ABX_PROPERTY_DEFAULT_VALUE ( 2 )`
: Default property value. The size of `ABX_PROPERTY_DEFAULT_VALUE` is determined by the value assigned to `ABX_PROPERTY_LENGTH`.

`ABX_PROPERTY_DESCRIPTION ( 1 )`
: Short (100-character) description of property.

`ABX_PROPERTY_ID ( 4 )`
: Property ID.

`ABX_PROPERTY_LENGTH ( 6 )`
: Length of property value.

`ABX_PROPERTY_NAME ( 3 )`
: Property name.

`ABX_PROPERTY_READ_ONLY ( 0 )`
: “1” property is read-only.
“0” property is read/write.

VB Example

```
Dim datatype As String
Dim length_flag As String
Dim return_value As Integer
return_value = ab.GetPropertyAttributeX ( "MIXED CASE",
ABX_PROPERTY_DATA_TYPE, datatype )
return_value = ab.GetPropertyAttributeX ( "INIT_LIST",
ABX_PROPERTY_LENGTH, length )
```

GetRecordX

Advances the pointer to the next record in the output record buffer.

Syntax

```
Integer GetRecordX ()
```

Arguments

None.

Return Values

Returns **1** if successful, **0** if unsuccessful.

Prerequisites

ProcessRecordsX

Alternates

None.

Notes

The first call to **GetRecordX** sets a pointer to the first output record. Subsequent calls advance the pointer. When no more data is found, the return value **1** is returned.

Use the GetFieldX* functions to retrieve record field values. Use the ResetRecordX function to reset the record pointer to the first record.

VB Example

```
Dim AddrLn
Do while ab.GetRecordX ( ) )
    ab.GetFieldX ( "AddressLine",AddrLn )
    ...
Loop
```

GetStatusX

Returns status or error codes and messages.

Syntax

```
Integer GetStatusX (
   Variant out_Status,
   Variant out_StatusMsg )
```

Arguments

out_Status	Status or error code returned. *Output*.
out_StatusMsg	Status or error message returned. *Output*.

Return Values

Returns **1** if successful, **0** if unsuccessful.

Prerequisites

None.

Alternates

None.

Notes

Generally, a 2048-character buffer is sufficient, although the actual message size varies.

InitializeX

initializes the control for processing.

Syntax

```
Integer InitializeX ( )
```

Arguments

None.

Return Values

Returns **1** if successful, **0** if unsuccessful.

Prerequisites

None.

Alternates

None.

Notes

Before calling **InitializeX**, set the “ActiveX only” properties.

Clients require: `HostList`, `TransportProtocol`, `LocalMode`, `UserName`, `Password`, and (optionally) `InitializationFileName`.

See AddressBroker ActiveX properties beginning on page 318 for detailed information about each property.

Set the properties required for a client object. Then call **InitializeX** before calling any other function.

VB Example

```
ab.HostList = "localhost:4660"
ab.TransportProtocol = "socket"
ab.UserName = "MyUserName"
ab.Password = "MyPassword"
return_value = ab.InitializeX ( )
```

LookupRecordX

Processes a single incomplete U.S. address record or performs a reverse lookup on a Canadian postal code.

Syntax

```
Integer LookupRecordX ( )
```

Arguments

None.

Return Values

The `OUTPUT_FIELD_LIST` property defines the fields populated by **`LookupRecordX`**, and the return codes listed below describe the search outcome. Individual codes are returned only when the relevant fields are included in `OUTPUT_FIELD_LIST`. A return value of zero (**0**) indicates an internal failure.

Return Codes

Note: You must work with the numeric values provided in parentheses if you are using a coding environment that does not support enumerated types.

ABX_LOOKUP_ADDRESS_LINE_INCOMPLETE (3)

For a U.S. address, the FirmName or UnitNumber could not be resolved. Multiple incomplete records returned. User can be prompted to submit more information. The most useful fields for resolving a match generally are `FirmName`, `HighUnitNumber`, `LowUnitNumber`, `MatchCode`, and `UnitType`.

Other helpful fields include `AddressLine`, `AddressLine2`, `CarrierRoute`, `CountyName`, `FIPSCountyCode`, `GovernmentBuildingIndicator`, `HighEndHouseNumber`, `LACSAddress`, `LastLine`, `LowEndHouseNumber`, `PostfixDirection`, `PrefixDirection`, `RoadClassCode`, `SegmentBlockLeft`, `SegmentBlockRight`, `State`, `UrbanizationName`, `USPSRangeRecordType`, `ZIP`, `ZIPCarrtSort`, `ZIPCityDelivery`, `ZIPClass`, `ZIPFacility`, and `ZIPUnique`.

For a Canadian postal code, the input Postal Code is resolved to a range of possible addresses that contain a single street number. The street number suffix or unit number values will vary over the range.

ABX_LOOKUP_LAST_LINE_NOT_FOUND (4)

For a U.S. address, multiple incomplete records returned. Did not resolve LastLine. Use iterative calls to GetRecordX to retrieve the possible matches. Only the following output fields are returned: `MatchCode`, `City`, `State`, `ZIP` and `ZIPFacility`. For a Canadian postal code, this return code indicates that the input postal code was not found in the CPC data and is invalid.

ABX_LOOKUP_MULTIPLE_MATCH (**2**)
For a U.S. address, the address resolved to a multiple match. Multiple complete address records returned. Use iterative calls to GetRecordX to retrieve possible matches. For a Canadian postal code, the postal code resolved to a range of possible addresses that vary over the street.

ABX_LOOKUP_NOT_FOUND (**6**)
No records returned, no address matched. Provide a more complete address. (This return code is not used for Canada.)

ABX_LOOKUP_SUCCESS (**1**)
For a U.S. address, a single complete address was matched and returned. For a Canadian postal code, a single address was matched and returned.

ABX_LOOKUP_TOO_MANY_CITIES (**5**)
No records returned. An incomplete LastLine matched over 100 cities. Provide a more complete address. (This return code is not used for Canada.)

Prerequisites

None.

Alternates

SetRecordX

Notes

`LookupRecordX` processes a single input record and should be used only when address information is insufficient for standardization. To process single or multiple records containing complete addresses, use ProcessRecordsX.

Minimally, address information for **`LookupRecordX`** must include a street number, a partial street name, and/or valid LastLine information. For Canada, a valid postal code is required and will return a single address or a range of addresses.

`LookupRecordX` is most useful in interactive programs, when an application may have to make several calls to **`LookupRecordX`** in order to find a match for an incomplete address. In client/server and Internet environments, the record is transferred across the network with each call to `LookupRecordX`. The function call does not return until the record is processed. When `LookupRecordX` processes an address record and fails to find an exact match, it does an extensive search to find cities and streets that are possible matches.

The `INPUT_FIELD_LIST` property specifies the list of fields passed to `LookupRecordX`. Generally, provide at least `FirmName`, `AddressLine` and `LastLine` fields as input to `LookupRecordX`. For Canada, a valid Canadian Postal Code is the only input, and it is set using the PostalCode input field. Only one Postal Code can be processed at a time.

The `OUTPUT_FIELD_LIST` property specifies the list of possible fields returned.

The `MAXIMUM_LOOKUPS` property limits the number of multiples—possible matches—that are returned by `LookupRecordX`. The upper limit of `MAXIMUM_LOOKUPS` is 100. For a Canadian postal code, if the `MAXIMUM_LOOKUPS` is set to 100, the AddressBroker software increases the `MAXIMUM_LOOKUPS` to 200.

Retrieve the list of possible matches using a '`while (GetRecord) do GetField`' loop. No records are returned when the return value of `LookupRecordX` is **ABX_LOOKUP_NOT_FOUND** or **ABX_LOOKUP_TOO_MANY_CITIES**.

Precisely recommends using `ProcessRecordsX` instead of `LookupRecordX`.

VB Example

In an interactive application, a user submits a partial address to `LookupRecordX`. The return code is **ABX_LOOKUP_LAST_LINE_NOT_FOUND**. For a U.S. address, this code indicates that the user did not enter enough information for `LookupRecordX` to resolve the city, state, or ZIP Code. The application prompts the user to select from the list of possible cities and states returned by `LookupRecordX`. The user selects the necessary information and resubmits the address to `LookupRecordX`. For a Canadian postal code, this return code indicates that the input postal code was not found in the CPC data and is invalid.

This time the return code is **ABX_LOOKUP_ADDRESS_LINE_INCOMPLETE**. The user resolved the last line problem, but the return code indicates the address line could be more specific. For a U.S. address, it is missing information on the firm name or unit number (suite, apartment, etc.). The application can prompt the user to select from the list of possibles returned by this call to `LookupRecordX`. The user enters the additional information and resubmits the address to `LookupRecordX`, and **ABX_LOOKUP_SUCCESS** is returned. For a Canadian postal code, the **ABX_LOOKUP_ADDRESS_LINE_INCOMPLETE** code indicates that the input Postal Code resolved to a range of possible addresses that contain a single street number. The street number suffix or unit number values will vary over the range. For example, a Canadian postal code of T3C 2K7 could resolve to 123 A - 123 G Maple Street (when the street suffix varies) or 123 Maple Street Unit 1-100 (when the unit number changes). A valid postal code for one address submitted to `lookupRecord` returns **ABX_LOOKUP_SUCCESS**

When the next address is entered, `LookupRecordX` returns the status code **ABX_LOOKUP_MULTIPLE_MATCH.** This indicates multiple complete matches were found. For a U.S. address, the user may then be prompted to select from the list of possible matches. The selected address is resubmitted to `LookupRecordX` to ensure that it is entirely correct, and that **ABX_LOOKUP_SUCCESS** is returned. For a Canadian postal code, the **ABX_LOOKUP_MULTIPLE_MATCH** code indicates a postal code that resolved to a range of possible addresses that vary over the street. For example, a Canadian postal code could resolve to 100-120 Elm, Calgary, AB or 150-165 Maple, Calgary, AB.

ProcessRecordsX

Processes a set of one or more input records.

Syntax

```
Integer ProcessRecordsX ( )
```

Return Values

Returns **1** if successful, **0** if unsuccessful.

Prerequisites

None.

Alternates

`SetRecordX`

Notes

Each record should contain enough address information for standardization. For records containing incomplete addresses, use LookupRecordX, which progressively returns address choices for one input record at a time.

The function does not return until all of the records are processed.

See Also

See Chapter 13, "Properties" for more information on properties.

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information on fields.

ResetFieldX

Resets the output pointer to the first value of an output field.

Syntax

```
Integer ResetFieldX (
   String in_FieldName,
   String in_LogicalName )
```

Arguments

in_FieldName	A valid field name listed in the `OutputFieldList` property. Spatial+ and Demographic fields require logical names. The logical name may be appended to *in_FieldName* in brackets, or passed in the *in_LogicalName* parameter (see Example). The property name is not case sensitive, and spaces and underscores are ignored. *Input*.
in_LogicalName	The logical name required by the value of *in_FieldName*. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.

Return Values

Returns **1** if successful, **0** if unsuccessful.

Prerequisites

`GetFieldX`

Alternates

None.

Notes

`ResetFieldX` returns **0** when, for any reason, *in_FieldName* is not found.

All Spatial+, GDL, and Demographic fields require logical names. GeoStan and GeoStan Canada fields do not.

VB Example

```
//Example using field name with its logical name in brackets.
Dim PolygonName
Dim return_value As Integer
Do while ab.GetFieldX ( "PolygonName[COUNTIES]", PolygonName
   ...
Loop
```

```
return_value = ab.ResetFieldX ( "PolygonName", COUNTIES")
```

See Also

See "INPUT_FIELD LIST and OUTPUT_FIELD_LIST" on page 66 for more information on fields.

ResetRecordX

Resets output record pointer to the first record in the output record buffer.

Syntax

```
Integer ResetRecordX ( )
```

Arguments

None.

Return Values

Returns **1** if successful, **0** if unsuccessful.

Prerequisites

GetFieldX

Alternates

None.

SetFieldX

Sets an input field value in the current input record.

Syntax

```
Integer SetFieldX (
   String in_FieldName,
   String in_FieldValue )
```

Arguments

in_FieldName	A valid field name listed in the `InputFieldList` property. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.
in_FieldValue	The string value to assign to the field. Maximum string length is 256 characters. *Input*.

Return Values

Returns **1** if successful, **0** if unsuccessful.

Prerequisites

`SetPropertyX`

Alternates

None.

Notes

Reserved characters: The `RecordDelimiter`, `FieldDelimiter`, and `ValueDelimiter` properties have default values of line feed, tab, and CTRL-A, respectively.

- If your data contains any of these characters, you **must** reset the associated property to a different character.
- Your data **must not** contain the null character.

For example, a resetting of the ValueDelimiter property in the AddressBroker server initialization file might appear as follows:

```
VALUEDELIMITER = 2
```

This would reset the value delimiter from CTRL-A (ASCII decimal 1 or “start of heading”) to CTRL-B (ASCII decimal 2 or “start of text”).

VB Example

```
Dim addressline As String
Dim lastline As String
Dim return_value As Integer
 ...
addressline = “2900 Center Green Court”
lastline = “Boulder Colorado”
ab.SetFieldX ( “AddressLine”, addressline )
ab.SetFieldX ( “LastLine”, lastline )
```

See Also

See “INPUT_FIELD LIST and OUTPUT_FIELD_LIST” on page 66 for more information on fields.

SetPropertyX*

Assign a property value.

Syntax

```
Integer SetPropertyX (
   String in_PropName,
   String in_PropValue )
Integer SetPropertyXBool (
   String in_PropName,
   Integer in_PropValue )
Integer SetPropertyXLong (
   String in_PropName,
   Long in_PropValue )
```

Arguments

in_PropName	A valid property name. The property name is not case sensitive, and spaces and underscores are ignored. *Input*.
in_PropValue	The value to assign to the property. *Input*.

Return Values

Returns **1** if successful, **0** if unsuccessful.

Prerequisites

InitializeX

Alternates

None.

Notes

The specific **`SetPropertyX*`** function to use depends on the data type of the property you are setting.

Note: This function manipulates AddressBroker properties only. Using these functions with an "ActiveX only" property results in error. See "AddressBroker ActiveX properties" on page 318.

The AddressBroker ActiveX interface supports two ways of setting most AddressBroker properties:

- with the **`SetPropertyX*`** function, described here.
- with ActiveX properties using the `PropertyName=value` syntax. See "AddressBroker ActiveX properties" on page 318. See the Quick reference section on page 318 for a complete list of AddressBroker properties that can be set using ActiveX properties.

VB Example 1

```
' This example shows how to set properties in VB using functions
' VB requires you to specify the return value, even if unused.

result = ab.SetPropertyX("INIT_LIST", "GEOSTAN|GEOSTAN_Z9|Counties")
result = ab.SetPropertyXBool("MIXED CASE", True)
' Set enumerated values using the Property ID or the equivalent value
result = ab.SetPropertyXLong("INPUT MODE", 0)
result = ab.SetPropertyXLong("INPUT MODE", ABX_INPUT_NORMAL)
```

VB Example 2

```
' This example shows how to set properties in VB using ActiveX
properties

ab.InitList = "GEOSTAN|GEOSTAN_Z9|Counties"
ab.MixedCase = True
' Set enumerated values using the Property ID or the equivalent value
ab.InputMode = ABX_INPUT_NORMAL
ab.InputMode = 0
```

SetRecordX

Adds data for the current record to the input record buffer and advances the input record pointer to the next empty record in the buffer.

Syntax

```
Integer SetRecordX ( )
```

Arguments

None.

Return Values

Returns **1** if successful, **0** if unsuccessful.

Prerequisites

`SetFieldX`

Alternates

None.

ValidatePropertiesX

Validates properties for consistency and completeness.

Syntax

`Integer` **`ValidatePropertiesX`** `( )`

Arguments

None.

Return Values

Returns **1** if successful, **0** if unsuccessful.

Prerequisites

`SetPropertyX`

Alternates

None.

Notes

The **`ValidatePropertiesX`** function verifies the values of initialization and processing control properties to ensure a complete and compatible set of values are available to AddressBroker. Call this function after one or more AddressBroker properties have been set and before calling SetFieldX or any processing functions.

When **`ValidatePropertiesX`** returns **1**, it indicates all properties have been successfully validated and that AddressBroker is ready to process records. In some cases, all properties can be validated in a single function call.

See Also

See Chapter 13, "Properties" for more information on properties.

AddressBroker ActiveX properties

This section describes in detail the properties available through the AddressBroker ActiveX component. The ActiveX properties have a 1:1 naming correspondence with AddressBroker properties (unless otherwise noted). For information about AddressBroker properties, see , “Properties”.

This section also describes a small set of non-AddressBroker properties—that is, properties specific to the AddressBroker ActiveX interface. These have been identified throughout as “ActiveX only”. ActiveX-only properties cannot be manipulated programmatically with the SetPropertyX* or GetPropertyX* functions.

Setting and validating AddressBroker properties

The AddressBroker ActiveX interface supports two ways of setting most AddressBroker properties:

- with the SetPropertyX* function. All AddressBroker properties can be set using `SetPropertyX*`.
- with ActiveX properties. The syntax for using ActiveX properties is a simple “name = value” statement. See the Quick reference section, next, for a complete list of AddressBroker properties that can be set this way.

AddressBroker property values are invalid until InitializeX has been successfully called and ValidatePropertiesX has been called.

Setting and validating ActiveX only properties

The ActiveX only properties are set using a “name = value” statement. These properties must be assigned values before InitializeX is invoked.

Quick reference: properties

ActiveX only properties

The properties listed here are true ActiveX properties and have all of the characteristics generally attributed to ActiveX properties. ActiveX only properties cannot be manipulated programmatically with AddressBroker ActiveX functions.

Set the ActiveX only properties as the first step in all applications using the ActiveX component.

HostList	LogFileName	TransportProtocol
InitializationFileName	Password	UserName

QMSActiveXv1 class properties

The AddressBroker ActiveX component has one class. All of the properties described in this section are members of this class.

AddressPreference	AllInputFields
AllOutputFields	BufferRadius
BufferRadiusTable	CacheSize
CarrtProcessed	CentroidPreference
CentroidPreference	DataType
Datum	DaysRemaining
DpbcProcessed	FieldDelimiter
FileDate	GeoRecordTotal
InitList	InputFieldList
InputMode	KeepCounts
KeepMultimatch	LogicalNames
MatchMode	MaximumLookups
MaximumPoints	MaximumPolygons
MiscCounts	MixedCase
OffsetDistance	OutputFieldList
RecordDelimiter	RecordsMatched
RecordsProcessed	RecordsRemaining
Timeout	ValueDelimiter
Version	Z4ChangeDate
Zip4Processed	Zip4Skipped
ZipProcessed	ApproxPbKey

AddressPreference

Sets address preference when a single record contains more than one address.

Data type

`Integer.`

Notes

Valid values are:

ABX_ADDRESS_BOTTOM	1 (Default)
ABX_ADDRESS_POBOX	2
ABX_ADDRESS_STREET	3

See "Address preference" on page 450.

AllInputFields

Retrieves a list of all valid input field names. Based on the value assigned to the `InitList` property.

Data Type

`String.`

Notes

Read-only.

The list of available input fields depends upon values set in several other AddressBroker properties including `InitList`, `InputMode`, `GEOSTAN_PATHS`, `GEOSTAN_Z9_PATHS`, `GEOSTAN_Z5_PATHS`, `GEOSTAN_CANADA_PATHS`, `SPATIAL_PATHS`, and `DEMOGRAPHICS_PATHS`, properties.

AllOutputFields

Retrieves a list of all valid output field names. Based on the value of the InitList property.

Data Type

`String.`

Notes

Read-only.

The list of available output fields depends upon values set in several other properties including `InitList`, `InputMode`, `GEOSTAN_PATHS`, `GEOSTAN_Z9_PATHS`, `GEOSTAN_Z5_PATHS`, `GEOSTAN_CANADA_PATHS`, `SPATIAL_PATHS`, and `DEMOGRAPHICS_PATHS` properties.

BufferRadius

Sets the spatial buffer radius (or width), in feet, to apply to the features in a polygon (spatial) search.

Data Type

`Long.`

Notes

The default value is 0 feet. Range = 0 - 5280000.

BufferRadiusTable

Sets the list of Spatial+ buffer radius entries.

Syntax

```
ab.BufferRadiusTable = "Value"
where Value = location code:buffer radius[LOGICAL NAME] | location
code:buffer radius[LOGICAL NAME]...
or where Value = location code:buffer radius[LOGICAL NAME] \t location
code:buffer radius[LOGICAL NAME]...
```

Data Type

`String.`

Notes

This property is a delimited list. Each item in the list consists of three elements. The first element is a location quality code (specified fully or with a wild card character) followed by a colon (:). The second element is the radius buffer (in feet). The last element, in brackets, is the logical name of a Spatial+ data file. The logical name must be specified in the `SPATIAL_PATHS` server property.

There are two properties that specify the buffer radius for spatial analysis: `BufferRadius` and `BufferRadiusTable`. AddressBroker uses the value assigned to `BufferRadius` for the general case.

`BufferRadiusTable` lets you specify the radius to use based on the `LocationQualityCode` output field value of an individual record.

You can use `BufferRadiusTable` without listing `LocationQualityCode` in the `OutputFieldList` property.

For example, a table entry of:

AS0:50[FLOODPLAIN]

specifies that when AddressBroker does a spatial analysis on addresses with the location code “ASO”, a buffer radius of 50 feet be used with the FLOODPLAIN data.

To minimize the number of `BUFFER_RADIUS_TABLE` entries, you can use the star (*) character as a wild card to replace the trailing end of a location code. For example, a table entry of:

A*:1000[COUNTIES]

indicates that when AddressBroker does a spatial analysis on addresses with a location code starting with “A” followed by any other value, a buffer radius of 1000 feet be used with the COUNTIES data.

The match algorithm for `BufferRadiusTable` is a linear left-to-right search. That is, the first entry in the buffer radius table to match the location code is the one used. This is particularly important to note when using wild cards.

The most specific table entries should be first (left-most) in the table. The most general entries should be toward the end (right-most) of the table. For example:

AS0:10[COUNTIES] | A*:1000[COUNTIES]

specifies that when AddressBroker does a spatial analysis on addresses with a “best” location quality code (“AS0”) a buffer radius of 10 feet be used with the COUNTIES data. However, the spatial analysis of addresses with more general location quality codes (A*) is done with a radius buffer of 1000 feet.

If these two `BufferRadiusTable` entries were reversed, the “AS0:10[COUNTIES]” would never be applied, as “A*:1000[COUNTIES]” is the more general match. When making `BufferRadiusTable` entries, it is important to specify location codes and order the entries carefully for your particular needs.

If no `BufferRadusTable` entry matches the location code assigned to an address, the value assigned to `BufferRadius` is used.

The ValidatePropertiesX function can only validate the syntax of your entries.

Example

```
BUFFER_RADIUS       = 50
BUFFER_RADIUS_TABLE = AS0:100[COUNTIES] | AS1:200[COUNTIES] |
A*:1000[COUNTIES]
```

See Also

“LogicalNames” on page 329.

“GeoStan location codes” on page 433

CacheSize

Sets the size of caching polygons.

Data Type

```
Integer
```

Notes

Valid values are:

ABX_CACHE_SIZE_NONE	1
ABX_CACHE_SIZE_MEDIUM	2 (Default).
ABX_CACHE_SIZE_LARGE	3

CarrtProcessed

Retrieves the number of processed records returned that were assigned Carrier Routes.

Data Type

`Long.`

Notes

Read-only. `KeepCounts` must be set to **TRUE** and `KeepMultimatch` must be set to **FALSE** for counts to be meaningful.

CentroidPreference

Sets Centroid preference.

Data Type

`Integer.`

Notes

Valid values are:

ABX_CENTROID_NONE	1
ABX_CENTROID_ADDRESS_UNAVAILABLE	2 (Default).
ABX_CENTROID_NO_ADDRESS	3

DataType

Retrieves GeoStan data types.

Data Type

`Integer.`

Notes

Read-only. Valid values are:

ABX_DATA_TYPE_USPS	0
ABX_DATA_TYPE_TIGER	1
ABX_DATA_TYPE_TOMTOM	2
ABX_DATA_TYPE_SANBORN_POINT	3

ABX_DATA_TYPE_TELE_ATLAS	4
ABX_DATA_TYPE_GEOSYS Deprecated	5
ABX_DATA_TYPE_NAVTEQ	6
ABX_DATA_TYPE_TOMTOM_POINT	7
ABX_DATA_TYPE_CENTRUS_POINT	8
ABX_DATA_TYPE_AUXILIARY	9
ABX_DATA_TYPE_USER_DICTIONARY	10
ABX_DATA_TYPE_NAVTEQ_POINT	11
ABX_DATA_TYPE_MASTER_LOCATION	12

Datum

Sets the GeoStan datum.

Data Type

`Integer.`

Notes

Valid values are:

ABX_DATUM_NAD27	1
ABX_DATUM_NAD83	2 (Default).

DaysRemaining

Retrieves the number of days remaining before license expiration.

Data Type

`Long.`

Notes

Read-only. A value of **ABX_LICENSE_UNLIMITED** indicates there is no license-based time limit.

DpbcProcessed

Retrieves the number of processed records returned that were assigned Delivery Point Bar Codes.

Data Type

`Long.`

Notes

Read-only. `KEEP_COUNTS` must be set to **TRUE** and `KEEP_MULTIMATCH` must be set to **FALSE** for counts to be meaningful.

FieldDelimiter

Delimits fields.

Data Type

`Long.`

Notes

The default value is 9 (ASCII value for TAB).

FileDate

Retrieves the publish date of GSD data.

Data Type

`String.`

Notes

Read-only.

GeoRecordTotal

Retrieves the total number of records geocoded with the current license.

Data Type

`Long.`

Notes

Read-only.

HostList “ActiveX only”

A list of host names. Use in client applications only.

Data Type

```
String.
```

Notes

A delimited list of host names (for more information, see “Using multiple servers” on page 90. This is not an AddressBroker property; it is specific to the AddressBroker ActiveX interface. Use in clients only. The default value is “localhost:4660”. Set this property before calling InitializeX.

Example

```
Socket protocol
ab.HostList = “primary:1234 | secondary:1235”
ab.HostList = “centrus.com:1234 | centrus-software.com:1235”
ab.HostList = “204.180.129.200:1234 | 209.38.36.44:1235”
```

InitializationFileName “ActiveX only”

Sets the (optional) initialization file name.

Data Type

```
String.
```

Notes

This is not an AddressBroker property; it is specific to the AddressBroker ActiveX interface. The default value is a null string. You must set this property before calling InitializeX.

InitList

Sets a list of logical names.

Data Type

```
String.
```

Notes

This property is a tab- (\t) or pipe- (|) delimited list of logical names referencing AddressBroker geo-demographic data files to use in your application. Logical names are defined in AddressBroker's `GEOSTAN_PATHS`, `GEOSTAN_Z9_PATHS`, `SPATIAL_PATHS`, `DEMOGRAPHICS_PATHS`, and `GEOSTAN_CANADA_PATHS` properties. See "INIT_LIST Property" on page 374.

When setting `InitList`, assign only the logical names of the geo-demographic data your application accesses. Be sure that the GeoStan and GeoStan ZIP9 data you assign are compatible.

InputFieldList

Sets a list of input field names.

Data Type

`String.`

Notes

The `InputFieldList` property is a delimited list of field names your application uses as input. To find out which input field names you can assign to `InputFieldList`, examine the `AllInputFields` as an argument. See "INPUT_FIELD_LIST Property" on page 375.

By specifying only those fields the application uses (as opposed to all of the fields in your data), AddressBroker manages memory more efficiently, and optimally transfers data across the network in client/server applications.

InputMode

Sets the input mode to parsed, two-line, multiline, or parsed lastline.

Data Type

`Integer.`

Notes

Valid values are:

ABX_INPUT_NORMAL	1 (Default).
ABX_INPUT_MULTILINE	2
ABX_INPUT_PARSED	3
ABX_INPUT_PARSED_LASTLINE	4

KeepCounts

If true, save count match and location codes; otherwise do not save counts.

Data Type

`Boolean.`

Notes

The default value is **FALSE**. `KEEP_COUNTS` must be set to **TRUE** and `KEEP_MULTIMATCH` must be set to **FALSE** for counts to be meaningful.

KeepMultimatch

If true, output all matches; otherwise output a single record only.

Data Type

`Boolean.`

Notes

The default value is **TRUE**.

LogFileName “ActiveX only”

Specifies a file to use for error messages.

Data Type

`String.`

Notes

This is not an AddressBroker property; it is specific to the AddressBroker ActiveX interface. Defaults to **ab.log**. Set this property before calling InitializeX.

LogicalNames

Retrieves a list of all valid logical names.

Data Type

`String.`

Notes

Read-only.

The `LogicalNames` read-only property is a tab- (\t) or pipe- (|) delimited list of all logical names defined in the `GEOSTAN_PATHS`, `GEOSTAN_Z9_PATHS`, `SPATIAL_PATHS` and `DEMOGRAPHICS_PATHS` properties. Each item in the list consists of three elements. The first element is the logical name. It is followed by a colon (**:**). The last element is an alphabetic code indicating the type of data file associated with the logical name:

- **G**—GeoStan
- **D**—Demographics
- **S**— Spatial+
- **Z**—GeoStan ZIP9
- **C**— GeoStan Canada
- **L**—GDL

The `LogicalNames` property is particularly useful when the logical names are unknown in advance. This property lets you query the server for a list of logical names at run time.

MatchMode

Specifies a match strategy for ProcessRecords.

Data Type

`Integer.`

Notes

Valid values are:

ABX_MODE_EXACT	1
ABX_MODE_CLOSE	2
ABX_MODE_RELAX	3 (Default).
ABX_MODE_CASS	4
ABX_MODE_INTERACTIVE	5

MaximumLookups

Sets a maximum number of matched `LookupRecord` fields.

Data Type

`Long.`

Notes

The default value is ten lookups. Range = **1 - 100,000**.

MaximumPoints

Sets a maximum number of points to match in a Closest Site search.

Data Type

`Long.`

Notes

The default value is four points. Range = **1 - 100,000**.

MaximumPolygons

Sets a maximum number of polygons to match in a Point in Polygon search.

Data Type

`Long.`

Notes

The default value is four polygons. Range = **1 - 100**.

MiscCounts

Retrieves miscellaneous statistics about records processed.

Data Type

`String.`

Notes

Read-only. This property contains a tab- (\t) or pipe- (|) delimited list of miscellaneous counters and their values. Each item in the list consists of three elements: the counter label, a colon, and a numeric count. The list contains counts for all counter labels. Figure 21 provides a complete listing of counter labels.

`KeepCounts` must be set to **TRUE** and `KeepMultimatch` must be set to **FALSE** for counts to be meaningful. Misc_Counts counter labels by type.

Successful match codes	Location codes	Error match codes
standardized and matched records	address-level geocodes	address not found
intersection matched records	ZIP + 4 centroid level geocodes	low-level error
non-USPS matched records	block group accuracy geocodes	GSD file not found error
address lines corrected	census tract accuracy geocodes	incorrect GSD file signature or version ID error
street types corrected	county-level accuracy geocodes	GSD file out of date error
pre-directionals corrected	geocodes based on 5-digit ZIP centroid	city + state or ZIP not found error
post-directionals corrected	geocodes based on ZIP+2 centroid	input ZIP not found in directory error
street names corrected	geocodes based on ZIP + 4 centroid	input city not found in directory error
last lines corrected		input city not unique in directory error
ZIPs corrected		out of license area error
cities corrected		license expired error
states corrected		matching street not found in directory error
ZIP + 4s corrected		matching cross street not found for intersection match error
		matching ranges not found error
		unresolved match error
		too many possible cross streets for intersection match error
		address not found in multiline match error

Counts are returned in top-down left-to-right order, as listed in the table above.

MixedCase

If true, use mixed case; otherwise use all upper case.

Data Type

`Boolean.`

Notes

The default value is **FALSE**.

OffsetDistance

Sets an offset distance (in feet) to use when geocoding.

Data Type

```
Long.
```

Notes

The default value is fifty feet. Range = **0 - 5280**.

OutputFieldList

Sets a list of output field names to be returned.

Syntax

```
ab.OutputFieldList = “Value”
```

where, for fields that reference to GeoStan data,

```
Value = FieldName | FieldName | …
or Value = FieldName \t FieldName \t …for fields
```

and where, for fields that reference to Spatial+ or Demographics Library data,

```
Value = FieldName [Logical Name] | FieldName [Logical Name] | …
or Value = FieldName [Logical Name] \t FieldName [Logical Name] \t....
```

Data Type

```
String.
```

Notes

This property is a delimited list of field names to be retrieved by the application.

When assigning a list of output fields, you must append a logical name, in square brackets ([ ]), to each field name that requires reference to Spatial+ or Demographics Library data. The logical name establishes the data source your application uses to generate these output field values. See “OUTPUT_FIELD_LIST Property” on page 384.

By specifying a subset of output fields to retrieve (as opposed to all of the possible output fields AddressBroker can generate given your input), AddressBroker manages memory more efficiently, and optimally transfers data across the network in client/server applications.

Password "Activex Only"

Specifies a password to use when logging on to an AddressBroker server. Client applications only.

Data Type

`String.`

Notes

This is not an AddressBroker property; it is specific to the AddressBroker ActiveX interface. Used by clients only. This property must be set (for client applications) before calling **`InitializeX`**.

RecordDelimiter

Delimits records.

Data Type

`Long.`

Notes

The default value is 10 (ASCII value for line feed).

Recordsmatched

Retrieves the number of matched records returned.

Data Type

`Long.`

Notes

Read-only. `KeepCounts` must be set to **TRUE** and `KeepMultimatch` must be set to **FALSE** for counts to be meaningful.

RecordsProcessed

Retrieves the number of processed records returned.

Data Type

`Long.`

Notes

Read-only.

RecordsRemaining

Retrieves the number of records that can be processed before license expiration.

Data Type

`Long.`

Notes

Read-only. A value of **ABX_LICENSE_UNLIMITED** indicates there is no license-based record limit.

Timeout

Sets the Client time-out in seconds.

Data Type

`String.`

Notes

The default value is ten seconds.

TransportProtocol “ActiveX only”

Specifies a transport protocol to use. Client applications only.

Data Type

`String.`

Notes

The valid value for this property is “socket”. This property is not an AddressBroker property; it is specific to the AddressBroker ActiveX interface. Case-insensitive string that specifies the network protocol AddressBroker uses. Used by clients only. Set this property before calling InitializeX.

UserName “ActiveX only”

Specifies a user name to use when logging on to the AddressBroker server. Client applications only.

Data Type

`String.`

Notes

This property is not an AddressBroker property; it is specific to the AddressBroker ActiveX interface. Used by clients only. Set this property before calling InitializeX.

ValueDelimiter

Delimits values in multi-value fields.

Data Type

`Long.`

Notes

The default value is 1 (ASCII value for CTRL-A).

Version

Retrieves the AddressBroker version.

Data Type

`String.`

Notes

Read-only.

Z4ChangeDate

Indicates a request for address change information after the date specified.

Data Type

`String.`

Notes

The ZIP* input fields must also be set per record in order for this request to be fulfilled. If the ZIP + 4 of an input record is unchanged for the time period, no corresponding output record is calculated or returned. Use MMYYYY format to specify the date.

Zip4Processed

Retrieves the number of processed records returned that were assigned ZIP + 4.

Data Type

`Long.`

Notes

Read-only. `KeepCounts` must be set to **TRUE** and `KeepMultimatch` must be set to **FALSE** for counts to be meaningful.

Zip4Skipped

Retrieves the number of records skipped when using `Z4_CHANGE_DATE`.

Data Type

`Long.`

Notes

Read-only. `KeepCounts` must be set to **TRUE** and `KeepMultimatch` must be set to **FALSE** for counts to be meaningful.

ZipProcessed

Retrieves the number of processed records returned that were assigned a 5-digit ZIP.

Data Type

`Long.`

Notes

Read-only. `KeepCounts` must be set to **TRUE** and `KeepMultimatch` must be set to **FALSE** for counts to be meaningful.

Errors, messages, and status logs

There are no errors, messages, or logging specific to the AddressBroker ActiveX client. All errors and messages are currently logged by the AddressBroker server.

13 – Properties

In this chapter

This chapter provides information about an import utility that helps you retrieve attribute information.

The chapter also contains a complete listing of AddressBroker properties. The tables list each property by character string name. The tables also list each property's corresponding property ID, the property's data type, the AddressBroker class in which it is used, status, and a brief description including the default value (if any).

Some properties have a set of pre-defined values. Refer to "Pre-defined property values" on page 354 for a complete list of these values.

The discussion about properties is organized into the following types:

- Initialization properties – Initialize AddressBroker.
- Processing control properties – Configure application processing.
- Read-only output properties – Report on AddressBroker's status and processing statistics.

Using Spatial Import

AddressBroker allows you to use an import utility located in the `\AddressBroker\bin` directory to retrieve attribute information. You must create a Spatial+ object (GSB) file and associate it with a GSA attribute file. These synchronized GSA and GSB files allow you to retrieve an unlimited amount of attribute information that is not currently available from the Name and Name2 fields. For instructions on using the Spatial+ import utility, see the *Spatial+ Reference Manual*.

SpatialImport does not process special characters. All characters between 31 and 127 on the ASCII standard code page are valid. Other characters are not supported and causes unpredictable behavior for attribute (GSA) data. Examples are:

- Spanish n with tilde: ñ
- Long Dash: -
- Reverse Quote: '
- Copyright: ©

Additional information

Detailed descriptions of several properties are given in the next chapter, "Properties descriptions."

Initialization properties

String Property Name	Property ID AB_* = C/C++/.NET/Java Property ID ABX_* = ActiveX	Data Type	Status	Description
"TIMEOUT"	AB_TIMEOUT	String	Server only	Client time-out in seconds. **Default** = 10.
"INIT_LIST" See page 374.	AB_INIT_LIST	String	Server only	Delimited list of logical names to be used. This property must be set before validating properties.
"LICENSE_PATH"	AB_LICENSE_PATH	String	Server only	Path and file name of license file.
"LICENSE_KEY"	AB_LICENSE_KEY	String	Server only	License key.
"STATUS_LOG"	AB_STATUS_LOG	String	Server only	Path and filename of status log file or for "CONSOLE" to display to screen, "EVENTLOG" to send to event log (on Windows systems) or syslog (on UNIX systems.) Default = console.
"REQUEST_LOG"	AB_REQUEST_LOG	String	Server only	Path and filename of the request log file, which contains a summary of each request sent to the server.
"REQUEST_LOG_OPTIONS"	AB_REQUEST_LOG_OPTIONS	String	Server only	Modifies REQUEST_LOG. Specifies the format of the request log and the delimiter that separates fields.
"LOG_ROLLOVER"	AB_LOG_ROLLOVER	Long	Server only	Sets age and size criteria for the status and request log files for the periodic rollover of file names. Default=NORMAL
"IP_FILTER"	AB_IP_FILTER	String	Server only	Allows or denies IP addresses access to the server.
"CLOSEST_SITE_FILTER"	AB_CLOSEST_SITE_FILTER	String	Client only	Limits the number of returned ClosestSite records by a user-specified filter. Value=filter criteria

String Property Name	Property ID AB_* = C/C++/.NET/Java Property ID ABX_* = ActiveX	Data Type	Status	Description
"STATUS_LEVEL"	AB_STATUS_LEVEL	String	Server only	Not supported in the Java API. Sets the type of error and status messages returned. FATAL—fatal errors, errors and warnings. ERROR—errors and warnings only. WARN— warnings only. INFO—all informational messages. NONE—none. DEBUG—status messages, development only. SERVER—returns server level debug messages (default).
"GEOSTAN_PATHS" See page 371.	AB_GEOSTAN_PATHS	String	Server only	Logical names, path and directory names of GeoStan data. This property must be set before validating properties in server implementations.
"GEOSTAN_Z9_PATHS" See page 371.	AB_GEOSTAN_Z9_PATHS	String	Server only	Logical names, path and file name of GeoStan ZIP + 4 data. This property must be set before validating properties in server implementations. Server only.
"GEOSTAN_Z5_PATHS"	AB_GEOSTAN_Z5_PATHS	String	Server only	Logical names, path and file name of GeoStan ZIP data. This property must be set before validating properties in server implementations.Server only.
"GEOSTAN_CANADA_PATHS"	AB_GEOSTAN_CANADA_PATH	String	Server only	Logical names, path and directory name of GeoStan Canada data. This property must be set before validating properties in server implementations. Server only.
"GEOSTAN_TMP_PATH"	AB_GEOSTAN_TMP_PATH	String	Server only	Logical names, path and directory name of a temporary directory for GeoStan data to be used with GDL. This property must be set before validating properties in server implementations.
"DEMOGRAPHICS_PATHS" See page 368.	AB_DEMOGRAPHICS_PATHS	String	Server only	Logical names, path and file name of DemoLib data. This property must be set before validating properties in server implementations. Server only.

String Property Name	Property ID AB_* = C/C++/.NET/Java Property ID ABX_* = ActiveX	Data Type	Status	Description
"GDL_SPATIAL_PATHS"	AB_GDL_SPATIAL_PATHS	String	Server only	DEPRECATED. Logical names, path and file name of Geographic Determination Library data. This property must be set before validating properties in server implementations.Server only.
"SPATIAL_PATHS"	AB_SPATIAL_PATHS	String	Server only	Logical names, path and file name of Spatial+ data. This property must be set before validating properties in server implementations. Server only. **NOTE:** You can have multiple SPATIAL_PATHS instances in your file. These instances are additive: AddressBroker uses the values listed in each instance.
"CACHE_SIZE"	AB_CACHE_SIZE	32-bit integer	Server only	Size of caching polygons. See "Pre-defined Property Values" table, page 354.
"RECORD_DELIMITER"	AB_RECORD_DELIMITER	32-bit integer	Client and Server	Delimits records. **Default** = LF (line feed).
"FIELD_DELIMITER"	AB_FIELD_DELIMITER	32-bit integer	Client and Server	Delimits fields. **Default** = TAB.
"VALUE_DELIMITER"	AB_VALUE_DELIMITER	32-bit integer	Client and Server	Delimits values in multi-value fields. **Default** = CTRL-A.
"HOTSWAP_DIRECTORY"	AB_HOTSWAP_DIRECTORY	String	Server only	Path and name of the directory where the server administrator places the GSB files that AddressBroker loads for hot swap data files.
"WORKING_DIRECTORY"	AB_WORKING_DIRECTORY	String	Server only	Path and name of the directory where the server holds GSB files that the server is currently processing.
"DISCARD_DIRECTORY"	AB_DISCARD_DIRECTORY	String	Server only	Path and name of the directory where the server places old versions of the GSB files.
"ERROR_DIRECTORY"	AB_ERROR_DIRECTORY	String	Server only	Path and name of the directory where the server places GSB files that failed verification.
"POLLING_TIME"	AB_POLLING_TIME	String	Client only	Time interval, in seconds, between successive polls of the hot swap directory. Valid range is 1 to 86400 seconds, with a default of 10.

String Property Name	Property ID AB_* = C/C++/.NET/Java Property ID ABX_* = ActiveX	Data Type	Status	Description
"MAX_OPEN_GSBS" See page 381.	AB_MAX_OPEN_GSBS	Long	Server only	Maximum number of open GSB files (1-4096). Default=0. For Linux systems only.
"GS_MEMORY_LIMIT" See page 373.	AB_GSMEM_LIMIT	Long	Server only	**NOTE:** This property only applies to 64-bit applications. For 32-bit applications, data files are not memory-mapped and attempts to set this property will be ignored. When AddressBroker is initialized, it will memory-map as many data files into memory as the GS_MEMORY_LIMIT allows. 0- 256000 megabytes. Default=16000 megabytes.

Processing control properties

String Property Name	Property ID AB_* = C/C++/.NET/Java Property ID ABX_* = ActiveX	Data Type	Description
"ADDR_POINT_INTERP"	AB_ADDR_POINT_INTERP	Boolean	True turns on address point interpolation in GeoStan. False turns it off.
"ALTERNATE_LOOKUP"	AB_ALTERNATE_LOOKUP	Boolean	True sets find property GS_FIND_ALTERNATE_LOOKUP to true in GeoStan. False turns it off. Sets values for GS_FIND_ALTERNATE_LOOKUP. The values are: • 0 is GS_PREFER_UNDEFINED - undefined. • 1 is GS_PREFER_STREET_LOOKUP - Matches to the address line, if a match is not made, then GeoStan matches to the Firm name line. • 2 is GS_PREFER_FIRM_LOOKUP - matches to the Firm name line, if a match is not made, then GeoStan matches to address line. • 3 is GS_STREET_LOOKUP_ONLY - default value if GS_FIND_ALTERNATE_LOOKUP not in the list.
"DPV_DATA_ACCESS"	AB_DPV_DATA_ACCESS	Long	DPV data access options are 1-4, see the following: • 1 = DPV full data loaded in buffered memory • 2 = DPV full data loaded completely into memory • 3 = DPV split data loaded in buffered memory • 4 = DPV flat data loaded completely into memory
"FIRST_LETTER_EXPANDED"	AB_FIRST_LETTER_EXPANDED	Boolean	True sets find property GS_FIND_FIRST_LETTER_EXPANDED to true in GeoStan. False turns it off.
"MUST_MATCH_ADDR_NUM"	AB_MUST_MATCH_ADDR_NUM	Boolean	True sets find property GS_FIND_MUST_MATCH_ADDRNUM to true in GeoStan. False turns it off. Usable match modes: Custom
"ADDRESS_PREFERENCE"	AB_ADDRESS_PREFERENCE	32-bit integer	Address preference. See the table "Pre-defined property values" on page 354.

String Property Name	Property ID AB_* = C/C++/.NET/Java Property ID ABX_* = ActiveX	Data Type	Description
"ALWAYS_FIND_CANDIDATES"	AB_ALWAYSFINDCANDIDATES	Boolean	Enables AddressBroker to keep multiple candidate records when matching with point-level data for use with centerline matching. Used to return multiple candidate records when street locator matching is enabled. Additional information can be obtained about matching street segments for both a single or multiple match. Not valid when using the reverse geocoding options. TRUE = Keep candidates FALSE = **Default.** Do not keep candidates
"APN_DATA"	AB_APN_DATA	Boolean	TRUE = Load and use Centrus Points APN data FALSE = Don't load Centrus Points APN data
"APPROXPBKEY"	AB_APPROX_PBKEY	Boolean	When using the Master Location Dataset (MLD), when a match is not made to an MLD record, this feature returns the pbKey of the nearest MLD point location. The search radius for the nearest MLD point location can be configured to 0-5280 feet. The default is 150 feet. This type of match returns a pbKey with a leading 'X' rather than a 'P', for example, X00001XSF1IF. For more information, see "PreciselyID Fallback" on page 20. TRUE = Enables PBKey Fallback. FALSE = Disables PBKey Fallback.(**default**)
"BUFFER_RADIUS_TABLE"	AB_BUFFER_RADIUS_TABLE	String	Table of location codes vs. Spatial buffer radii or widths Overrides BUFFER_RADIUS.
"BUFFER_RADIUS"	AB_BUFFER_RADIUS	32-bit integer	Spatial buffer (radius or width) in feet to apply to features in the object file. **Default** = 0; range = 0 - 5280000.
"BUILDING_SEARCH"	AB_BUILDING_SEARCH	Boolean	TRUE = Enables matching to building and business names entered in the address line. FALSE = Disables matching to building and business names entered in the address line (**default**).
"CENTERLINE_OFFSET"	AB_CENTERLINE_OFFSET	32-bit integer	Distance, in feet, to offset the centerline geocode from the street centerline toward the parcel centroid. **Default** is 0 feet, which returns the street centerline geocode. Any value which takes the geocode past the parcel centroid will return the parcel centroid. Range = 0 - 5280.

String Property Name	Property ID AB_* = C/C++/.NET/Java Property ID ABX_* = ActiveX	Data Type	Description
"CENTROID_PREFERENCE"	AB_CENTROID_PREFERENCE	32-bit integer	Centroid preference. See the table "Pre-defined property values" on page 354.
"CLOSESTPOINT"	AB_CLOSEST_POINT	Boolean	Specifies whether matching should be done to the closest feature or point address. TRUE = Matches to the closest point address within the search radius. FALSE = **Default**. Matches to the closest feature including street segments and intersections in addition to address points. **NOTE:** This feature requires that at least one points data set and one streets data set are loaded; otherwise, the match will be made to the closest feature.
"COORDINATE_TYPE"	AB_COORDINATE_TYPE	32-bit integer	Determines format of coordinate data. **Default** = AB_COORD_INTEGER See the table "Pre-defined property values" on page 354.
"CORRECT_LAST_LINE"	AB_CORRECT_LAST_LINE	Boolean	True corrects elements of the output last line, providing a good ZIP Code or close match on the soundex even if the address would not match or was non-existent.
"DATUM"	AB_DATUM	32-bit integer	GeoStan datum. See the table "Pre-defined property values" on page 354.
"DPV_DATA_PATH"	AB_DPV_DATA_PATH	String	The file and path of the DPV data.
"DPV_MAILER_ADDRESS"	AB_DPV_MAILER_ADDRESS	String	The address of your company. Used for the DPV false-positive report.
"DPV_MAILER_CITY"	AB_DPV_MAILER_CITY	String	The city where your company resides. Used for the DPV false-positive report.
"DPV_MAILER_COMPANY"	AB_DPV_MAILER_COMPANY	String	The name of your company. Used for the DPV false-positive report.
"DPV_MAILER_STATE"	AB_DPV_MAILER_STATE	String	The state where your company resides. Used for the false-positive report.
"DPV_MAILER_ZIP9"	AB_DPV_MAILER_ZIP9	String	The ZIP + 4 where your company is located. Used for the DPV false-positive report.
"DPV_REPORT_FILE"	AB_DPV_REPORT_FILE	String	The location and file name of the DPV false-positive report.
"DPV_SECURITY_KEY"	AB_DPV_SECURITY_KEY	String	The security key used to access the DPV functionality.

String Property Name	Property ID AB_* = C/C++/.NET/Java Property ID ABX_* = ActiveX	Data Type	Description
"ELEVATION_DATA"	AB_ELEVATION_DATA	Boolean	TRUE = Load and use Centrus Points parcel elevation data FALSE = Don't load Centrus Points parcel elevation data
"FALLBACK_GEOGRAPHIC"	AB_FALLBACK_GEOGRAPHIC	Boolean	True allows for the cascading geocoding of CityCountyState. False turns it off.
"GDL_BUFFER_WIDTH"	AB_GDL_BUFFER_WIDTH	32-bit integer	The distance used to buffer a street segment (feet). **Default** = 100; range = 1 - MAX (4,294,967,295).
"GDL_MAXIMUM_LINES"	AB_GDL_MAXIMUM_LINES	32-bit integer	Maximum number of lines to match in GDL nearest line search. **Default** = 4; range = 1 - MAX.
"GDL_MAXIMUM_POINTS"	AB_GDL_MAXIMUM_POINTS	32-bit integer	Maximum number of points to match GDL nearest point search. **Default** = 4; range = 1 - MAX.
"GDL_MAXIMUM_POLYGONS"	AB_GDL_MAXIMUM_POLYGONS	32-bit integer	Maximum number of polygons to match in GDL nearest polygon search. **Default** = 4; range = 1 - MAX.
"GDL_SEARCH_DISTANCE_TABLE"	AB_GDL_SEARCH_DISTANCE_TABLE	String	Table of GDL logical names versus GDL search distances (feet). Overrides GDL_SEARCH_DISTANCE.
"GDL_SEARCH_DISTANCE"	AB_GDL_SEARCH_DISTANCE	32-bit integer	GDL search distance (feet). **Default** = 5280; range = 1 - MAX.
"INPUT_FIELD_LIST" See page 375.	AB_INPUT_FIELD_LIST	String	Delimited list to be used as input field names.
"INPUT_MODE"	AB_INPUT_MODE	32-bit integer	Two-line, two-line parsed lastline, parsed, multiline input mode, or reverse APN. See the table "Pre-defined property values" on page 354.
"KEEP_COUNTS"	AB_KEEP_COUNTS	Boolean	TRUE = Count match and location codes. FALSE = Do not count (**default**).
"KEEP_MULTIMATCH"	AB_KEEP_MULTIMATCH	Boolean	TRUE = Output all matches (**default**). FALSE = Output single record only.
"LACS_DATA_PATH"	AB_LACS_DATA_PATH	String	The file and path of the LACS^Link data.
"LACS_MAILER_ADDRESS"	AB_LACS_MAILER_ADDRESS	String	The address of your company. Used for the LACS^Link false-positive report.
"LACS_MAILER_CITY"	AB_LACS_MAILER_CITY	String	The city where your company resides. Used for the LACS^Link false-positive report.
"LACS_MAILER_COMPANY"	AB_LACS_MAILER_COMPANY	String	The name of your company. Used for the LACS^Link false-positive report.

String Property Name	Property ID AB_* = C/C++/.NET/Java Property ID ABX_* = ActiveX	Data Type	Description
“LACS_MAILER_STATE”	ABA_LACS_MAILER_STATE	String	The state where your company resides. Used for the false-positive report.
“LACS_MAILER_ZIP9”	AB_LACS_MAILER_ZIP9	String	The ZIP + 4 where your company is located. Used for the LACS^{Link} false-positive report.
“LACS_REPORT_FILE”	AB_LACS_REPORT_FILE	String	The location and file name of the LACS^{Link} false-positive report.
“LACS_SECURITY_KEY”	AB_LACS_SECURITY_KEY	String	The security key used to access the LACS^{Link} functionality.
“MATCH_CODE_EXTENDED”	AB_MATCH_CODE_EXTENDED	Boolean	Specifies whether to return the Extended Match Code (3rd hex digit). For more information, see “Understanding Extended Match Codes” on page 43. TRUE = Return Extended Match Code FALSE= **Default.** Extended Match Code disabled
“MATCH_MODE”	AB_MATCH_MODE	32-bit integer	Determines match strategy for ProcessRecords. See the table “Pre-defined property values” on page 354.
“MAXIMUM_LOOKUPS”	AB_MAXIMUM_LOOKUPS	32-bit integer	Maximum number of matched LookupRecord fields. **Default** = 10; range = 1 - 100.
“MAXIMUM_POINTS”	AB_MAXIMUM_POINTS	32-bit integer	Maximum number of points to match in a Closest Site search. **Default** = 4; range = 1 - 100,000.
“MAXIMUM_POLYGONS”	AB_MAXIMUM_POLYGONS	32-bit integer	Maximum number of polygons to match in a Point in Polygon search. **Default** = 4; range = 1 - 100,000.
“MIXED_CASE”	AB_MIXED_CASE	Boolean	TRUE = Mixed case. FALSE = Upper case (**default**).
“MUST_MATCH_CITY"	AB_MUST_MATCH_CITY	Boolean	Default = FALSE. Usable match modes: Custom
“MUST_MATCH_MAINADDR"	AB_MUST_MATCH_MAINADDR	Boolean	Default = FALSE. Usable match modes: Custom
“MUST_MATCH_STATE"	AB_MUST_MATCH_STATE	Boolean	Default = FALSE. Usable match modes: Custom
“MUST_MATCH_ZIPCODE"	AB_MUST_MATCH_ZIPCODE	Boolean	Default = FALSE. Usable match modes: Custom
“OFFSET_DISTANCE”	AB_OFFSET_DISTANCE	32-bit integer	Geocode offset in feet. **Default** = 50; range = 0 - 5280.

String Property Name	Property ID AB_* = C/C++/.NET/Java Property ID ABX_* = ActiveX	Data Type	Description
"OUTPUT_FIELD_LIST" See page 384.	AB_OUTPUT_FIELD_LIST	String	Delimited list of output field names to be returned (with logical names, if any). ***Note:*** You can have multiple OUTPUT_FIELD_LIST instances in your file. These instances are additive: AddressBroker uses the values listed in each instance.
"PREFER_ZIP_OVER_CITY"	AB_PREFER_ZIP_OVER_CITY	Boolean	Allows a user to prefer candidates that match to input ZIP over candidates that match to input city. GeoStan creates multiple search areas when input city and ZIP do not correspond and this feature helps establish how the candidates should be scored.
"RANGED_ADDRESS"	AB_RANGED_ADDRESS	Boolean	True sets find property GS_FIND_ADDRESS_RANGE to true in GeoStan. False turns it off.
"RDI_DATAPATH"	AB_RDI_DATAPATH	String	Path to RDI data; string value is RDIDATAPATH
"REVERSE_GEOCODE"	AB_REVERSE_GEOCODE	Boolean	Indicates if AddressBroker reverse geocodes input latitudes and longitudes. TRUE = Reverse geocodes FALSE = **Default.** Does not reverse geocode
"REVGEO_SEARCH_DISTANCE"	AB_REVGEO_SEARCH_DISTANCE	32-bit integer	Maximum distance to search (feet) for a reverse geocode. Default = 150; range = 0 - 5280.
"SEARCH_DISTANCE"	AB_SEARCH_DISTANCE	32-bit integer	Maximum distance to search (feet) and closest site search. **Default** = 366,000 feet (approx. 69 miles or 1 degree).
"SQUEEZE_DIST"	AB_SQUEEZE_DIST	String	Distance, in feet, to offset address-level geocodes from the street endpoints. Default is 50 feet. If the squeeze distance is more than half the segment length, sets the distance to the midpoint of the segment.
"STREET_CENTROID"	AB_STREET_CENTROID	Boolean	Specifies whether to return a street segment geocode as an automatic geocoding fallback. TRUE = Return street segment geocode FALSE = **Default.** Street locator disabled
"SUITE_LINK_DATA_PATH"	AB_SUITE_LINK_DATAPATH	String	SuiteLink data path.

String Property Name	Property ID AB_* = C/C++/.NET/Java Property ID ABX_* = ActiveX	Data Type	Description
"THROW_LEVEL" See page 228.	AB_THROW_LEVEL	String	Determines the error level at which an exception is thrown: FATAL—fatal errors, errors and warnings. ERROR—errors and warnings only. WARN— warnings only. INFO—all informational messages. NONE—none. DEBUG—status messages, development only. When a status condition occurs a Status object is built. If the status object type meets or exceeds the THROW_LEVEL setting, the status object is 'thrown'. **Default** = ERROR. Not supported in the Java API.
"Z4_CHANGE_DATE"	AB_Z4_CHANGE_DATE	String	Indicates a request for any address changes after the date provided. The ZIP* input field must also be set per record in order for this request to be fulfilled. Use "MMYYYY" to specify the date. If the ZIP + 4 of an input record is unchanged for the time period, no corresponding output record is calculated or returned. A US.gsl file is required for this functionality. It must reside in the path assigned to your GEOSTAN_PATHS property.
"ZIP_PBKEYS" See page 17.	AB_ZIP_PBKEYS	Boolean	When set to TRUE, opens the file (zipsmld.gsd) needed to return pbKeys for ZIP centroid locations in Master Location Data. When an address point is not available for an address in Master Location Data, this option returns a ZIP centroid and the pbKey™ unique identifier, which can be used to unlock additional information about an address using GeoEnrichment data. **Default** = FALSE

Read-only properties

String Property Name	Property ID AB_* = C/C++/.NET/Java Property ID ABX_* = ActiveX	Data Type	Description
"LOGICAL_NAMES" See page 380.	AB_LOGICAL_NAMES	String	Tab-delimited list of all valid logical names determined by the values assigned to AddressBroker's path properties. ValidateProperties must be called before retrieving the value stored in this property.
"ALL_INPUT_FIELDS"	AB_ALL_INPUT_FIELDS	String	Tab-delimited list of all valid input field names based on value of INIT_LIST property. ValidateProperties must be called before retrieving the value stored in this property.
"ALL_OUTPUT_FIELDS" See page 384.	AB_ALL_OUTPUT_FIELDS	String	Tab-delimited list of all valid output field names based on value of INIT_LIST property. ValidateProperties must be called before retrieving the value stored in this property.
"VERSION"	AB_VERSION	String	AddressBroker version.
"FILE_DATE"	AB_FILE_DATE	String	Publish date of the GSD data.
"DATA_TYPE"	AB_DATA_TYPE	32-bit integer	GeoStan data type. See the table "Pre-defined property values" on page 354.
"RECORDS_REMAINING"*	AB_RECORDS_REMAINING	32-bit integer	Records remaining before license expiration. See the table "Pre-defined property values" on page 354.
"DAYS_REMAINING"	AB_DAYS_REMAINING	32-bit integer	Days remaining before license expiration. See the table "Pre-defined property values" on page 354.
"GEO_RECORD_TOTAL"	AB_GEO_RECORD_TOTAL	32-bit integer	Total number of records geocoded with current license.
"RECORDS_PROCESSED"*	AB_RECORDS_PROCESSED	32-bit integer	Number of processed records returned.
"RECORDS_MATCHED"*	AB_RECORDS_MATCHED	32-bit integer	Number of matched records returned.
"CARRT_PROCESSED"*	AB_CARRT_PROCESSED	32-bit integer	Number of processed records returned that were assigned Carrier Routes.
"DPBC_PROCESSED"*	AB_DPBC_PROCESSED	32-bit integer	Number of processed records returned that were assigned Delivery Point Bar Codes.
"MISC_COUNTS"* See page 382.	AB_MISC_COUNTS	String	Counts of match and location codes of processed records.
"ZIP_PROCESSED"**	AB_ZIP_PROCESSED	32-bit integer	Number of processed records returned that were assigned 5-digit ZIP.
"ZIP4_PROCESSED"*	AB_ZIP4_PROCESSED	32-bit integer	Number of processed records returned that were assigned ZIP + 4.

String Property Name	Property ID AB_* = C/C++/.NET/Java Property ID ABX_* = ActiveX	Data Type	Description
"ZIP4_SKIPPED"*	AB_ZIP4_SKIPPED	32-bit integer	Number of records skipped when using Z4_CHANGE_DATE.

* KEEP_COUNTS must be set to TRUE and KEEP_MULTIMATCH must be set to FALSE for counts to be meaningful.

Pre-defined property values

String Property Name[a]	C, C++, .NET, Java Value	ActiveX Value	Description[b]
"INPUT_MODE"	0	1	**Default**. For input fields, use AddressLine, AddressLine2, LastLine or pbKey. Defined constants: AB_INPUT_NORMAL or ABX_INPUT_NORMAL (ActiveX).
	1	2	Multiline input. For input fields, use Line1–Line6. Defined constants: AB_INPUT_MULTILINE or ABX_INPUT_MULTILINE (ActiveX) *Note*: GeoStan Canada does not support multiline processing.
	3	4	For input fields, use AddressLine, AddressLine2, City, State, and any ZIP field or pbKey. Defined constants: AB_INPUT_PARSED_LASTLINE or ABX_INPUT_PARSED_LASTLINE (ActiveX).
"MATCH_MODE"	0	1	Exact match required. Generates the fewest number of possibles to search. Defined constants: AB_MODE_EXACT or ABX_MODE_EXACT (ActiveX)
	1	2	Very close match required. Generates a moderate number of possibles to search. Defined constants: AB_MODE_CLOSE or ABX_MODE_CLOSE (ActiveX)
	2	3	**Default**. Close match required. Generates the largest number of possibles to search. Defined constants: AB_MODE_RELAX or ABX_MODE_RELAX (ActiveX)
	4	4	This setting imposes additional match rules. Defined constants: AB_MODE_CASS or ABX_MODE_CASS (ActiveX) **NOTE:** CASS mode is not supported in single-line address matching.
	8	5	For interactive single-line address matching only. Defined constants: AB_MODE_INTERACTIVE or ABX_MODE_INTERACTIVE (ActiveX)
	7	none	Allows applications to specify individual "must match" field matching rules for address number, address line, city, ZIP code, and state. Defined constants: AB_MODE_CUSTOM **NOTE:** Custom match mode, and consequently the "MUST_MATCH_*" parameters, are not supported in single-line address matching.
"DATUM"	0	1	NAD27. Defined constants: AB_DATUM_NAD27 or ABX_DATUM_NAD27 (ActiveX)

String Property Name[a]	C, C++, .NET, Java Value	ActiveX Value	Description[b]
	1	2	NAD83. Defined constants: AB_DATUM_NAD83 or ABX_DATUM_NAD83 (ActiveX)
"CENTROID_PREFERENCE"	0	1	Assign address-level geocodes only. Defined constants: AB_CENTROID_NONE or ABX_CENTROID_NONE (ActiveX)
	1	2	**Default**. Use if no address is available. Defined constants: AB_CENTROID_ADDRESS_UNAVAILABLE or ABX_CENTROID_ADDRESS_UNAVAILABLE (ActiveX)
	2	3	Use ZIP centroid geocoding only—no address matching. Defined constants:AB_CENTROID_NO_ADDRESS or ABX_CENTROID_NO_ADDRESS (ActiveX)
"ADDRESS_PREFERENCE"	0	1	**Default**. Select bottommost street address. Defined constants: AB_ADDRESS_BOTTOM or ABX_ADDRESS_BOTTOM (ActiveX)
	1	2	Prefer PO Box. Defined constants: AB_ADDRESS_POBOX or ABX_ADDRESS_POBOX (ActiveX)
	2	3	Prefer street address. Defined constants: AB_ADDRESS_STREET or ABX_ADDRESS_STREET (ActiveX)
"COORDINATE_TYPE" See page 67.	0	N/A	**Default**. Coordinate is a fixed point value represented as an integer one million times (six decimal places) larger than the actual number. Example: 40123456 = "40.123456" Defined constants: AB_COORD_INTEGER or ABX_COORD_INTEGER (ActiveX)
	1	N/A	A floating point decimal value. Example: 40.123456 = "40.123456" Defined constants: AB_COORD_FLOAT or ABX_COORD_FLOAT (ActiveX)
"RECORDS_REMAINING"	Read-only.	Read-only.	Number of records remaining or 'unlimited'. License-based. Defined constants: AB_LICENSE_UNLIMITED or ABX_LICENSE_UNLIMITED (ActiveX)
"DAYS_REMAINING"	Read-only.	Read-only.	Number of days remaining or 'unlimited'. License-based. Defined constants: AB_LICENSE_UNLIMITED or ABX_LICENSE_UNLIMITED (ActiveX)
"DATA_TYPE"	0 Read-only.	0 Read-only.	USPS Defined constants: AB_DATA_TYPE_USPS or ABX_DATA_TYPE_USPS (ActiveX)
	1 Read-only.	1 Read-only.	TIGER/Centrus Enhanced Defined constants: AB_DATA_TYPE_TIGER or ABX_DATA_TYPE_TIGER (ActiveX)

String Property Name[a]	C, C++, .NET, Java Value	ActiveX Value	Description[b]
	2 Read-only.	2 Read-only.	TOMTOM Defined constants: AB_DATA_TYPE_TOMTOM or ABX_DATA_TYPE_TOMTOM (ActiveX)
	3 Read-only.	3 Read-only.	DEPRECATED SANBORN_POINT Defined constants: AB_DATA_TYPE_SANBORN or ABX_DATA_TYPE_SANBORN (ActiveX)
	4 Read-only.	4 Read-only.	DEPRECATED TELE_ATLAS Defined constants: AB_DATA_TYPE_TELE_ATLAS or ABX_DATA_TYPE_TELE_ATLAS (ActiveX)
	5 Read-only.	5 Read-only.	DEPRECATED GEOSYS Defined constants: AB_DATA_TYPE_GEOSYS or ABX_DATA_TYPE_GEOSYS (ActiveX)
	6 Read-only.	6 Read-only.	HERE Defined constants: AB_DATA_TYPE_NAVTEQ or ABX_DATA_TYPE_NAVTEQ (ActiveX)
	7 Read-only.	7 Read-only.	TOMTOM_POINT Defined constants: AB_DATA_TYPE_TOMTOM_POINT or ABX_DATA_TYPE_TOMTOM_POINT (ActiveX)
	8 Read-only.	8 Read-only.	CENTRUS POINT Defined constants: AB_DATA_TYPE_CENTRUS_POINT or ABX_DATA_TYPE_CENTRUS_POINT (ActiveX)
	9 Read-only.	9 Read-only.	AUXILIARY FILE Defined constants: AB_DATA_TYPE_AUXILIARY_POINT or ABX_DATA_TYPE_AUXILIARY_POINT (ActiveX)
	10 Read-only.	10 Read-only.	USER DICTIONARY Defined constants: AB_DATA_TYPE_USER_DICTIONARY or ABX_DATA_TYPE_USER_DICTIONARY (ActiveX)
	11 Read-only.	11 Read-only.	HERE POINT Defined constants: AB_DATA_TYPE_NAVTEQ_POINT or ABX_DATA_TYPE_NAVTEQ_POINT (ActiveX)
	12 Read-only	12 Read-only	MASTER LOCATION DATA Defined constants: AB_DATA_TYPE_MASTER_LOCATION or ABX_DATA_TYPE_MASTER_LOCATION (ActiveX)
"CACHE_SIZE"	0	1	No cache used. Defined constants: AB_CACHE_SIZE_NONE or ABX_CACHE_SIZE_NONE (ActiveX)

String Property Name[a]	C, C++, .NET, Java Value	ActiveX Value	Description[b]
	1	2	**Default**. Moderate cache—15 objects. Defined constants: `AB_CACHE_SIZE_MEDIUM` or `ABX_CACHE_SIZE_MEDIUM` (ActiveX)
	2	3	Large cache —20 objects. Defined constants: `AB_CACHE_SIZE_LARGE` or `ABX_CACHE_SIZE_LARGE` (ActiveX)

a ActiveX string property names do not include the underscore.

b “Defined constants” are programmatic aids that may be substituted for the associated value in C, C++, .NET, Java, or ActiveX.

14 – Properties descriptions

In this chapter

This chapter provides detailed descriptions of selected AddressBroker properties. The included properties are either common to most AddressBroker applications, or require more discussion than fit into Chapter 13, "Properties". Each property discussion includes a summary statement, a syntax statement, and a description of the property's parameters and how the property is used. Most include a descriptive code fragment. Properties included in this section are listed alphabetically. For a complete list of AddressBroker properties, see Chapter 13, "Properties".

Quick reference

ALL_INPUT_FIELDS (read-only) Property

Delimited list of all valid *input* field names.

ALL_OUTPUT_FIELDS (read-only) Property

Delimited list of all valid *output* field names.

BUFFER_RADIUS Property

Spatial buffer (radius or width) to apply to features in an object file.

BUFFER_RADIUS_TABLE Property

Delimited list of Spatial+ buffer radius entries.

CLOSEST_SITE_FILTER Property

Limits the number of records by a user-specified filter.

DEMOGRAPHICS_PATHS Property

Delimited list of logical names, paths, and file names of Demographics data.

DPV_DATA_PATH Property

Specifies the file name and path for the Delivery Point Validation (DPV®) data.

DPV_SECURITY_KEY Property

Specifies the security key to use the DPV functionality.

GDL_SPATIAL_PATHS Property

Deprecated. Delimited list of logical names and directory paths to GDL data.

GEOSTAN_CANADA_PATHS Property

Delimited list of logical names and directory paths to GeoStan Canada data.

GEOSTAN_PATHS Property

Delimited list of logical names and directory paths to GeoStan data.

GEOSTAN_Z9_PATHS Property

Delimited list of logical names and file names to GeoStan ZIP Code data.

GS_MEMORY_LIMIT Property

Specifies the maximum amount of memory to use for memory-mapping data files. (for 64-bit applications only).

INIT_LIST Property

Delimited list of logical names.

INPUT_FIELD_LIST Property

Delimited list of field names.

INPUT_MODE Property

Defines how the application address input is organized and formatted.

IP_FILTER Property

Allows or denies access to individual or groups of IP addresses to the AddressBroker server. Set by the server administrator, rather than by the client programmer.

LACS_DATA_PATH Property

Specifies the file name and path for the LACS^Link®^ data.

LACS_SECURITY_KEY Property

Specifies the security key to use the LACS^Link^ functionality.

LOG_ROLLOVER (server-only) Property

Sets age and size criteria for the status and request log files for the periodic rollover of file names so that the log file does not become too large or too old.

LOGICAL_NAMES (read-only) Property

Tab-delimited list of logical names.

MAX_OPEN_GSBS Property

Specifies the maximum number of open GSB files (for Linux systems only).

MISC_COUNTS (read-only) Property

Tab-delimited list of miscellaneous counts.

OUTPUT_FIELD_LIST Property

Delimited list of fields names to be retrieved as output.

RDI_DATAPATH Property

Specifies the file name and path for the Residential Delivery Indicator (RDI™) data.

REQUEST_LOG Property

The log file that contains summaries of requests (client interactions with the server). Set by the server administrator, rather than by the client programmer.

REQUEST_LOG_OPTIONS Property

Specifies the format of the request log and the delimiter that separates fields.

SPATIAL_PATHS Property

Delimited list of logical names, paths, and file names of Spatial+ data.

STATUS_LOG Property

The log file that contains general server events. Set by the server administrator, rather than by the client programmer.

ALL_INPUT_FIELDS (read-only) Property

Delimited list of all valid *input* field names.

Syntax

```
ab.GetProperty ( "ALL_INPUT_FIELDS", buffer, buffersize )
where buffer returns Value, and Value = FieldName \t FieldName \t …
```

Type

String list of field names.

Notes

The `ALL_INPUT_FIELDS` read-only property holds a tab- (\t) delimited list of all valid input field names you can use in your program. The list returned in `ALL_INPUT_FIELDS` is dependent on the values set in several other properties including `INIT_LIST`, `GEOSTAN_PATHS`, `GEOSTAN_Z9_PATHS`, `GEOSTAN_Z5_PATHS`, `GEOSTAN_CANADA_PATHS`, `SPATIAL_PATHS`, `DEMOGRAPHICS_PATHS`, and `INPUT_MODE`.

C++ Example

```
ab.SetProperty ( "GEOSTAN_PATHS",
"[GEOSTAN]C:\Program Files\Precisely\cd2tiger |
[GDT] C:\Program Files\Precisely\cd2gdt" );
ab.SetProperty ( "GEOSTAN_Z9_PATHS",
"[GEOSTAN_Z9]C:\Program Files\Precisely\cd2tiger\US.z9");
ab.SetProperty ( "INIT_LIST", "GEOSTAN | GEOSTAN_Z9" );
ab.ValidateProperties ( );
   ...
ab.GetProperty ( "ALL_INPUT_FIELDS", buffer, buffersize );
printf ( "%s", buffer );
//printf output =
//RecordID\tFirmName\tAddressLine\tAddressLine2\tLastLine\t....
```

ALL_OUTPUT_FIELDS (read-only) Property

Delimited list of all valid *output* field names.

Syntax

```
ab.GetProperty ( "ALL_OUTPUT_FIELDS", buffer, buffersize )
where buffer returns Value, and Value = FieldName \t FieldName \t …
```

Type

String list of field names.

Notes

The ALL_OUTPUT_FIELDS read-only property holds a tab- (\t) delimited list of all valid output field names. The list returned in ALL_OUTPUT_FIELDS is dependent on the values set in several other properties including INIT_LIST, GEOSTAN_PATHS, GEOSTAN_Z9_PATHS, GEOSTAN_Z5_PATHS, GEOSTAN_CANADA_PATHS, SPATIAL_PATHS, DEMOGRAPHICS_PATHS, and INPUT_MODE.

C++ Example

```
ab.SetProperty ( "GEOSTAN_PATHS",
"[GEOSTAN]C:\Program Files\Precisely\cd2tiger |
[GDT] C:\Program Files\Precisely\cd2gdt" );
ab.SetProperty ( "GEOSTAN_Z9_PATHS",
"[GEOSTAN_Z9]C:\Program Files\Precisely\cd2tiger\US.z9");
ab.SetProperty ( "INIT_LIST", "GEOSTAN | GEOSTAN_Z9" );
ab.ValidateProperties ( );
   ...
ab.GetProperty ( "ALL_OUTPUT_FIELDS", buffer, buffersize );
printf ( "%s", buffer );
//printf output =
//RecordID\tFirmName\tAddressLine\tAddressLine2\tLastLine\t....
```

BUFFER_RADIUS Property

Spatial buffer (radius or width) to apply to features in an object file.

Syntax

```
ab.SetProperty ( "BUFFER_RADIUS" (unsigned long), "Value" );
where Value = number of feet.
```

Type

Long. Default = 0; range = 0 - 5280000.

Notes

There are two properties that specify the buffer radius for spatial analysis: BUFFER_RADIUS and BUFFER_RADIUS_TABLE. AddressBroker uses the value assigned to BUFFER_RADIUS for the general case.

The **ValidateProperties** function can only validate the syntax of your entries.

C++ Example

```
ab.SetProperty ( "BUFFER_RADIUS", (unsigned long) 50 );
```

See Also

"LOGICAL_NAMES (read-only) Property" on page 380.

Chapter 17, "Location Codes".

"Spatial+ output fields" on page 418.

BUFFER_RADIUS_TABLE Property

Delimited list of Spatial+ buffer radius entries.

Syntax

```
ab.SetProperty ( "BUFFER_RADIUS_TABLE", "Value" );
where Value = location code:buffer radius[LOGICAL NAME] | location
code:buffer radius[LOGICAL NAME]...
or where Value = location code:buffer radius[LOGICAL NAME] \t location
code:buffer radius[LOGICAL NAME]...
```

Type

String list of buffer radius entries.

Notes

The BUFFER_RADIUS_TABLE property holds a delimited list. Each element in the list consists of three elements. The first element is a location quality code (specified fully or with a wild card character). The first element is followed by a colon. The second element is a radius buffer (in feet). The last element, in brackets, is the logical name of a Spatial+ data file. The logical name must be specified in the SPATIAL_PATHS property.

There are two properties that specify the buffer radius for spatial analysis: BUFFER_RADIUS and BUFFER_RADIUS_TABLE. AddressBroker uses the value assigned to BUFFER_RADIUS for the general case.

BUFFER_RADIUS_TABLE lets you specify the radius to use based on the **LocationQualityCode** output field value of an individual record.

You can use BUFFER_RADIUS_TABLE without listing **LocationQualityCode** in the OUTPUT_FIELD_LIST property.

For example, a table entry of:

```
AS0:50[FLOODPLAIN]
```

specifies that when AddressBroker does a spatial analysis on addresses with the location code “ASO”, a buffer radius of 50 feet be used with the FLOODPLAIN data.

To minimize the number of BUFFER_RADIUS_TABLE entries, you can use the star (*) character as a wild card to replace the trailing end of a location code. For example, a table entry of:

```
A*:1000[COUNTIES]
```

indicates that when AddressBroker does a spatial analysis on addresses with a location code starting with “A” followed by any other value, a buffer radius of 1000 feet be used with the COUNTIES data.

The match algorithm for BUFFER_RADIUS_TABLE is a linear left-to-right search. That is, the first entry in the buffer radius table to match the location code is the one used. This is particularly important to note when using wild-cards.

The most specific table entries should be first (left-most) in the table. The most general entries should be toward the end (right-most) of the table. For example:

```
AS0:10[COUNTIES] | A*:1000[COUNTIES]
```

specifies that when AddressBroker does a spatial analysis on addresses with a “best” location quality code (“AS0”) a buffer radius of 10 feet be used with the COUNTIES data. However, the spatial analysis of addresses with more general location quality codes (A*) is done with a radius buffer of 1000 feet.

If these two BUFFER_RADIUS_TABLE entries were reversed, the “AS0:10[COUNTIES]” would never be applied, as “A*:1000[COUNTIES]” is the more general match. When making BUFFER_RADIUS_TABLE entries, it is important to carefully specify location codes and to carefully order the entries for your particular needs.

If no BUFFER_RADIUS_TABLE entry matches the location code assigned to an address, the value assigned to BUFFER_RADIUS is used.

The **ValidateProperties** function can only validate the syntax of your entries.

C++ Example

```
ab.SetProperty ( "BUFFER_RADIUS”, (unsigned long) 50 );
```

```
ab.SetProperty ( "BUFFER_RADIUS TABLE",
"AS0:100[COUNTIES] | AS1:200[COUNTIES] | A*:1000[COUNTIES]");
```

See Also

“LOGICAL_NAMES (read-only) Property” on page 380.

“Spatial+ output fields” on page 418.

Chapter 17, "Location Codes".

CLOSEST_SITE_FILTER Property

Limits the number of ClosestSite records by a user-specified filter.

Syntax

```
ab.SetProperty ( "CLOSEST_SITE_FILTER", “Criterion" );
where Criterion consists of a fieldname, operator, and value
combination.
```

Type

```
String.
```

Notes

The `CLOSEST_SITE_FILTER` property reduces the number of returned ClosestSite records by using a filter criteria. The criteria is compared to values in the attribute file (GSA) to determine if the point is kept. You must specify a logical name with the filter criteria. Field names must be fully qualified and include the logical name and the field name (for example, [MUNI] Population).

The filter criteria uses the following format: Fieldname operator value:

- The fieldname must be a valid field from the attribute file.
- String values support the following operators: **=**, **<>**, and **IN**.
- The **=** operator for string values supports the * wild card operator as long as it appears as the last character.
- Numeric values support the following operators: **=**, **<>**, **IN, >, >=, <, and <=.**
- The **IN** operator works as an **OR** condition. All of the values are compared until an exact match is found. Numeric values can be decimal numbers. Commas are supported with all numeric operators except the **IN** operator.
- Values for string searches must have an apostrophe on each end (for example, ‘string value’). Values for numeric searches do not have this restriction.

Examples of valid criteria include the following:

- “Lastname = ‘Smith’”
- “Population >= 20000”
- “Lastname IN (‘Jones’, ‘Smith’, ‘Johnson’, ‘Williams’)”
- “Lastname = ‘Sm*’”
- “City < > ‘Los Angeles’”

You can specify multiple filters for multiple logical names. If you use the same logical name more than once and use a different filter, only the first occurrence of that logical name is honored. For example, "[MUNI] Population > 80000 | [MUNI] Placename=’Boulder’” applies the Population filter and returns the records that meet the criteria.

If the criteria is not valid, no values are returned.

C++ Example

```
ab.SetProperty ( "CLOSEST_SITE_FILTER", "[MUNI] Population < 10000 |
[COUNTIES] Population > 5000");
```

See Also

"Spatial+ output fields" on page 418.

DEMOGRAPHICS_PATHS Property

Delimited list of logical names, paths, and file names of Demographics data.

Syntax

```
DEMOGRAPHICS_PATHS = [Logical_Name]path/<file>.dld
```

Type

String pairings of logical names with path and file names.

Notes

The DEMOGRAPHICS_PATHS property holds a delimited list of pairs. The first element of the pair is a logical name and the second is the full path and file name of a Demographics file.

This property is only required if Demographics Library is included in your Precisely license, and you are using Demographics data in your application.

Initialization File Example

The DEMOGRAPHICS_PATHS property is defined in the server.ini file only.

```
DEMOGRAPHICS_PATHS =
[Census2k]"C:\Program Files\Precisely\CENSUS2K.dld"
```

DPV_DATA_PATH Property

Specifies the file name and path for the DPV data.

Syntax

```
DPV_DATA_PATH = path/<file>
```

Type

```
String of path and file names.
```

Notes

This property specifies the location of the DPV data.

This property is only required if you are using the DPV functionality in your application.

C++ Examples

```
DPV_DATA_PATH = s:data\April05
```

DPV_SECURITY_KEY Property

Specifies the security key to use the DPV functionality.

Syntax

```
DPV_SECUIRTY_KEY = <security_key>
where security key is in the format xxxx-xxxx-xxxx-xxxx.
```

Type

String.

Notes

This property specifies the security key required to access the DPV functionality. You can obtain a security key from support.precisely.com.

This property is only required if you are using DPV functionality in your application.

C++ Examples

```
DPV_SECURITY_KEY = A123-4567-BC8D-9123
```

GDL_SPATIAL_PATHS Property

Delimited list of logical names and directory paths to GDL data.

Syntax

```
GDL_SPATIAL_PATHS = [Logical_Name]path/<file>.gsb
```

Type

String pairings of logical names with path and file names.

Notes

DEPRECATED

The `GDL_SPATIAL_PATHS` property holds a delimited list of pairs. The first element in the pair is a logical name and the second is one or more full path directory names.

GDL can use the `SPATIAL_PATHS` logical names to eliminate duplicate definitions for both Spatial and GDL.

Initialization File Example

```
The GDL_SPATIAL_PATHS property is defined in the server.ini file only.

GDL_SPATIAL_PATHS = \
[STATES2]
"C:\Program Files\Precisely\AddressBroker\Data
\States.gsb" );
```

GEOSTAN_CANADA_PATHS Property

Delimited list of logical names and directory paths to GeoStan Canada data.

Syntax

```
GEOSTAN_CANADA_PATHS = [Logical_Name]path/
```

Type

String pairings of logical names with directory paths.

Notes

The `GEOSTAN_CANADA_PATHS` property holds a delimited list of pairs. The first element in the pair is a logical name and the second is one or more full path directory names.

This property is only required if GeoStan Canada is included in your Precisely license, and you are using GeoStan Canada data in your application.

Initialization File Example

```
The GEOSTAN_CANADA_PATHS property is defined in the server.ini file
only.

GEOSTAN_CANADA_PATHS = \[GEOSTAN_C]
"C:\Program Files\Precisely\AddressBroker\Data" );
```

GEOSTAN_PATHS Property

Delimited list of logical names and directory paths to GeoStan data.

Syntax

```
GEOSTAN_PATHS = [Logical_Name]path/
```

Type

String pairings of logical names with directory paths.

Notes

The GEOSTAN_PATHS property holds a delimited list of pairs. The first element in the pair is a logical name and the second is a full path directory name.

The GEOSTAN_PATHS property allows multiple GeoStan data paths to be concatenated together for a single logical name, separated by semicolons.

The GEOSTAN_PATHS property is a list of directory names, not file names.

You *must* set the GEOSTAN_PATHS property in the server .ini file. You can also set the GEOSTAN_PATHS property and logical name on the client, but it must be identical to what you specify in the server .ini file.

Note: Comments are not allowed on the GEOSTAN_PATHS line in the AddressBroker server .ini file.

Initialization File Example

```
GEOSTAN_PATHS = \
[GEOSTAN]"C:\Program Files\Precisely\cd2tiger" | \
[GDT]"C:\Program Files\Precisely\cd2gdt"
```

GEOSTAN_Z9_PATHS Property

Delimited list of logical names and file names to GeoStan ZIP Code data.

Syntax

```
GEOSTAN_Z9_PATHS = [Logical_Name]path/us.z9
```

Type

String pairings of logical names with path and file names.

Notes

The GEOSTAN_Z9_PATHS property holds a delimited list of pairs. The first element of the pair is a logical name and the second is the full path and file name of a GeoStan ZIP Code data file.

You *must* set the GEOSTAN_Z9_PATHS property in the server .ini file. You can also set the GEOSTAN_Z9_PATHS property and logical name on the client, but it must be identical to what you specify in the server .ini file.

Initialization File Example

```
GEOSTAN_Z9_PATHS =
[GEOSTAN_Z9] “C:\Program Files\Precisely\cd2tiger\US.z9” | \
[GDT_Z9] “C:\Program Files\Precisely\cd2gdt\US.z9”
```

GS_MEMORY_LIMIT Property

The maximum amount of memory, in megabytes, to allocate for memory-mapping data files. This property only applies to 64-bit applications. For 32-bit applications, data files are not memory-mapped and attempts to set this property will be ignored.

When AddressBroker is initialized, it will memory-map as many data files into memory as `GS_MEMORY_LIMIT` allows. The default value of 16 GB is sufficient for memory-mapping two streets and two points datasets. Memory-mapping your data files can provide a 10-15% performance improvement compared to mapping none of them. However, if your environment has memory constraints, you can set the `GS_MEMORY_LIMIT` to the number of megabytes you can afford.

This initialization property can only be set once per executable process. If you attempt to set this property again (i.e., on a separate thread), the request is ignored.

Syntax

```
GS_MEMORY_LIMIT = Value
```

Where `Value` can be a number in the range 0-256000 (megabytes).

Type

Long. Default=16000 (megabytes).

INIT_LIST Property

Delimited list of logical names.

Syntax

```
ab.SetProperty ( "INIT_LIST", "Value" )
where Value = Logical Name | Logical Name | …
or where Value = Logical Name \t Logical Name \t …
```

Type

String list of logical names.

Notes

The INIT_LIST property holds a delimited list of logical names referencing the geo-demographic data files your application uses. Logical names are defined in the GEOSTAN_PATHS, GEOSTAN_Z9_PATHS, GEOSTAN_Z5_PATHS, GEOSTAN_CANADA_PATHS, SPATIAL_PATHS, and DEMOGRAPHICS_PATHS properties.

Precisely recommends setting the path properties to contain all available reference file information. However, when setting INIT_LIST, assign only the logical names of the geo-demographic data your application accesses. The client cannot create logical names that do not exist in the server .ini file; therefore, ensure that the logical names you specify using INIT_LIST are declared in one of the *_PATHS properties in the server .ini file. Additionally, ensure that the GeoStan and GeoStan ZIP9 data you assign to INIT_LIST are compatible.

Initialization File Example

```
; Here we assign all possible reference files to the path properties.
GEOSTAN_PATHS       =
[GEOSTAN]"C:\Program Files\Precisely\cd2tiger" | \
[GDT]"C:\Program Files\Precisely\cd2gdt"
GEOSTAN_Z9_PATHS    =
[GEOSTAN_Z9]"C:\Program Files\Precisely\cd2tiger\US.z9" \|
[GDT]"C:\Program Files\Precisely\cd2gdt\US.z9"
SPATIAL_PATHS       =
[COUNTIES]"C:\Program Files\Precisely\COUNTIES.gsb"
DEMOGRAPHICS_PATHS =
[Census2k]"C:\Program Files\Precisely\CENSUS2K.dld"
; Here we assign only the logical names for data the
; application will access.
; Note that GeoStan and GeoStan ZIP9 data are compatible.
INIT_LIST           = GEOSTAN | GEOSTAN_Z9 | COUNTIES
```

INPUT_FIELD_LIST Property

Delimited list of field names.

Syntax

```
ab.SetProperty ( "INPUT_FIELD_LIST", "Value" )
where Value = FieldName | FieldName | …
or where Value = FieldName \t FieldName \t …
```

Type

String list of field names.

Notes

The INPUT_FIELD_LIST property holds a delimited list of field names to be used by the application as input. To find out which input field names you can assign to INPUT_FIELD_LIST, use the **GetProperty** function call with ALL_INPUT_FIELDS as an argument.

By specifying only those fields the application uses (as opposed to all of the fields in your data), AddressBroker manages memory more efficiently, and optimally transfers data across the network in client/server applications.

C++ Example

```
ab.SetProperty ( "INPUT_FIELD_LIST",
               "FirmName \t AddressLine | LastLine" );
```

INPUT_MODE Property

Delimited list of input modes.

Syntax

```
ab.SetProperty ( "INPUT_MODE" (unsigned long), "Value" );
where Value = AB_INPUT_NORMAL, AB_INPUT_MULTILINE, or
AB_INPUT_PARSED_LASTLINE.
```

Type

Long. Default = NORMAL.

Notes

AddressBroker sets this property to control how the input data is provided from the client application to the server. It communicates the format of the address information to the sever. For example:

`AB_INPUT_NORMAL` (or "`INPUT NORMAL`") - The application supplies `AddressLine`, `AddressLine2`, and `LastLine`.

`AB_INPUT_MULTILINE` (or "`INPUT MULTILINE`") - The application supplies `Line1` through `Line6`.

`AB_INPUT_PARSED_LASTLINE` (or "`INPUT PARSED LASTLINE`") - The application supplies `AddressLine`, `AddressLine2`, `City`, `State`, and any `ZIP` field.

C++ Example

```
ab.SetProperty ( "INPUT_MODE", (unsigned long) ???????? );
```

See Also

"LOGICAL_NAMES (read-only) Property" on page 380.

Chapter 17, "Location Codes".

IP_FILTER Property

Allows or denies access to individual or groups of IP addresses to the AddressBroker server. Set by the server administrator, rather than by the client programmer.

Syntax

```
IP_FILTER = [Allow] <ip values> | [Deny] <ip values>
```

Type

Two lists of IP values.

Notes

Items in the Allow list are allowed access, while items in the Deny list are not allowed access. Wild card characters (*) are allowed, and you can use one wildcard character to include all. You can specify multiple IP addresses in each list, but they must be comma separated (for example: `[Allow} 172.17.*, 127.0.0.1, 65.209.*`).

Example

```
IP_FILTER = [Allow] 172.17.* | [Deny] 65*
```

This example allows clients with IP addresses of the form 172.17.* and denies access to IP addresses of the form 65*.

In the case where an IP is in both the Allow and Deny lists, the most specific address takes precedence. If the precedence is the same, then the action is to deny the address, as shown in the following example:

```
IP_FILTER = [Allow] 172.17.* | [Deny] 172.17.1.114
```

The above example causes machine 114 to be denied access, since it is a more specific address than the Allow list items. All other machines in the range of 17.17.* would be allowed access.

If you specify either Allow or Deny, then it is inferred that you want to allow only those machines and deny all others, event though there was no Deny list explicitly specified. The opposite is true as well. If a user is denied access to the server, a message is written to the log file. The following example allows only those machines with 172.17.*:

```
IP_FILTER = [Allow] 172.17.*
```

LACS_DATA_PATH Property

Specifies the file name and path for the LACS^{Link} data.

Syntax

```
LACS_DATA_PATH = path/<file>
```

Type

String of path and file names.

Notes

This property specifies the location of the LACS^{Link} data.

This property is only required if you are using the LACS^{Link} functionality in your application.

C++ Examples

```
LACS_DATA_PATH = s:data\April05
```

LACS_SECURITY_KEY Property

Specifies the security key to use the LACS^{Link} functionality.

Syntax

```
LACS_SECUIRTY_KEY = <security_key>
where security key is in the format xxxx-xxxx-xxxx-xxxx.
```

Type

String.

Notes

This property specifies the security key required to access the LACS^{Link} functionality. You can obtain a security key from support.precisely.com.

This property is only required if you are using LACS[Link] functionality in your application.

C++ Examples

```
LACS_SECURITY_PATH = A123-4567-BC8D-9123
```

LOG_ROLLOVER (server-only) Property

Sets criteria for the request_log in the server INI file. Set by the server administrator, rather than by the client programmer.

Syntax

```
LOG_ROLLOVER = | [STATUS]<xxxMB|yyyD> | REQUEST]<xxxMB|yyyD>
```

Type

Long. Default = NORMAL.

Notes

The LOG_ROLLOVER property sets age and size criteria for the status and request log files for the periodic rollover of file names. This property ensures that the log file does not become too large or too old to be useful.

The log files that are rolled over include the STATUS_LOG and REQUEST_LOG. The Status log file contains general server events, such as starting or stopping the server. The Request log file summarizes requests (client interactions with the server). Both log files are set by the server administrator.

Example

```
LOG ROLLOVER = [STATUS]10MB | [STATUS]10D |[REQUEST]10MB |
[REQUEST]10D
```

This example performs a rollover (by closing the log files and opening new log files with a sequential number at the end) when the log files reach 10 MBytes in size or become 10 days old.

See Also

"REQUEST_LOG Property" on page 386.

"REQUEST_LOG_OPTIONS Property" on page 387.

LOGICAL_NAMES (read-only) Property

Tab-delimited list of logical names.

Syntax

```
ab.GetProperty ("LOGICAL NAME", buffer, buffersize)
where buffer returns Value and
Value = Logical Name:Type \t Logical Name:Type \t …
and Type = G, D, S, Z, Y, C, or L.
```

Type

String list of logical names.

Notes

The LOGICAL_NAMES read-only property holds a tab- (\t) delimited list of all logical names defined in the GEOSTAN_PATHS, GEOSTAN_Z9_PATHS, GEOSTAN_Z5_PATHS, GEOSTAN_CANADA_PATHS, SPATIAL_PATHS, and DEMOGRAPHICS_PATHS properties. Each list item is composed of two elements separated by a colon. The first element is the logical name. The last element is an alphabetic code indicating the type of data file associated with the logical name:

- **G**—GeoStan
- **D**—Demographics
- **S**— Spatial+
- **Z**—GeoStan ZIP9
- **Y**—GDL ZIP5
- **C**—GeoStan Canada

The LOGICAL_NAMES property is particularly useful when the logical names are unknown in advance, i.e. client applications. This property lets you query the server for a list of logical names at runtime.

C++ Example

```
ab.SetProperty ( "GEOSTAN_PATHS",
"[GEOSTAN]C:\Program Files\Precisely\cd2tiger |
[GDT] C:\Program Files\Precisely\cd2gdt" );
ab.SetProperty ( "SPATIAL_PATHS",
"[COUNTIES]C:\Program Files\Precisely\COUNTIES.gsb" );
ab.SetProperty ( "GEOSTAN_Z9_PATHS",
   "[GEOSTAN_Z9]C:\Program Files\Precisely\cd2tiger\US.z9 | [GDT_Z9]
C:\Program Files\Precisely\cd2gdt\US.z9" );
ab.SetProperty ( "DEMOGRAPHICS_PATHS",
[Census2k]"C:\Program Files\Precisely\CENSUS2K.dld"
:
ab.GetProperty ( "LOGICAL_NAMES", buffer, buffersize );
printf ( "%s", buffer );
```

```
//printf output =
//GEOSTAN:G\tGDT:G\tGEOSTAN_Z9:Z\tGDT_Z9:Z\tCOUNTIES:S
```

MAX_OPEN_GSBS Property

Specifies the maximum number of open gsb files. Set by the server administrator, rather than by the client programmer.

Syntax

```
MAX_OPEN_GSBS = Value
```

Where `Value` can be a number in the range 1-4096.

Type

Long. Default=0.

Notes

This is the maximum number of GSBs that the AddressBroker server can open simultaneously. If the number of open GSBs reaches this limit, AddressBroker will close some of the GSBs that are open, but not currently in use servicing a client request, until the number of open GSBs falls below this number.

This property is only applicable to Linux systems.

C++ Example

The following code sample would be included in the server .ini file.

```
MAX_OPEN_GSBS = 2048
```

MISC_COUNTS (read-only) Property

Tab-delimited list of miscellaneous counts.

Syntax

```
ab.GetProperty ( "MISC_COUNTS", buffer, buffersize )
where buffer returns Value and
Value = Counter Label:Count \t Counter Label:Count \t …
and where Count = Integer value
```

Type

String list of counters.

Notes

The `MISC_COUNTS` read-only property contains a tab- (\t) delimited list of miscellaneous counters and their values. Each item in the list consists of three elements. The first element is the counter label. It is followed by a colon (:). The last element is a numeric count. The `MISC_COUNTS` property is a list of counts for all counter labels. The table below provides a complete listing of counter labels.

`KEEP_COUNTS` must be set to **TRUE** and `KEEP_MULTIMATCH` must be set to **FALSE** for counts to be meaningful. Counts are not meaningful in a multiple match situation.

Counts are kept when **`ProcessRecords`** is called.

Successful match codes	Location codes	Error match codes
standardized and matched records	address-level geocodes	address not found
intersection matched records	ZIP + 4 centroid level geocodes	low-level error
non-USPS matched records	block group accuracy geocodes	GSD file not found error
address lines corrected	census tract accuracy geocodes	incorrect GSD file signature or version ID error
street types corrected	county-level accuracy geocodes	GSD file out of date error
pre-directionals corrected	geocodes based on 5-digit ZIP centroid	city + state or ZIP not found error
post-directionals corrected	geocodes based on ZIP+2 centroid	input ZIP not found in directory error
street names corrected	geocodes based on ZIP + 4 centroid	input city not found in directory error

Successful match codes	Location codes	Error match codes
last lines corrected		input city not unique in directory error
number of ZIP Codes corrected		out of license area error
cities corrected		license expired error
states corrected		matching street not found in directory error
Number of ZIP + 4 Codes corrected		matching cross street not found for intersection match error
		matching ranges not found error
		unresolved match error
		too many possible cross streets for intersection match error
		address not found in multiline match error

C++ Example

```
ab.GetProperty ( "MISC_COUNTS", buffer, buffersize );
printf ( "%s", buffer );
//printf output =
//standardized and matched records:10\tintersection matched
records:2\t…
```

Counts are returned in top-down left-to-right order, as listed in the table above.

OUTPUT_FIELD_LIST Property

Delimited list of fields names to be retrieved as output.

Syntax

```
ab.SetProperty ( "OUTPUT_FIELD_LIST", "Value" )
where, for fields that reference to GeoStan data,
Value = FieldName | FieldName | …
or Value = FieldName \t FieldName \t …for fields
and where, for fields that reference to Spatial+, GDL, or Demographics
Library data,
Value = FieldName [Logical Name] | FieldName [Logical Name] | …
or Value = FieldName [Logical Name] \t FieldName [Logical Name] \t....
```

Type

String list of field names.

Notes

The `OUTPUT_FIELD_LIST` property holds a delimited list of field names to be retrieved by the application. To find out which output field names you can assign to `OUTPUT_FIELD_LIST`, use the **`GetProperty`** function call with `ALL_OUTPUT_FIELDS` property as an argument.

When assigning the list of output fields, you must append a logical name, in square brackets ([ ]), to each field name that requires reference to Spatial+ or Demographics Library data. The logical name establishes the geo-demographic data your application uses to generate these output field values. See "Decimals in input/output field values" on page 67 for more information on this topic.

A field name-logical name pair may not exceed 32 bytes.

By specifying the subset of output fields you want retrieved (as opposed to all of the possible output fields AddressBroker could generate, given your input), AddressBroker manages memory more efficiently, and optimally transfers data across the network in client/server applications.

C++ Example

```
ab.SetProperty (
"OUTPUT_FIELD_LIST","AddressLine|LASTLINE|PolygonName[COUN
TIES] | PolygonName[States] | POP00[Census2k]" );
```

RDI_DATAPATH Property

Specifies the file name and path for the Residential Delivery Indicator (RDI™) data.

Syntax

```
RDI_DATAPATH = path/<file>
```

Type

```
String of path and file names.
```

Notes

This property specifies the location of the RDI data.

This property is only required if you are using the RDI functionality in your application.

C++ Examples

```
RDI_DATAPATH = s:data\April05
```

REQUEST_LOG Property

Specifies a log file that contains a final summary of each request (client interaction with the server). Set by the server administrator, rather than by the client programmer.

Syntax

```
REQUEST_ LOG = "DriveLetter:\path\filename"
The log name must specify a file where the information will
be written.
```

Type

String of path and name of request log file.

Notes

This property specifies a log file that contains a final summary of each request (client interaction with the server). For each request, the following information is supplied:

- Request type.
- Request ID.
- Creation time.
- Client IP.
- Logical names used by the client.
- Username.
- Server handle number that processed the request.
- Number of records processed.
- Elapsed seconds on request queue, elapsed seconds being processed.
- Total seconds in the server.

Following is a typical entry in the request log:

```
Request type: Process Records. Request# 31. Create time: Tue Mar 23
09:51:48 2004. Client IP: 175.18.2.76.
Logical Names: GEOSTAN|GEOSTAN_Z9. User Name: .
Handle# 0. Num Records: 100. Elapsed seconds on queue: 0. Elapsed
seconds in processing: 1.
Total seconds in server: 1.
```

C++ Example

```
REQUEST_ LOG = "C:\work\request.log"
```

See Also

"REQUEST_LOG_OPTIONS Property" on page 387.

"LOG_ROLLOVER (server-only) Property" on page 379.

REQUEST_LOG_OPTIONS Property

Specifies the format of the request log and the delimiter that separates fields.

Syntax

```
REQUEST_ LOG_OPTIONS = <OUTPUT_FORMAT>|<DELIMITER>
```

Where `OUTPUT_FORMAT` defines the format of the request log output, either `BRIEF` or `VERBOSE`.

Where `DELIMITER` defines the delimiter that separates fields when using `BRIEF` format. Valid delimiters are as follows: `SEMICOLON`, `PIPE`, `COMMA`, or `TAB`. When the setting is `VERBOSE`, the delimiter is ignored.

Type

String of output format and delimiter type.

Notes

This property specifies the format of the request log and the delimiter that separates the fields.

When set to `BRIEF`, the request log file begins with a column header row with the names of the eleven columns. Each request is summarized in those 11 columns on its own row.

The default values are `VERBOSE` and `PIPE`.

C++ Examples

```
REQUEST_LOG_OPTIONS = BRIEF|PIPE
REQEUST_LOG_OPTIONS = VERBOSE
REQUEST_LOG_OPTIONS = BRIEF
```

See Also

"REQUEST_LOG Property" on page 386.

"LOG_ROLLOVER (server-only) Property" on page 379.

SPATIAL_PATHS Property

Delimited list of logical names, paths, and file names of Spatial+ data.

Syntax

```
SPATIAL_PATHS = [Logical_Name]path/<file>gsb
```

Type

String pairings of logical names with full path and file names.

Notes

The `SPATIAL_PATHS` property holds a delimited list of pairs.

The first element of the pair is a logical name and the second is the full path and file name of a Spatial+ (.gsb) file.

You can use `SPATIAL_PATHS` to specify the logicals for the GDL fields you want returned (for example, `GDLPolygonName` or `PolygonOverlap`).

GDL can use the `SPATIAL_PATHS` logical names to eliminate duplicate definitions for both Spatial and GDL.

This property is only required if Spatial+ is included in your Precisely license, and you are using spatial data in your application.

You can have more than one `SPATIAL_PATHS` properties in the `abserver.ini` file, with each property defining one or more logicals. If you have a large number of data sources, Group 1 recommends that you use more than one `SPATIAL_PATHS` property to avoid errors.

Initialization File Example

The `CANADA_PATHS` property is defined in the server.ini file only.

```
SPATIAL_PATHS =
[COUNTIES] "C:\Program Files\Precisely\COUNTIES.gsb” | \
[States]"C:\Program Files\Precisely\STATES.gsb”
```

STATUS_LOG Property

Describes general server events. Set by the server administrator, rather than by the client programmer.

Syntax

```
STATUS_LOG = "DriveLetter:\path\filename"
```

The file name specifies where the information is written.

Set AddressBroker's STATUS_LOG property to either of the following:

- The path and file name for a status log to save status messages.
- The value **CONSOLE** to display status messages to a console window.

Type

String of path and name of status log file.

Notes

The STATUS_LOG property specifies a log file that contains information about general server events, such as when the server was started and stopped. The property is set by the server administrator. When the status log reaches 2GB in size, AddressBroker starts writing over the status log from the beginning of the log.

C++ Example

```
STATUS_LOG = "C:\work\server.log"
STATUS_LEVEL = SERVER
```

Set AddressBroker's STATUS_LEVEL property to the appropriate level of message reporting you require:

- NONE—No messages. The least verbose.
- FATAL—Fatal errors, errors, and warnings.
- ERROR—Errors and warnings only.
- WARN—Warnings only.
- INFO—All information messages.
- DEBUG—Debug messages; for development only.
- SERVER—Server-level only messages. Default.

See Also

"REQUEST_LOG Property" on page 386.

“LOG_ROLLOVER (server-only) Property” on page 379.

15 – Fields

In this chapter

This chapter is a complete listing of AddressBroker fields. The chapter is divided into two sections, input fields and output fields. Within each section, fields are listed by type—GeoStan, GeoStan Canada, Spatial+, or Demographics Library.

The information in this chapter is primarily given in tables. The tables include the following information about each field: its character string name, data type, size, and a brief description.

Tables of input fields

The input fields available to your application depend on the values assigned to several AddressBroker properties. The list of available fields can be retrieved by using a `GetProperty` call with `ALL_INPUT_FIELDS` property as its argument. The value you assign to `AB_INPUT_MODE` property restricts the fields you can use for address input.

GeoStan input fields

Input String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description	INPUT_MODE
`AddressLine`	C	256	0	1st address line. For single line addresses, must include lastline information and can also include address information.	`AB_INPUT_NORMAL` `AB_INPUT_PARSED_LASTLINE`
`AddressLine2`	C	61	0	2nd address line.	`AB_INPUT_NORMAL` `AB_INPUT_PARSED_LASTLINE`
`ApnId`	C	46	0	Assessor's Parcel Number. **NOTE:** Reverse APN matching is only available with Centrus Points and Centrus APN data. This feature is not supported using MLD and MLD Extended Attributes data.	`AB_INPUT_NORMAL`

`CountyName`	C			Input county name used for geographic fallback.	`AB_INPUT_NORMAL`
					`AB_INPUT_PARSED`
					`AB_INPUT_PARSED_LASTLINE`

`City`	C	29	0	City name.	`AB_INPUT_PARSED` `AB_INPUT_PARSED_LASTLINE`
`CountyFips`	C	4	0	County FIPS.	`AB_INPUT_NORMAL`

`FirmName`	C	41	0	Company name.	`AB_INPUT_NORMAL`
					`AB_INPUT_PARSED`
					`AB_INPUT_PARSED_LASTLINE`

`LastLine`	C	61	0	Complete last address line.	`AB_INPUT_NORMAL`
`Latitude`	C	11	0	Latitude of located point (in millionths of degrees).	`AB_INPUT_NORMAL`

Input String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description	INPUT_MODE
`Longitude`	C	12	0	Longitude of located point (in millionths of degrees).	`AB_INPUT_NORMAL`
`Line1`	C	104	0	Address line 1.	`AB_INPUT_MULTILINE`
`Line2`	C	104	0	Address line 2	`AB_INPUT_MULTILINE`
`Line3`	C	104	0	Address line 3.	`AB_INPUT_MULTILINE`
`Line4`	C	104	0	Address line 4.	`AB_INPUT_MULTILINE`
`Line5`	C	104	0	Address line 5.	`AB_INPUT_MULTILINE`
`Line6`	C	104	0	Address line 6.	`AB_INPUT_MULTILINE`

`PBKEY`	C	16	0	PreciselyID unique identifier used as input for matching with Reverse PreciselyID Lookup. For more information, see "Reverse PreciselyID Lookup" on page 21.
				
				NOTE: This field is only available for the Master Location Dataset.
					`AB_INPUT_NORMAL`
					`AB_INPUT_PARSED`
					`AB_INPUT_PARSED_LASTLINE`

`RecordID`	C	32	0	User-provided unique record identifier.	`Any value`
`State`	C	3	0	State abbreviation.	`AB_INPUT_PARSED` `AB_INPUT_PARSED_LASTLINE`
`StateFips`	C	3	0	State FIPS.	`AB_INPUT_NORMAL`

`Urbanization Name`	C	31	0	Urbanization name for Puerto Rico.	`AB_INPUT_NORMAL`
					`AB_INPUT_PARSED`
					`AB_INPUT_PARSED_LASTLINE`

Input String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description	INPUT_MODE
ZIP	C	10	0	5-digit ZIP Code. If you have input files containing addresses with both 5-digit ZIP Codes and 9-digit ZIP Codes, use the” ZIP“ Input field to allow both formats.	AB_INPUT_PARSED AB_INPUT_PARSED_LASTLINE
ZIP4	C	5	0	4-digit ZIP Code extension (same field as for Demographics data).	AB_INPUT_PARSED AB_INPUT_PARSED_LASTLINE
ZIP9	C	10	0	9-digit ZIP (ZIP + 4).	AB_INPUT_PARSED AB_INPUT_PARSED_LASTLINE
ZIP10	C	11	0	9-digit ZIP (ZIP + 4) with hyphen.	AB_INPUT_PARSED AB_INPUT_PARSED_LASTLINE

GeoStan Canada input fields

Input String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description	INPUT_MODE
`AddressLine`	C	61	0	Address line.	AB_INPUT_NORMAL AB_INPUT_PARSED_LASTLINE
`City`	C	29	0	City name.	AB_INPUT_PARSED AB_INPUT_PARSED_LASTLINE
`Country`	C	29	0	Country name.	AB_INPUT_PARSED AB_INPUT_PARSED_LASTLINE
`LastLine`	C	61	0	Complete last address line.	AB_INPUT_NORMAL
`Municipality`	C	29	0	Canadian Municipality.	AB_INPUT_PARSED AB_INPUT_PARSED_LASTLINE
`PostalCode`	C	10	0	Canadian Postal Code.	AB_INPUT_PARSED AB_INPUT_PARSED_LASTLINE
`Province`	C	31	0	Canadian Province.	AB_INPUT_PARSED AB_INPUT_PARSED_LASTLINE
`State`	C	3	0	State abbreviation.	AB_INPUT_PARSED AB_INPUT_PARSED_LASTLINE
`ZIP`	C	10	0	7-digit Canadian Postal Code.	AB_INPUT_PARSED AB_INPUT_PARSED_LASTLINE

Spatial+ input fields

Input String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`Latitude`	N	11	6	Latitude of located point (in millionths of degrees) overrides geocoded value used in Spatial searches. See "Decimals in input/output field values" on page 67.
`Longitude`	N	12	6	Longitude of located point (in millionths of degrees) overrides geocoded value used in Spatial searches. See "Decimals in input/output field values" on page 67.

GDL input fields

Input String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`AddressLine`	C	61	0	1st address line.
`AddressLine2`	C	61	0	2nd address line.
`City`	C	29	0	City name.
`Country`	C	29	0	Country name.
`FirmName`	C	41	0	Company name.
`LastLine`	C	61	0	Complete last address line.
`Line1`	C	104	0	Address line 1.
`Line2`	C	104	0	Address line 2
`Line3`	C	104	0	Address line 3.
`Line4`	C	104	0	Address line 4.
`Line5`	C	104	0	Address line 5.
`Line6`	C	104	0	Address line 6.
`RecordID`	C	32	0	User-provided unique record identifier.
`State`	C	3	0	State abbreviation.
`UrbanizationName`	C	31	0	Urbanization name for Puerto Rico.
`ZIP`	C	10	0	5-digit ZIP Code.
`ZIP4`	C	5	0	4-digit ZIP Code extension (same field as for Demographics data).
`ZIP9`	C	10	0	9-digit ZIP (ZIP + 4).
`ZIP10`	C	11	0	9-digit ZIP (ZIP + 4) with hyphen.

Demographics input fields

Input String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`CensusBlockID`	C	16	0	15-digit census block ID, 12-digit block group, or an 11-digit census tract. Overrides geocoded value used in Demographics searches.
`ZIP9`	C	10	0	9-digit ZIP (ZIP + 4) (same field as for GeoStan data) overrides geocoded value used in Demographics searches.

Tables of output fields

The output fields available to your application depend on the values assigned to several AddressBroker properties. The list of available fields can be retrieved by using a **`GetProperty`** call with `ALL_OUTPUT_FIELDS` property as its argument.

Spatial+, Demographics, and GDL output fields require a logical name when assigned to the `OUTPUT_FIELD_LIST` property or when used as arguments to **`GetField`** or **`ResetField`**. See “Decimals in input/output field values” on page 67, “OUTPUT_FIELD_LIST Property” on page 384, and the **`GetField`** and **`ResetField`** function references in each API for more information. GeoStan and GeoStan Canada output fields do not require a logical name.

GeoStan output fields

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`AddressLine`	C	256	0	1st address line.
`AddressLine2`	C	61	0	2nd address line.
`AddressType`	C	2	0	Address Type regarding number of units: S – Single unit M – Multiple units P – Post Office box X – Unknown **NOTE:** This field is only available with the MLD Extended Attributes Dataset
`Alternate`	C	2	0	Base/Alternate record flag; B = base; A = alternate.
`ApnID`	C	46	0	Assessor’s parcel number
`AuxUserData`	C	301	0	User data from the auxiliary file. Blank if no auxiliary file.
`BlockSuffix`	C	2	0	Single character block suffix for split Tiger data blocks.
`CarrierRoute`	C	5	0	Carrier route.
`CenterlineLatitude`	C	11	0	Latitude of located point (in millionths of degrees) for a centerline match.
`CenterlineLongitude`	C	12	0	Longitude of located point (in millionths of degrees) for a centerline match.
`CenterlineNearestDistance`	C	8	0	Used differently with reverse geocoding and centerline matching: • *Reverse geocoding* – Distance, in feet, from the input location to the matched street segment, point address, or intersection. • *Centerline* – Distance, in feet, from the point-level match to the centerline match.
`CenterlineBearing`	C	6	0	Compass direction, in decimal degrees, from the point data match to the centerline match. Measured clockwise from 0 degrees north.

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`CenterlineSegmentID`	C	11	0	Unique 10-digit Segment ID for a centerline match assigned by the Street Network Provider: Tiger, HERE, or TomTom.
`CenterlinePreDir`	C	3	0	Prefix direction for a centerline match.
`CenterlineStreetName`	C	41	0	Street name for a centerline match.
`CenterlineStreetType`	C	5	0	Street type or suffix for a centerline match.
`CenterlinePostDir`	C	3	0	Postfix direction for a centerline match.
`CenterlineDataType`	C	3	0	The type of data used to make the centerline match. • 0 – USPS data • 1 – TIGER data • 2 – TomTom data • 3 – Sanborn point-level data • 4 – Deprecated • 6 – HERE data • 7 – TomTom point-level data • 8 - Centrus point-level data • 9 - Auxiliary file • 10 - User Dictionary • 11- HERE Point • 12 - Master Location Data
`CenterlineSegmentDir`	C	2	0	Unique 10-digit Segment ID for the centerline match assigned by the Street Network Provider: Tiger, HERE, or TomTom.

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`CenterlineIsAlias`	C	4	0	Centerline match located by an index alias. Returns 3 characters. The first is an N for normal street match or A for alias match (including buildings, aliases, firms, etc.). The next 2 characters are: • 01 – Basic index (normal address match) • 02 – USPS street name alias index • 03 – USPS building index • 04 – USPS firm name index • 05 – Statewide intersection alias match (when using the Usw.gsi or Use.gsi file) • 06 – Spatial data street name alias (requires the Us_pw.gsi, Us_pe.gsi, Us_psw.gsi, or Us_pse.gsi file) • 07 – Alternate index (when using Zip9.gsu, Zip9e.gsu, and Zip9w.gsu) • 08 – LACS^{Link} • 09 - Auxiliary file • 10 - Centrus Alias Data Set index (usca.gsi) • 11 - POI index file (poi.gsi) • 12 - USPS Preferred Alias • 13 - ZIPMove match (when using us.gsz) • 14 - Expanded Centroids match (when using us_cent.gsc and/or bldgcent.gsc)
`CenterlineBlockLeft`	C	16	0	Provides the Census FIPS Code that indicates the address is on the left side of the street for a centerline match.
`CenterlineBlockRight`	C	16	0	Provides the Census FIPS Code that indicates the address is on the right side of the street for a centerline match.
`CenterlineLeftBlockSuffix`	C	2	0	Current left Block suffix for Census 2010 Geography for a centerline match. Returns A or B. Only available in Centrus Enhanced data.
`CenterlineRightBlockSuffix`	C	2	0	Current right Block suffix for Census 2010 Geography for a centerline match. Returns A or B. Only available in Centrus Enhanced data.
`CBSADivisionName`	C	73	0	CBSA division name.

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`CBSADivisionNumber`	C	6	0	CBSA division number.
`CBSAname`	C	76	0	CBSA name.
`CBSAnumber`	C	6	0	CBSA name.
`CSAname`	C	4	0	CSA name.
`CSAnumber`	C	2	0	CSA name.
`CensusBlockID`	C	16	0	15-digit census block ID.
`CheckDigit`	C	2	0	Check digit.
`City`	C	29	0	City name.
`CityStateRecordName`	C	29	0	City Name for the matched address from the City State record.
`CMSAName`	C	31	0	CMSA name.
`CMSANumber`	C	5	0	CMSA number.
`ConfidenceSurfaceType`	C	16	0	Generates a confidence surface type based on information from a GeoStan match.
`Country`	C	29	0	Country name.
`CountyName`	C	128	0	County name.
`CrossStreetPostfixDirection`	C	3	0	Cross street postfix direction.
`CrossStreetPrefixDirection`	C	3	0	Cross street prefix direction.
`CrossStreetType`	C	5	0	Cross street type.

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`DataType`	C	3	0	The type of data used to make the match. The type of data used to make the centerline match. • 0 – USPS data • 1 – TIGER data • 2 – TomTom data • 3 – Sanborn point-level data • 4 – Deprecated • 6 – HERE data • 7 – TomTom point-level data • 8 - Centrus point-level data • 9 - Auxiliary file • 10 - User Dictionary • 11- HERE Point • 12- Master Location Data
`DefaultFlag`	C	2	0	Y = Either HiRiseDefault or RuralRouteDefault returned Y. Blank = Both HiRiseDefault and RuralRouteDefault returned N or Blank.
`DeliveryPointBarcode`	C	3	0	Delivery point barcode.
`DPVConfirm`	C	2	0	DPV confirmation indicator. N = Nothing confirmed. Y = Confirmed ZIP + 4, primary, and secondary. S = Confirmed ZIP + 4 and primary. D = Confirmed ZIP + 4 and primary and a default match (GS_HI_RISE_DFLT = Y). Blank = Non-matched input address to USPS ZIP + 4 data, or DPV data not loaded.
`DPVCMRA`	C	2	0	DPV CMRA indicator. Y = address found in CMRA table. N = Address not found in CMRA table. Blank = DPV not loaded
`DPVFalsePositive`	C	2	0	DPV false-positive indicator. Returns Y if a false-positive address match occurs.

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
DPVFootnote1	C	3	0	Provides information about matched DPV records. AA = ZIP + 4 matched record A1 = for failure to match a ZIP + 4 record Blank for address not presented to hash table or DPV data not loaded.
DPVFootnote2	C	3	0	Provides information about matched DPV records. BB = record where all DPV categories matched CC = DPV matched primary/house number, where the secondary unit number did not match (present but invalid) M1 = Missing primary/house number M3 = Invalid primary/house number N1 = DPV matched primary/house number, with a missing highrise secondary number P1 = Missing PS, RR, or HC Box number P3 = Invalid PS, RR or HC Box number F1 = All military addresses G1 = All general delivery addresses U1 = All unique ZIP Code addresses Blank = Address not presented to hash table or DPV data not loaded
DPVFootnote3	C	3	0	Provides information about matched DPV records. R1 = Matched to CMRA, without a Private mail box (PMB) RR = Matched to CMRA and PMB present. Blank = Address not presented to hash table or DPV data not loaded
DPVFootnote4	C	3	0	Reserved.
DPVFootnote5	C	3	0	Reserved by USPS for future use.
DPVFootnote6	C	3	0	Reserved by USPS for future use.
DPVVacancyStatus	C	3		DPV vacancy status. • Y – the address is vacant. • N – the address is not vacant. • Blank – DPV is not loaded or DPV did not confirm.

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`DPVNoStat`	C	3		Indicates that the address is not receiving delivery and the address is not counted as a possible delivery. These addresses are not receiving delivery because a) delivery has not been established; b) customer receives mail as part of a drop; or c) the address is no longer a possible delivery because the carrier destroys or returns all of the mail. Values: • Y - address was valid for computerized delivery sequence (CDS) pre-processing. • N - address not valid for CDS. • Blank - address not presented to No Stat table or DPV data not loaded.
`EWSMatch`	C	2	0	Y = Address record match denied because input record matched to EWS data. Blank = Input record did not match to EWS data.
`FIPSCountyCode`	C	6	0	FIPS code for county.
`FirmName`	C	41	0	Firm name.
`GeographicRank`	C			Geographic rank.
`GovernmentBuildingIndicator`	C	2	0	Government building indicator.
`HighEndHouseNumber`	C	12	0	House number at high end of range.
`HighUnitNumber`	C	12	0	High unit number.
`HiRiseDefault`	C	2	0	N = Matched to an exact high-rise record or a street record. Y= An exact record was not found. Matched to the USPS default high-rise record or a street record. Check the input address for accuracy and completeness. Blank = The flag does not apply to the input address (for example, PO Boxes and General Delivery addresses) or no match was found.
`HouseNumber`	C	12	0	House number of input address.
`HouseNumberHighSuffix`	C	7	0	House number high suffix of input address.

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`HouseNumberSuffix`	C	7	0	House number suffix of input address.
`IncorpPlaceInd`	C	2	0	Incorporated Place Indicator. I – Incorporated place N – Not an incorporated place X – Unknown **NOTE:** This field is only available with the MLD Extended Attributes Dataset
`Intersection`	C	2	0	Cross street match found indicated by flag (T,F).
`IsAlias`	C	4	0	Match record located by an index alias. Returns 3 characters. The first is an `N` for normal street match or A for alias match (including buildings, aliases, firms, etc.). The next 2 characters are: • 01 – Basic index (normal address match) • 02 – USPS street name alias index • 03 – USPS building index • 04 – USPS firm name index • 05 – Statewide intersection alias match (when using the Usw.gsi or Use.gsi file) • 06 – Spatial data street name alias (requires the Us_pw.gsi, Us_pe.gsi, Us_psw.gsi, or Us_pse.gsi file) • 07 – Alternate index (when using Zip9.gsu, Zip9e.gsu, and Zip9w.gsu) • 08 – LACS^{Link} • 09 - Auxiliary file • 10 - Centrus Alias Data Set index (usca.gsi) • 11 - POI index file (poi.gsi) • 12 - USPS Preferred Alias • 13 - ZIPMove match (when using us.gsz) • 14 - Expanded Centroids match (when using us_cent.gsc and/or bldgcent.gsc)
`LACSAddress`	C	2	0	L = LACS (Locatable Address Correction System) address

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`LACSLinkInd`	C	2	0	Provides information about matched $LACS^{Link}$ records. Y = Matched $LACS^{Link}$ record N = $LACS^{Link}$ match was NOT found F = False-positive $LACS^{Link}$ record S = Records where the secondary information (unit number) was removed to make a $LACS^{Link}$ match Blank = Records not processed through $LACS^{Link}$
`LACSLinkRetCode`	C	3	0	A = Matched $LACS^{Link}$ record 00 = $LACS^{Link}$ match was NOT found 14 = Found $LACS^{Link}$ match, but no $LACS^{Link}$ conversion 92 = The secondary information (unit number) was removed to make a $LACS^{Link}$ match Blank = not processed through $LACS^{Link}$
`LastLine`	C	61	0	Complete last address line.
`Latitude`	N	11	6	Latitude of located point (in millionths of degrees). See "Decimals in input/output field values" on page 67.
`LeftBlockSuffix`	C	2	0	Left side of block suffix. *Blank* if the matched record is from point-level data.
`Line1`	C	104	0	Address line 1. Available only when `INPUT_MODE = AB_INPUT_MULTILINE`.
`Line2`	C	104	0	Address line 2. Available only when `INPUT_MODE = AB_INPUT_MULTILINE.`
`Line3`	C	104	0	Address line 3. Available only when `INPUT_MODE = AB_INPUT_MULTILINE`.
`Line4`	C	104	0	Address line 4. Available only when `INPUT_MODE = AB_INPUT_MULTILINE`.
`Line5`	C	104	0	Address line 5. Available only when `INPUT_MODE = AB_INPUT_MULTILINE`.

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`Line6`	C	104	0	Address line 6. Available only when `INPUT_MODE = AB_INPUT_MULTILINE`.
`LocationQualityCode`	C	5	0	Code indicating the quality of location. See “GeoStan location codes” on page 433.
`Longitude`	N	12	6	Longitude of located point (in millionths of degrees). See “Decimals in input/output field values” on page 67.
`LOTCode`	C	2	0	Requires standardizable input address, A = ascending, D = descending.
`LOTNumber`	C	5	0	4-digit LOT number, requires standardizable input address
`LotSize`	N	11	0	Lot size of the parcel expressed in square feet; 0 if none. **NOTE:** This field is only available with the MLD Extended Attributes Dataset
`LotSizeMeters`	N	11	0	Lot size of the parcel expressed in square meters; 0 if none. **NOTE:** This field is only available with the MLD Extended Attributes Dataset
`LowEndHouseNumber`	C	12	0	House number at low end of range.
`LowUnitNumber`	C	12	0	Low unit number.
`MatchCode`	C	5	0	Match code. See“GeoStan return codes” on page 423.
`MatchedDB`	C			Returns the index of the GSD or User Dictionary matched to from GeoStan.
`MailStop`	C	61	0	Mail Stop.
`MCDName`	C	41	0	Minor Civil Division name from the auxiliary file. Blank if no auxiliary file.
`MCDNumber`	C	6	0	Minor Civil Division number from the auxiliary file. Blank if no auxiliary file

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`MECLat`	N	13	0	Latitude of Minimum Enclosing Circle expressed with an implied 6 digits of decimal precision; 0 if none. For example: 34809676 means 34.809676 **NOTE:** This field is only available with the MLD Extended Attributes Dataset.
`MECLon`	N	13	0	Longitude of Minimum Enclosing Circle expressed with an implied 6 digits of decimal precision; 0 if none. For example: -92447089 means -92.447089 **NOTE:** This field is only available with the MLD Extended Attributes Dataset.
`MECRadius`	N	12	1	Radius of Minimum Enclosing Circle (in square feet) expressed as a whole number. For example: 1234 means 1,234 feet. **NOTE:** This field is only available with the MLD Extended Attributes Dataset.
`MECRadiusMeters`	N	12	1	Radius of Minimum Enclosing Circle (in meters) expressed with 1 digit of decimal precision. For example: 123.4 meters **NOTE:** This field is only available with the MLD Extended Attributes Dataset.
`MetroFlag`	C	2	0	Metropolitan or micropolitan flag.
`MSA`	C	66	0	MSA or PMSA name.
`MSANumber`	C	5	0	MSA or PMSA number.
`ParcelCentroidElevation`	N	7	1	Elevation above sea level (in feet). For example: 125 feet.
`ParCenElevationMeters`	N	7	1	Elevation above sea level (in meters) expressed with 1 digit of decimal precision. For example: 12.5 meters. **NOTE:** This field is only available with the MLD Extended Attributes Dataset.

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`PBKEY`	C	16	0	PreciselyID unique identifier. This is a unique address identifier returned when an address match is returned from the Master Location Dataset. The leading character is a ‘P’. For example: P00001XSF1IF For a fallback pbKey, the leading character is an ‘X’. For example, X00001XSF1IF. For more information, see “PreciselyID Fallback” on page 20. **NOTE:** This field is only available for the Master Location Dataset.
`PMBDesignator`	C	5	0	Private mail box (PMB) designator. Field is not output if using multiline input mode.
`PMBNumber`	C	9	0	Private mail box (PMB) number. Field is not output if using multiline input mode.
`PointID`	C	11	0	Unique point ID of the matched record when matched to point-level data. *Blank* if the matched record is not from point-level data.
`PostfixDirection`	C	3	0	Postfix direction.
`PreferredCityName`	C	29	0	Preferred city name for the output ZIP Code of the matched address.
`PrefixDirection`	C	3	0	Prefix direction.
`RDIRetCode`	C	2	0	USPS Residential Delivery Indicator (RDI) return code description: • Y = Residence • N = Business • Blank = Not processed through RDI. t
`RecordID`	C	32	0	User-provided unique record identifier.

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`RecordType`	C	2	0	USPS range record type: • A = Auxiliary file • G = General Delivery • H = High Rise • F = Firm • S = Street • P = PO Box • R = Rural route/highway contract • T = Tiger file • U = User Dictionary
`ResidentialBusiness`	C	2	0	Usage Indicator: R – Residential use B – Business use M – Mixed use – residential and business X – Unknown use **NOTE:** This field is only available with the MLD Extended Attributes Dataset
`Right Block Suffix`	C	2	0	Right side of block suffix. *Blank* if the matched record is from point-level data.
`RoadClassCode`	C	3	0	Road class code: • 1 = major road, main data file. • 11 = major road, supplemental data file. • 0 = minor road, main data file. • 10 = minor road; supplemental file.
`RuralRouteDefault`	C	2	0	N = Matched to an exact rural route record. Y = An exact record was not found. Matched to the USPS default rural route record. Check the input address for accuracy and completeness. Blank = The flag does not apply to the input address (for example, PO Boxes and General Delivery addresses) or no match was found.
`SegmentBlockLeft`	C	16	0	Block on left side of segment.
`SegmentBlockRight`	C	16	0	Block on right side of segment.

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`Segment ID`	C	11	0	Unique 10-digit Segment ID assigned by the Street Network Provider: Tiger, HERE, or TomTom.
`Short Address Line`	C	61		Shortest possible address determined by CASS rules.
`Short Street Name`	C	41		Shortest possible street name determined by CASS rules.
`Short Cross Street Name`	C	41		Shortest possible cross street name determined by CASS rules.
`Short Cross Street Postfix Direction`	C	3		Shortest possible cross street postfix direction determined by CASS rules.
`Short Prefix Direction`	C	3		Shortest possible prefix direction determined by CASS rules.
`Short Cross Street Prefix Direction`	C	3		Shortest possible cross street prefix direction determined by CASS rules.
`Short Street Type`	C	5		Shortest possible street type determined by CASS rules.
`Short Cross Street Type`	C	5		Shortest possible cross street type determined by CASS rules.
`Short Postfix Direction`	C	3		Shortest possible postfix direction determined by CASS rules.
`Short City`	C	29		Shortest possible city name determined by CASS rules.
`Short Last Line`	C	61		Shortest possible last line determined by CASS rules.
`State`	C	31	0	State name.
`StreetName`	C	41	0	Street name.
`StreetSide`	C	2	0	The matched address is on the following side of the street: • *L* – Left side of the street • *R* – Right side of the street • *B* – Both sides of the street • *U* – Unknown side of the street This is relative to the segment endpoints and the segment direction.

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`StreetType`	C	5	0	Street type or suffix.
`SuiteLink RetCode`	C	4	0	SuiteLink Return Code. • A - SuiteLink record match. • 00 - No SuiteLink match. • Blank - This address was not processed through SuiteLink.
`TigerFaceID`	C	10	0	TIGER Face Identifier. This field can be used to match to all Census geocodes using external data; 0 if none. **NOTE:** This field is only available with the MLD Extended Attributes Dataset
`TigerPlace`	C	8	0	TIGER Place code; 0 if none. **NOTE:** This field is only available with the MLD Extended Attributes Dataset
`UnitNumber`	C	12	0	Unit number. **NOTE:** This field is only available with the MLD Extended Attributes Dataset
`UnitNumber2`	C	12	0	Second unit number parsed from the address line. Only available in CASS mode.
`UnitType`	C	5	0	Unit type.
`UnitType2`	C	5	0	Second unit type parsed from the address line. Only available in CASS mode.
`UrbanAreaID`	C	6	0	TIGER Urban Area Identifier. Defines the urban area if any; 0 if none. **NOTE:** This field is only available with the MLD Extended Attributes Dataset
`UrbanAreaPop`	N	11	0	Census population of the urban area; 0 if none. **NOTE:** This field is only available with the MLD Extended Attributes Dataset.

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
Urbanicity		2	0	Urbanicity Indicator. An indicator that defines, per the Census, the Urbanicity of the Address using TIGER UACE codes for categorization. L – Large Urban Area (50,000 or greater population) S – Small Urban Area (2,500-50,000 population) R – Rural X – Unknown **NOTE:** This field is only available with the MLD Extended Attributes Dataset.
UrbanizationName	C	31	0	Urbanization name for Puerto Rico.
USPSRangeRecordType	C	2	0	USPS range record type.
ZIP	C	6	0	5-digit ZIP Code.
ZIP4	C	5	0	4-digit ZIP Code extension.
ZIP9	C	10	0	9-digit ZIP Code (ZIP + 4).
ZIP10	C	11	0	9-digit ZIP Code (ZIP + 4) with dash separator.
ZIPCARRTSort	C	2	0	**September 2000 data and later:** • A = Automation cart allowed, optional cart merging allowed. • B = Automation cart allowed, no optional cart merging allowed. • C = No automation cart allowed, optional cart merging allowed. • D = No automation cart allowed, no optional cart merging allowed.
ZipCityDelivery	C	2	0	Indicates whether Post Office has city-delivery carrier routes (Y or N).

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
ZipClass	C	2	0	ZIP Classification Code: • blank = standard ZIP Code • M = Military ZIP Code • P = ZIP Code has PO Boxes only • U = Unique ZIP Code (ZIP assigned to a single organization).
ZipFacility	C	2	0	Returns the USPS State Name Facility Code.

GeoStan Canada output fields

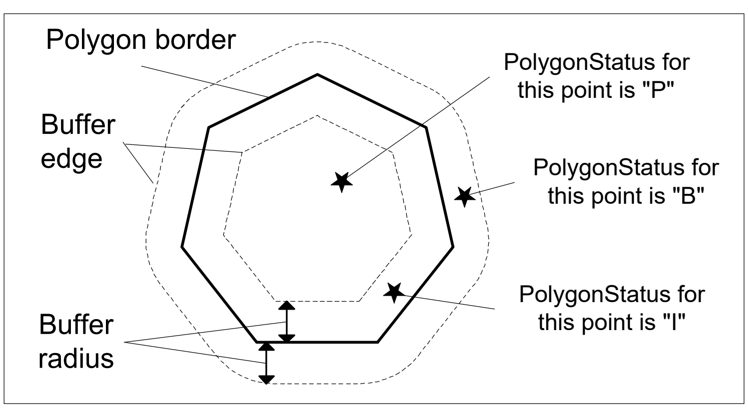
GeoStan Canada output fields are not available on AIX and Digital UNIX systems.

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
AddressLine	C	256	0	Address line.
City (same as Municipality)	C	29	0	City name.
LastLine	C	61	0	Complete last address line.
Latitude	N	11	6	Latitude of located point (in millionths of degrees). See “Decimals in input/output field values” on page 67.
LocationQualityCode	C	5	0	Code indicating the quality of location (see “GeoStan location codes” on page 433).

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`Longitude`	N	12	6	Longitude of located point (in millionths of degrees). See “Decimals in input/output field values” on page 67.
`MatchCode`	C	5	0	Match code (see “GeoStan return codes” on page 423).
`Municipality`	C	29	0	Canadian Municipality.
`PostalCode`	C	10	0	Canadian Postal Code.
`Province`	C	31	0	Canadian Province.
`State (same as State)`	C	31	0	State name.
`ZIP (same as PostalCode)`	C	6	0	5-digit ZIP Code.
`Country`	C	29	0	Country name.
`HighEndHouseNumber`	C	12	0	House number at high end of range.
`HighUnitNumber`	C	12	0	High unit number.
`HouseNumber`	C	12	0	House number.
`HouseNumberHighSuffix`	C	7	0	House number high suffix.
`HouseNumberLowSuffix`	C	7	0	House number low suffix.
`HouseNumberSuffix`	C	7	0	House number suffix.
`LowUnitNumber`	C	12	0	Low unit number.
`PostfixDirection`	C	3	0	Postfix direction.
`RecordID`	C	32	0	User-provided unique record identifier.
`StreetName`	C	41	0	Street name, lock box, Rural Route, or General Delivery address.
`StreetType`	C	5	0	Street type or suffix.
`UnitNumber`	C	12	0	Unit number.
`UnitType`	C	5	0	Unit type.

Spatial+ output fields

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`ClosestSiteBearing`	N	16	0	Bearing to nearest located points. (The maximum number of bearings returned is determined by the `MAXIMUM_POINTS` property.)
`ClosestSiteDistance`	N	16	0	Distances to nearest located points. (The maximum number of distances returned is determined by the `MAXIMUM_POINTS` property.)
`ClosestSiteID`	C	128	0	IDs of nearest located points. (The maximum number of IDs returned is determined by the `MAXIMUM_POINTS` property.)
`ClosestSiteName`	C	128	0	Names of nearest located points. (The maximum number of names returned is determined by `MAXIMUM_POINTS` property.)
`PolygonDistance`	N	16	0	Distance from border of located polygon, in feet. (The number of values returned is determined by the `MAXIMUM_POLYGONS` property.) This value determined the sort order of the returned polygons.
`PolygonName`	C	128	0	Names of located polygons. (The maximum number of names returned is determined by the `MAXIMUM_POLYGONS` property.)
`PolygonStatus`	C	2	0	Location of point with respect to polygon (see diagram, left). **P** = A polygon was found that contained the point. **B** = A buffer area, but not a polygon, was found that contained the point. **I** = The point is inside the polygon and the associated buffer. This can only occur for a polygon, not a line or point spatial object. (The number of values returned is determined by the `MAXIMUM_POLYGONS` property.) *Note:* Buffer radius is set in the `BUFFER_RADIUS` and `BUFFER_RADIUS_TABLE` properties.



Geographic Determination Library (GDL) output fields

Output String Field Name	Data Type	Width	Description
`GdlPolygonName`	String	128	Name of located polygon.
`LineFarDistance`	String	16	Far distance between located line and the geo-variance buffer (feet).
`LineName`	String	128	Name of located line.
`LineNearDistance`	String	16	Near distance between located line and the geo-variance buffer (feet).
`PointFarDistance`	String	16	Far distance between located point and the geo-variance buffer (feet).
`PointManhattanDistance`	String	16	Manhattan distance between geocoded point and located point (feet).
`PointName`	String	128	Name of located point.
`PointNearDistance`	String	16	Near distance between located point and the geo-variance buffer (feet).
`PointStraightDistance`	String	16	Straight-line distance between geocoded point and located point (feet).
`PolygonOverlap`	String	16	Overlap of located polygon and the geo-variance buffer (percent).

Demographic (Census 2010) output fields

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
AGGHHI10	N	16	0	2010 Aggregate Household ($000's)
AVGHHSZ10	N	11	2	2010 Average Household Size See "Decimals in input/output field values" on page 67.
FAM10	N	10	0	2010 Families
FEMPOP10	N	10	0	2010 Total Female Population
GQPOP10	N	10	0	2010 Population in Group Quarters
HHCHLD1810	N	10	0	2010 Total Households with Children under 18
HHOVER6010	N	10	0	2010 Total Households with Adults over 60
HH15TO2410	N	10	0	2010 Householder Age 15-24
HH25TO3410	N	10	0	2010 Householder Age 25-34
HH35TO4410	N	10	0	2010 Householder Age 35-44
HH45TO5410	N	10	0	2010 Householder Age 45-54
HH55TO6410	N	10	0	2010 Householder Age 55-64
HH65TO7410	N	10	0	2010 Householder Age 65-74
HH75TO8410	N	10	0	2010 Householder Age 75-84
HH85OVR10	N	10	0	2010 Householder Age 85+
HH10	N	10	0	2010 Households
HU10	N	10	0	2010 Total Housing Units
OWNOCCHU10	N	10	0	2010 Owner Occupied Housing Units
MEDAGE10	N	11	1	2010 Median Age See "Decimals in input/output field values" on page 67.
MEDAGEF10	N	11	1	2010 Median Age Female See "Decimals in input/output field values" on page 67.
MEDAGEM10	N	11	1	2010 Median Age Male See "Decimals in input/output field values" on page 67.
MEDFAMI10	N	10	0	2010 Median Family Income

Output String Field Name	Data Type N—numeric C—char string	Width—Number of characters including null terminator	Number of decimals if numeric	Description
`MEDHHI10`	N	10	0	2010 Median Household Income
`MEDHOMEV10`	N	10	0	2010 Median Housing Value
`POP10`	N	10	0	2010 Total Population
`XASNPOP10`	N	11	2	2010 % Asian/Pacific Islander Population See “Decimals in input/output field values” on page 67.
`XBLKPOP10`	N	11	2	2010 % Black Population See “Decimals in input/output field values” on page 67.
`XINDPOP10`	N	11	2	2010 % American Indian/Eskimo Population See “Decimals in input/output field values” on page 67.
`XWHTPOP10`	N	11	2	2010 % White Population See “Decimals in input/output field values” on page 67.
`BG`	N	12	0	Census 2010 Block Group index
`BG10`	N	12	0	Census 2010 Block Group output field
`CSUBFIPS10`	C	70	0	comma-delimited set of Census County Subdivision FIPS 55 Code values
`PLACEFPS10`	C	50	0	comma-delimited set of Place FIPS 55 Code values
`NECTA`	C	11	0	comma-delimited set of 2010 New England City and Town Area code values
`NECTA_NAME`	C	120	0	comma-delimited set of 2010 New England City and Town Area names
`DIVISION`	C	11	0	comma-delimited set of 2010 Division values
`DIVISION_N`	C	100	0	comma-delimited set of 2010 Division names

16 – Match codes

In this chapter

When you use Centrus® AddressBroker to perform address standardization, a match code is returned in the MatchCode output field. The match code is an alpha-numeric code that encapsulates information about the address standardization process—including whether or not a match was found, information about the type of match found (when applicable), and information about why no match was found (when applicable). This chapter provides the information you need to interpret these codes.

GeoStan return codes

The following table contains the match code values. You can find a description of the hex digits for the different match codes in the table following the match code table.

Code	Description
Ahhh	Same as Shhh, but indicates match to an alias name record or an alternate record.
Chh	Street address did not match, but located a street segment based on the input ZIP Code or city.
D00	Matched to a small town with P.O. Box or General Delivery only.
Ghhh	Matched to an auxiliary file.
Hhhh	House number was changed.
Jhhh	Matched to a User Dictionary.
P	Successful Reverse APN lookup match.
Qhhh	Matched to USPS range records with unique ZIP Codes. CASS rules prohibit altering an input ZIP if it matches a unique ZIP Code value.
Rhhh	Matched to a ranged address.
Shhh	Matched to USPS data. This is considered the best address match, because it matched directly against the USPS list of addresses. S is returned for a small number of addresses when the matched address has a blank ZIP + 4.
Thhh	Matched to a street segment record.
Uhhh	Matched to USPS data but cannot resolve the ZIP + 4 code without the firm name or other information. CASS mode returns an E023 (multiple match) error code.
Vhhh	Matched to MLD and DVDMLDR using Reverse PBKey Lookup. For match code values, see “Definitions for the Reverse PBKey Lookup “Vhhh” return code values” on page 426.
Xhhh	Matched to an intersection of two streets, for example, “Clay St & Michigan Ave.” The first hex digit refers to the last line information, the second hex digit refers to the first street in the intersection, and the third hex digit refers to the second street in the intersection. **NOTE:** The USPS does not allow intersections as a valid deliverable address.
Yhhh	Same as Xhhh, but an alias name record was used for one or both streets.
Z[a]	No address given, but verified the provided ZIP Code .

a Zh may be returned if Correct Last Line is set to True. For more information, see Correct last line match codes and Using correct last line.

Definitions for 1st-3rd hex digit match code values

The following table contains the description of the hex digits for the match code values.

Note: The third hex digit is only populated for intersection matches or as part of the Extended Match Code.

– For intersection matches, use the table below for the 3rd hex digit definition.
– For Extended Match Code, see Definitions for Extended Match Code (3rd hex digit) values in the next section.

Code	In first hex position means:	In second and third hex position means:
0	No change in last line.	No change in address line.
1	ZIP Code changed.	Street type changed.
2	City changed.	Predirectional changed.
3	City and ZIP Code changed.	Street type and predirectional changed.
4	State changed.	Postdirectional changed.
5	State and ZIP Code changed.	Street type and postdirectional changed.
6	State and City changed.	Predirectional and postdirectional changed.
7	State, City, and ZIP Code changed.	Street type, predirectional, and postdirectional changed.
8	ZIP + 4 changed.	Street name changed.
9	ZIP and ZIP + 4 changed.	Street name and street type changed.
A	City and ZIP + 4 changed.	Street name and predirectional changed.
B	City, ZIP, and ZIP + 4 changed.	Street name, street type, and predirectional changed.
C	State and ZIP + 4 changed.	Street name and postdirectional changed.
D	State, ZIP, and ZIP + 4 changed.	Street name, street type, and postdirectional changed.
E	State, City, and ZIP + 4 changed.	Street name, predirectional, and postdirectional changed.
F	State, City, ZIP, and ZIP + 4 changed.	Street name, street type, predirectional, and postdirectional changed.

Definitions for Extended Match Code (3rd hex digit) values

As mentioned in Understanding Extended Match Codes, when set to True, `MATCH_CODE_EXTENDED` returns additional information about any changes in the house number, unit number and unit type fields in the matched address, as well as whether there was address information that was ignored. This additional information is provided in a 3rd hex digit that is appended to match codes for address-level matches only - A, G, H, J, Q, R, S, T or U (see GeoStan return codes).

"Address information ignored" is specified when any of the following conditions apply:

- The output address has a mail stop (Mailstop).
- The output address has a second address line (AddressLine2).
- The input address is a dual address (two complete addresses in the input address). For example, "4750 Walnut St. P.O Box 50".
- The input last line has extra information that is not a city, state or ZIP Code, and is ignored. For example, "Boulder, CO 80301 USA", where "USA" is ignored when matching.

The following table contains the description of the 3rd hex digit Extended match code return values:

Note: For Auxiliary file matches, the 3rd hex digit is always "0".

Code	In 3rd hex position means:
0	Matched on all address information on line, including Unit Number and Unit Type if included.
1	Matched on Unit Number and Unit Type if included. Extra information on address line ignored. Extra information not considered for matching moved to `AddressLine2` or `Mail Stop` field.
2	Matched on Unit Number. Unit Type changed.
3	Matched on Unit Number. Unit Type changed. Extra information on address line ignored. Extra information not considered for matching moved to `AddressLine2` or `Mail Stop` field.
4	Unit Number changed or ignored.
5	Unit Number changed or ignored. Extra information on address line ignored. Extra information not considered for matching moved to `AddressLine2` or `Mail Stop` field.
6	Unit Number changed or ignored. Unit Type changed or ignored.
7	Unit Number changed or ignored. Unit Type changed or ignored. Extra information on address line ignored. Extra information not considered for matching moved to `AddressLine2` or `Mail Stop` field.
8	Matched on Unit Number and Unit Type if included. House Number changed or ignored.
9	Matched on Unit Number and Unit Type if included. House Number changed or ignored. Extra information on address line ignored. Extra information not considered for matching moved to `AddressLine2` or `Mail Stop` field.

Code	In 3rd hex position means:
A	Matched on Unit Number. Unit Type changed. House Number changed or ignored.
B	Matched on Unit Number. Unit Type changed. House Number changed or ignored. Extra information on address line ignored. Extra information not considered for matching moved to `AddressLine2` or `Mail Stop` field.
C	House Number changed or ignored. Unit Number changed or ignored.
D	House Number changed or ignored. Unit Number changed or ignored. Extra information on address line ignored. Extra information not considered for matching moved to `AddressLine2` or `Mail Stop` field.
E	House Number changed or ignored. Unit Number changed or ignored. Unit Type changed or ignored.
F	House Number changed or ignored. Unit Number changed or ignored. Unit Type changed or ignored. Extra information on address line ignored. Extra information not considered for matching moved to `AddressLine2` or `Mail Stop` field.

Definitions for the Reverse PBKey Lookup “Vhhh” return code values

The following table lists the “`Vhhh`” hex digit values returned with Reverse PBKey Lookup. For more information, see “Reverse PreciselyID Lookup” on page 21.

Match Code	Definition
`V000`	Match made using input pbKey. One Standard or Enhanced point address result returned depending on license.
`V001`	Match made using input pbKey. Multiple Standard and/or Enhanced point address variations results returned depending on license.
`V002`	Match made using input pbKey. One Standard, some Enhanced point address variations results returned depending on license.
`V003`	Match made using input pbKey. Multiple Standard, some Enhanced point address variations results depending on license.

Definitions for “Ennn” return code values

The following table describes the values returned when the application cannot find a match code

Code	Description
`Ennn`[a]	Indicates an error, or no match. This can occur when the address entered does not exist in the database, or the address is badly formed and cannot be parsed correctly. The last three digits of an error code indicate which parts of an address the application could not match to the database.
`nnn = 000`	No match made.

Code	Description
nnn = 001	Low level error.
nnn = 002	Could not find data file.
nnn = 003	Incorrect GSD file signature or version ID.
nnn = 004	GSD file out of date. Only occurs in CASS mode.
nnn = 010	No city and state or ZIP Code found.
nnn = 011	Input ZIP not in the directory.
nnn = 012	Input city not in the directory.
nnn = 013	Input city not unique in the directory.
nnn = 014	Out of licensed area. Only occurs if using Group 1 licensing technology.
nnn = 015	Record count is depleted and license has expired.
nnn = 020	No matching streets found in directory.
nnn = 021	No matching cross streets for an intersection match.
nnn = 022	No matching segments.
nnn = 023	Unresolved match.
nnn = 024	No matching segments. (Same as 022.)
nnn = 025	Too many possible cross streets for intersection matching.
nnn = 026	No address found when attempting a multiline match.
nnn = 027	Invalid directional attempted.
nnn = 028	Record also matched EWS data, therefore the application denied the match.
nnn = 029	No matching range, single street segment found
nnn = 030	No matching range, multiple street segments found
nnn = 040	No match found using input PBKey with Reverse PBKey Lookup.
nnn = 041	Not licensed to return Enhanced point address(es) found for input pbKey. Additional Reverse PBKey Lookup license option required to return results.

a Ehnn may be returned if Correct Last Line is set to True. For more information see Correct last line match codes and Using correct last line.

Correct last line match codes

As mentioned in Using correct last line, when set to True, `GS_FIND_CORRECT_LASTLINE` corrects elements of the output last line, providing a good ZIP Code or close match on the soundex even if the address would not match or was non-existent.

The feature works when `GS_FIND_ADDRCODE` is True and the address does not match a candidate or when `GS_FIND_Z_CODE` is True and only last line information is input. The match codes returned are similar to `Z` and `Ennn` in that the first letter remains the same with the second digit changing.

Code	Description
Zh	No address given, but verified the provided ZIP Code .
h = 0	No change in last line.
h = 1	ZIP Code changed.
h = 2	City changed.
h = 3	City and ZIP Code changed.
h = 4	State changed.
h = 5	State and ZIP Code changed.
h = 6	State and City changed.
h = 7	State, City, and ZIP Code changed.
h = 8	ZIP + 4 changed.
h = 9	ZIP and ZIP + 4 changed.
h = A	City and ZIP + 4 changed.
h = B	City, ZIP, and ZIP + 4 changed.
h = C	State and ZIP + 4 changed.
h = D	State, ZIP, and ZIP + 4 changed.
h = E	State, City, and ZIP + 4 changed.
Ehnn	Indicates an error, or no match. This can occur when the address entered does not exist in the database, or the address is badly formed and cannot be parsed correctly. The second digit of the error code is a hex digit which details the changes that were made to the last line information to correct the last line. The last two digits of an error code indicate which parts of an address the application could not match to the database.
h = 0	No change in last line.
h = 1	ZIP Code changed.
h = 2	City changed.
h = 3	City and ZIP Code changed.
h = 4	State changed.
h = 5	State and ZIP Code changed.
h = 6	State and City changed.
h = 7	State, City, and ZIP Code changed.

Code	Description
h = 8	ZIP + 4 changed.
h = 9	ZIP and ZIP + 4 changed.
h = A	City and ZIP + 4 changed.
h = B	City, ZIP, and ZIP + 4 changed.
h = C	State and ZIP + 4 changed.
h = D	State, ZIP, and ZIP + 4 changed.
h = E	State, City, and ZIP + 4 changed.
nn = 00	No match made.
nn = 01	Low level error.
nn = 02	Could not find data file.
nn = 03	Incorrect GSD file signature or version ID.
nn = 04	GSD file out of date. Only occurs in CASS mode.
nn = 10	No city and state or ZIP Code found.
nn = 11	Input ZIP not in the directory.
nn = 12	Input city not in the directory.
nn = 13	Input city not unique in the directory.
nn = 14	Out of licensed area. Only occurs if using Group 1 licensing technology.
nn = 15	Record count is depleted and license has expired.
nn = 20	No matching streets found in directory.
nn = 21	No matching cross streets for an intersection match.
nn = 22	No matching segments.
nn = 23	Unresolved match.
nn = 24	No matching segments. (Same as 022.)
nn = 25	Too many possible cross streets for intersection matching.
nn = 26	No address found when attempting a multiline match.
nn = 27	Invalid directional attempted.
nn = 28	Record also matched EWS data, therefore the application denied the match.
nn = 29	No matching range, single street segment found
nn = 30	No matching range, multiple street segments found

GeoStan Canada return codes

The following tables describe the codes returned in MatchCode when a match is found. The first character, an alphabetic element, describes the type of match found. The two- or three-digit numeric (or hexadecimal) element of the code provides detailed information about the match.

standardized address is the correct ZIP Code because GeoStan did not standardize the address; therefore, GeoStan does not return geocoding or Census Block information.

Values returned in MatchCode when a match is found

Return code	Explanation
Chhh	Indicates a match found in CPC data; changes were made to the address to make it deliverable.
Vhhh	Indicates a match found in CPC data; the input address is valid and no changes were made.

The returned address is the best address because it was matched directly against the CPC list of deliverable addresses. See below for the interpretation of the hex digits.

Code	First hex position indicates	Second and third hex position indicates
0	No change in last line.	No change in street type, direction, number, or name.
1	Postal Code was changed.	Street type was changed.
2	Municipality was changed.	Postfix direction was changed.
3	Municipality and Postal Code were changed.	Street type and Postfix direction were changed.
4	Province was changed.	House number was changed.
5	Province and Postal Code were changed.	Street type and House number were changed.
6	Province and Municipality were changed.	House number and Postfix direction were changed.
7	Province, Municipality, and Postal Code were changed.	Street type, Postfix direction, and House number were changed.
8	Reserved for future use.	Street name was changed.
9	Reserved for future use.	Street name and type were changed.
A	Reserved for future use.	Street name and Postfix direction were changed.
B	Reserved for future use.	Street name, Street type, and Postfix direction were changed.

Code	First hex position indicates	Second and third hex position indicates
C	Reserved for future use.	Street name and House number were changed.
D	Reserved for future use.	Street name, Street type, and House number were changed.
E	Reserved for future use.	Street name, House number and Postfix direction were changed.
F	Reserved for future use.	Street name, Street type, House number and Postfix direction were changed.

Code	Explanation
`Ecnn`	Indicates an error or no match. This can occur when the address entered either did not exist in the GeoStan Canada Directory, or the address was badly malformed and could not be passed correctly.
`nn = 01`	Internal error.
`nn = 10`	No Municipality+Province or Postal Code found.
`nn = 20`	No matching addresses.
`nn = 22`	Missing or wrong street name.
`nn = 23`	Could not resolve address.
`nn = 25`	Inconsistent address.
`nn = 26`	Missing or wrong box range.
`nn = 27`	Missing or wrong unit range.
`nn = 30`	Reverse lookup was performed
`nn = 40`	Address vs. Postal Code conflict, SERP rule prevents correction.
`nn = 41`	Postal Code has multiple street names, SERP rule prevents correction.
`nn = 42`	Change of delivery mode attempted, SERP rule prevents correction.
`nn = 43`	Total number of changed address elements exceeds maximum.
`nn = 44`	Change of address type is not allowed for current setting.
`nn = 50`	Minor error.
`nn = 66`	Invalid record. During validation, an error occurred.

17 – Location Codes

In this chapter

When you use AddressBroker to geocode your address data, a location quality code is returned in the LocationQualityCode output field. The location quality code encapsulates information about the geocoding process—including whether or not the address was assigned a geocode and its level of accuracy (when applicable). This chapter provides the information you need to interpret these codes.

GeoStan location codes

There are two types of geocodes: Address, and ZIP centroids. The value “E” is assigned when no location is available.

Address geocodes are simple to interpret because they indicate a geocode made directly to a street network segment (or two segments, in the case of an intersection). ZIP centroids, however, have a range of “confidence” depending on how the centroid was determined.

“E” indicates that no location is available. There are a number of situations that resolve to this code including:

- Submitting an input address that had no ZIP Code information or otherwise failed to standardize. The ZIP Code returned with the non-standardized address cannot be assumed to be the correct ZIP Code because the address was not standardized; therefore, no geocoding or Census Block information is returned.
- Requesting ZIP Code centroids of a high quality, and one is not available for that match.
- Requesting location quality codes when you are not licensed for geocoding.
- Requesting ZIP Code centroids, and no matching 5-digit ZIP Code is found (infrequent) in the z9 file.

Address location codes

Address location codes detail the known qualities about the geocode. An address location code has the following characters.

1st character	Always an A indicating an address location.	
2nd character	May be one of the following	
	C	Interpolated address point location.
	G	Auxiliary file data location.
	I	Application infers the correct segment from the candidate records.
	P	Point-level data location.
	R	Location represents a ranged address.
	S	Location on a street range.
	X	Location on an intersection of two streets.
3rd and 4th characters	Digit indicating other qualities about the location.	

The following table contains the address codes.

Code		Description
AGn		Indicates an auxiliary file for a geocode match where n is one of the following values:
	n=0	The geocode represents the center of a parcel or building.
	n=1	The geocode is an interpolated address along a segment.
	n=2	The geocode is an interpolated address along a segment, and the side of the street cannot be determined from the data provided in the auxiliary file record.
	n=3	The geocode is the midpoint of the street segment.
APnn		Indicates a point-level geocode match representing the center of a parcel or building, where nn is one of the following values:
	nn=00	User Dictionary centroid. Geocode returned by a User Dictionary.
	nn=02	Parcel centroid Indicates the center of an assessor's parcel (tract or lot) polygon. When the center of an irregularly shaped parcel falls outside of its polygon, the centroid is manually repositioned to fall inside the polygon as closely as possible to the actual center.
	nn=04	Address point Represents field-collected GPS points with field-collected address data.
	nn=05	Structure point Indicates a location within a building footprint polygon that is associated with the matched address. Usually, residential addresses consist of a single building. For houses with outbuildings (detached garages, sheds, barns, etc.), the structure point will typically fall on the primary structure. Condominiums and duplexes have multiple, individual addresses and may have multiple structure points for each building. Multi-unit buildings are typically represented by a single structure point associated with the primary/base address, rather than discrete structure points for each unit. Shopping malls, industrial complexes, and academic or medical center campuses are commonly represented by a single structure point associated with the primary/base address for the entire complex. When multiple addresses are assigned to multiple buildings within one complex, multiple structure points may be represented within the same complex.
	nn=07	Manually placed Address points are manually placed to coincide with the midpoint of an assessor's parcel's street frontage at a distance from the center line.
	nn=08	Front door point Represents the designated primary entrance to a building. If a building has multiple entrances and there is no designated primary entrance or the primary entrance cannot readily be determined, the primary entrance is chosen based on proximity to the main access street and availability of parking.
	nn=09	Driveway offset point Represents a point located on the primary access road (most commonly a driveway) at a perpendicular distance of between 33-98 feet (10-30 meters) from the main roadway.

Code		Description
	nn=10	Street access point Represents the primary point of access from the street network. This address point type is located where the driveway or other access road intersects the main roadway.
	nn=21	Base parcel point The Centrus point data includes individual parcels that may be "stacked".These stacked parcels are individually identified by their unit or suite number, and GeoStan is able to match to this unit number and return the correct APN.If an input address is for a building or complex, without a unit number.The "base" parcel information returns and will not standardize to a unit number or return additional information such as an APN.
	nn=22	Backfill address point The precise parcel centroid is unknown. The address location assigned is based on two known parcel centroids.
	nn=23	Virtual address point The precise parcel centroid is unknown. The address location assigned is relative to a known parcel centroid and a street segment end point.
	nn=24	Interpolated address point The precise parcel centroid is unknown. The address location assigned is based on street segment end points.
AIn		The correct segment is inferred from the candidate records at match time.
ASn		House range address geocode. This is the most accurate street interpolated geocode available.
AIn, ASn, and ACnh share the same values for the 3rd character "n" as follows:		
	n=0	Best location.
	n=1	Street side is unknown. The Census FIPS Block ID is assigned from the left side; however, there is no assigned offset and the point is placed directly on the street.
	n=2	Indicates one or both of the following: • The address is interpolated onto a TIGER segment that did not initially contain address ranges. • The original segment name changed to match the USPS spelling. This specifically refers to street type, predirectional, and postdirectional. **Note:** Only the second case is valid for non-TIGER data because segment range interpolation is only completed for TIGER data.
	n=3	Both 1 and 2.
	n=7	Placeholder. Used when starting and ending points of segments contain the same value and shape data is not available.
ACnh		Indicates a point-level geocode that is interpolated between 2 parcel centroids (points), a parcel centroid and a street segment endpoint, or 2 street segment endpoints.
	The ACnh 4th character "h" values are as follows:	
	h=0	Represents the interpolation between 2 points, both coming from User Dictionaries.

Code		Description
	h=1	Represents the interpolation between 2 points. The low boundary came from a User Dictionary and the high boundary, from a non-User Dictionary.
	h=2	Represents the interpolation between 1 point and 1 street segment end point, both coming from User Dictionaries.
	h=3	Represents the interpolation between 1 point (low boundary) and 1 street segment end point (high boundary). The low boundary came from a User Dictionary and the high boundary from a non-User Dictionary.
	h=4	Represents the interpolation between 2 points. The low boundary came from a non-User Dictionary and the high boundary from a User Dictionary.
	h=5	Represents the interpolation between 2 points, both coming from non-User Dictionaries.
	h=6	Represents the interpolation between 1 point (low boundary) and 1 street segment end point (high boundary). The low boundary came from a non-User Dictionary and the high boundary from a User Dictionary.
	h=7	Represents the interpolation between 1 point and 1 street segment end point and both came from non-User Dictionaries.
	h=8	Represents the interpolation between 1 street segment end point and 1 point, both coming from User Dictionaries.
	h=9	Represents the interpolation between 1 street segment end point (low boundary) and 1 point (high boundary). The low boundary came from a User Dictionary and the high boundary from a non-User Dictionary.
	h=A	Represents the interpolation between 2 street segment end points, both coming from User Dictionaries.
	h=B	Represents the interpolation between 2 street segment end points. The low boundary came from a User Dictionary and the high boundary from a non-User Dictionary.
	h=C	Represents the interpolation between 1 street segment end point (low boundary) and 1 point (high boundary). The low boundary came from a non-User Dictionary and the high boundary from a User Dictionary.
	h=D	Represents the interpolation between 1 street segment end point and 1 point, both coming from non-User Dictionary.
	h=E	Represents the interpolation between 2 street segment end points. The low boundary came from a non-User Dictionary and the high boundary from a User Dictionary.
	h=F	Represents the interpolation between 2 street segment end points, both coming from non-User Dictionaries.
ARn		Ranged address geocode, where “n” is one of the following:
	n=1	The geocode is placed along a single street segment, midway between the interpolated location of the first and second input house numbers in the range.
	n=2	The geocode is placed along a single street segment, midway between the interpolated location of the first and second input house numbers in the range, and the side of the street is unknown. The Census FIPS Block ID is assigned from the left side; however, there is no assigned offset and the point is placed directly on the street.

Code		Description
	n=4	The input range spans multiple USPS segments. The geocode is placed on the endpoint of the segment which corresponds to the first input house number, closest to the end nearest the second input house number.
	n=7	Placeholder. Used when the starting and ending points of the matched segment contain the same value and shape data is not available.
AXn		Intersection geocode, where "n" is one of the following:
	n=3	Standard single-point intersection computed from the center lines of street segments.
	n=8	Interpolated (divided-road) intersection geocode. Attempts to return a centroid for the intersection.

Street centroid location codes

Street centroid location codes indicate the Census ID accuracy and the position of the geocode on the returned street segment. A street centroid location code has the following characters.

1st character	Always "C" indicating a location derived from a street segment.
2nd character	Census ID accuracy based on the search area used to obtain matching Street Segment.
3rd character	Location of geocode on the returned street segment.

The following table contains the values and descriptions for the location codes.

Character position	Code	Description
2nd Character		
	B	Block Group accuracy (most accurate). Based on input ZIP Code.
	T	Census Tract accuracy. Based on input ZIP Code.
	C	Unclassified Census accuracy. Normally accurate to at least the County level. Based on input ZIP Code.
	F	Unknown Census accuracy. Based on Finance area.
	P	Unknown Census accuracy. Based on input City.
3rd Character		
	C	Segment Centroid.
	L	Segment low-range end point.
	H	Segment high-range end point.

ZIP + 4 centroid location codes

ZIP + 4® centroid location codes indicate the quality of two location attributes: Census ID accuracy and positional accuracy. A ZIP + 4 centroid location code has the following characters.

1st character	Always “Z” indicating a location derived from a ZIP centroid.
2nd character	Census ID accuracy.
3rd character	Location type.
4th character	How the location and Census ID was defined. Provided for completeness, but may not be useful for most applications.

The following table contains the values and descriptions for the location codes.

Character position	Code	Description
2nd Character		
	B	Block Group accuracy (most accurate).
	T	Census Tract accuracy.
	C	Unclassified Census accuracy. Normally accurate to at least the County level.
3rd Character		
	5	Location of the Post Office that delivers mail to the address, a 5-digit ZIP Code centroid, or a location based upon locale (city). See the 4th character for a precise indication of locational accuracy.
	7	Location based upon a ZIP + 2 centroid. These locations can represent a multiple block area in urban locations, or a slightly larger area in rural settings.
	9	Location based upon a ZIP + 4 centroid. These are the most accurate centroids and normally place the location on the correct block face. For a small number of records, the location may be the middle of the entire street on which the ZIP + 4 falls. See the 4th character for a precise indication of locational accuracy.
4th Character		
	A	Address matched to a single segment. Location assigned in the middle of the matched street segment, offset to the proper side of the street.
	a	Address matched to a single segment, but the correct side of the street is unknown. Location assigned in the middle of the matched street segment, offset to the left side of the street, as address ranges increase.
	B	Address matched to multiple segments, all segments have the same Block Group. Location assigned to the middle of the matched street segment with the most house number ranges within this ZIP + 4. Location offset to the proper side of the street.

Character position	Code	Description
	b	Same as methodology B except the correct side of the street is unknown. Location assigned in the middle of the matched street segment, offset to the left side of the street, as address ranges increase.
	C	Address matched to multiple segments, with all segments having the same Census Tract. Returns the Block Group representing the most households in this ZIP + 4. Location assigned to t he middle of the matched street segment with the most house number ranges within this ZIP + 4. Location offset to the proper side of the street.
	c	Same as methodology C except the correct side of the street is unknown. Location assigned in the middle of the matched street segment, offset to the left side of the street, as address ranges increase.
	D	Address matched to multiple segments, with all segments having the same County. Returns the Block Group representing the most households in this ZIP + 4. Location assigned to the middle of the matched street segment with the most house number ranges within this ZIP + 4. Location offset to the proper side of the street.
	d	Same as methodology D except the correct side of the street is unknown. Location assigned in the middle of the matched street segment, offset to the left side of the street, as address ranges increase.
	E	Street name matched; no house ranges available. All matched segments have the same Block Group. Location placed on the segment closest to the center of the matched segments. In most cases, this is on the mid-point of the entire street.
	F	Street name matched; no house ranges available. All matched segments have the same Census Tract. Location placed on the segment closest to the center of the matched segments. In most cases, this is on the mid-point of the entire street.
	G	Street name matched (no house ranges available). All matched segments have the same County. Location placed on the segment closest to the center of the matched segments. In most cases, this is on the mid-point of the entire street.
	H	Same as methodology G, but some segments are not in the same County. Used for less than .05% of the centroids.
	I	Created ZIP + 2 cluster centroid as defined by methodologies A, a, B, and b. All centroids in this ZIP + 2 cluster have the same Block Group. Location assigned to the ZIP + 2 centroid.
	J	Created ZIP + 2 cluster centroid as defined by methodologies A, a, B, b, C, and c. All centroids in this ZIP + 2 cluster have the same Census Tract. Location assigned to the ZIP + 2 centroid.
	K	Created ZIP + 2 cluster centroid as defined by methodologies A, a, B, b, C, c, D, and d. Location assigned to the ZIP + 2 centroid.
	L	Created ZIP + 2 cluster centroid as defined by methodology E. All centroids in this ZIP + 2 cluster have the same Block Group. Location assigned to the ZIP + 2 centroid.
	M	Created ZIP+2 cluster centroid as defined by methodology E and F. All centroids in this ZIP + 2 cluster have the same Census Tract. Location assigned to the ZIP + 2 centroid.

Character position	Code	Description
	N	Created ZIP + 2 cluster centroid as defined by methodology E, F, G, and H. Location assigned to the ZIP + 2 centroid.
	O	ZIP Code is obsolete and currently not used by the USPS. Historic location assigned.
	V	Over 95% of addresses in this ZIP Code are in a single Census Tract. Location assigned to the ZIP Code centroid.
	W	Over 80% of addresses in this ZIP Code are in a single Census Tract. Reasonable Census Tract accuracy. Location assigned to the ZIP Code centroid.
	X	Less than 80% of addresses in this ZIP Code are in a single Census Tract. Census ID is uncertain. Location assigned to the ZIP Code centroid.
	Y	Rural or sparsely populated area. Census code is uncertain. Location based upon the USGS places file.
	Z	P.O. Box or General Delivery addresses. Census code is uncertain. Location based upon the Post Office location that delivers the mail to that address

Geographic centroid location codes

Geographic centroid location codes indicate the quality of two location attributes: the geographic location and area type

1st character	Always "G" indicating a location derived from a geographic centroid.
2nd character	Geographic area type.

The following table contains the values and descriptions for the location codes.

Character position	Code	Description
2nd Character		
	M	Municipality (city).
	C	County.
	S	State.

GeoStan Canada location codes

If a valid Postal Code centroid is found, one of the following location codes is returned:

Code	Description
CAN6	Postal Code level geocode.
EC	Indicates that a geocode is unavailable.

18 – Status Codes

In this chapter

This chapter describes status codes, messages, and exceptions you may receive from AddressBroker.

AddressBroker status codes and messages provide an indication of the relative success of executing AddressBroker functions. These status codes and messages allow your code to handle problems that may arise during the execution of AddressBroker.

Understanding AddressBroker status codes

Most of the functions and methods in the C, C++, and ActiveX interfaces return a Boolean value **TRUE** (or **1**) to indicate successful execution. If **FALSE** is returned or an exception is thrown, use the **GetStatus** function to retrieve the status code and message.

Note: AddressBroker Java clients use a different mechanism for handling errors. Java exceptions are described in "AddressBroker Java exceptions" on page 139.

Status codes have ten digits. The code parses into the following categories: success status, severity, facility, message number, and a low-level product code. See the figure below, which explains how to interpret the code. Precisely recommends handling status codes based upon status severity:

Informative (**0**) – No special handling of these codes or messages required.

Status code and message provided for informational purposes only.

Warning (**1**) – Generally, no special handling of these codes or messages required.

The continued execution of AddressBroker does not result in any problems. However, the status code and message indicates a situation you need to be aware of. The most common occurrences of this status severity include:

- Calling **GetRecord** when no output records are available.
- Calling **GetField** when a multi-valued output field contains no further values.
- Calling **LookupRecord** when the match is not completely successful (i.e., the address line or last line are not completely resolved).

Severe (**2**) – Requires corrective action.

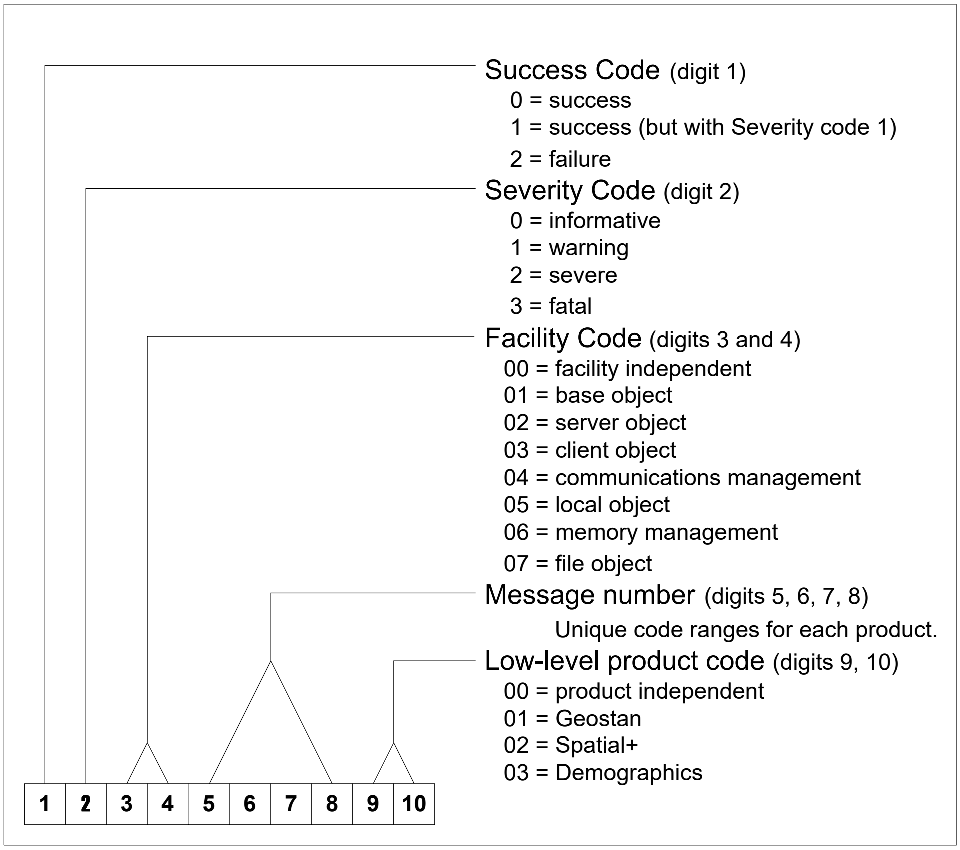
If corrective action is taken, continued execution of AddressBroker is possible. The most common occurrences of this status severity include:

- Calling **SetProperty** with invalid property names.
- **ValidateProperties** is to be unable to validate one or more properties.
- Calling **SetField** with invalid field names or values.
- Calling **LookupRecord** or **ProcessRecords** with invalid input (missing records).

Fatal (**3**) – Continued execution of AddressBroker results in unrecoverable errors.

Generally, these errors occur from internal memory management problems. Verify that your computer has sufficient memory to execute the AddressBroker application. If problems persist after correcting any memory management problems, please contact Precisely for assistance. Be sure to report the status code number and any status messages.

The example below shows the significance of each digit in the status code. Example status codes provides some example status codes. Explanation of digits in status codes



Example status codes

0000000000 (returned as 0)	Success code: 0 successful completion
	Severity code: 0 informative
	Facility code: 00 facility independent
	Message code: 0000 no message
	Product code: 00 product independent

1101020100	Success code: 1 successful completion
	Severity code: 1 warning
	Facility code: 01 base object
	Message code: 0201 message available
	Product code: 00 product independent
2302000600	Success code: 2 failure
	Severity code: 3 severe
	Facility code: 02 server object
	Message code: 0006 message available
	Product code: 00 product independent
2306105200	Success code: 2 failure
	Severity code: 3 fatal
	Facility code: 06 memory management
	Message code: 1052 message available
	Product code: 00 product independent
2305001001	Success code: 2 failure
	Severity code: 3 fatal
	Facility code: 05 local object
	Message code: 0010 message available
	Product code: 01 low-level GeoStan error

A – Advanced Concepts

In this appendix

This appendix discusses optional AddressBroker processing modes and other features you might want to incorporate into your applications.

Address line input modes

xx

xx no space2

x3

x4

AddressBroker provides four ways of entering your address information: two-line (normal), two-line parsed last line, parsed fields, and multiline. You select one of these predefined input modes with the AddressBroker `INPUT_MODE` property.

The input fields you select must be compatible with the value you assign to `INPUT_MODE`. For a complete listing of fields to use with `INPUT_MODE`, see “Tables of input fields” on page 392.

The output fields available depend on the settings of AddressBroker’s `INPUT_MODE` and the `OUTPUT_FIELD_LIST` properties.

The input mode you choose affects another AddressBroker property—`ADDRESS_PREFERENCE` (see page 450).

Two-line input mode

Two-line (normal) address input mode lets you pass two address lines as input to AddressBroker. For two-line matching, use the following fields and settings:

- Address input fields = `AddressLine`, `AddressLine2`, `LastLine`

  The information in these fields may include, but is not limited to: house number, street, unit number, PO Box, firm name, city, state, and ZIP Code. If your data includes addresses in Puerto Rico, you may also use the UrbanizationName field. If your data includes addresses in Canada, you may also use the Province, Municipality, and Postal Code fields.

- Set AddressBroker’s `INPUT_MODE` property to **AB_INPUT_NORMAL** (default).

AddressBroker extracts information from `AddressLine` and `AddressLine2`. A standardized address is returned (when possible) in both the `AddressLine` and `AddressLine2` output fields. LastLine address information is standardized and returned in the `LastLine` output field.

Sometimes a two-line input address is returned in a single output field. For example, given the input:

25 Main (`AddressLine`)

Set 200 (`AddressLine2`)

New York, NY (`LastLine`)

AddressBroker returns:

123 Main St Ste 200 (`AddressLine`)

(`AddressLine2`)

New York, NY, 10044-0052 (`LastLine`)

In this example, the unit number and type have been appended to the first line.

Two-line parsed last line input mode

Two-line parsed last line address input mode is the same as two-line input, except the `LastLine` field is replaced by the `City`, `State` and `ZIP Code` input fields. For two-line parsed last line matching, use the following fields and settings:

- Address input fields = `AddressLine`, `AddressLine2`, `City`, `State`, and any of the ZIP Code fields.

  The information in these fields may include, but is not limited to: house number, street, unit number, PO Box, firm name, city and state or ZIP Code. If your data includes addresses in Puerto Rico, you may also use the `UrbanizationName` field. If your data includes addresses in Canada, you may also use the `Province`, `Municipality`, and Postal Code fields.

- Set AddressBroker's `INPUT_MODE` property to **AB_INPUT_PARSED_LASTLINE**.

  AddressBroker extracts information from `AddressLine` and `AddressLine2`. A standardized address is returned (when possible) in both the `AddressLine` and `AddressLine2` output fields. City, State, and ZIP Code address information is standardized and returned in either the `LastLine` or the `City`, `State`, and ZIP Code output fields.

An example of two-line parsed last line input looks like this:

25 Main (`AddressLine`)

Ste 200 (`AddressLine2`)

New York (`City`)

NY (`State`)

AddressBroker returns:

123 Main St Ste 200 (`AddressLine`)

New York (`City`)

NY (`State`)

10044-0052 (`ZIP10`)

Multiline input mode

Multiline input mode lets you pass up to six address lines as input to AddressBroker. For multiline input mode, use the following fields and settings:

- Address Input fields: `Line1`, `Line2`, `Line3`, `Line4`, `Line5`, `Line6` (not all fields are required)
- Set AddressBroker's `INPUT_MODE` property to **AB_INPUT_MULTILINE**.

In multiline input mode, you do not use the `LastLine` input field for City, State, and ZIP Code data. AddressBroker extracts information from `Line1...6`. A standardized address is returned (when possible) in the `AddressLine` output field. City, State, and ZIP Code information is standardized and returned in the `LastLine` output field. Extraneous (non-address) information is returned in the `Line1... Line6` output fields. The `AddressLine2` output field is always empty. For example, given the input:

Mary Doe (`Line1`)

25 Main St (`Line2`)

Suite 200 (`Line3`)

Deliver around back (`Line4`)

New York (`Line5`)

NY (`Line6`)

AddressBroker returns:

25 Main St Suite 200(`AddressLine`)

New York, NY, 10044-0052(`LastLine`)

Mary Doe(`Line1`)

Deliver around back(`Line4`)

In this example, the `Line2`, `Line3`, `Line5`, `Line6`, and `AddressLine2` output field values are empty. You can also use **`GetField`** with "parsed" output fields as arguments to retrieve address elements individually.

Note: GeoStan Canada does not support multiline processing.

Address preference

Some address entries in your database may contain both a street address and a PO Box. Use AddressBroker's `ADDRESS_PREFERENCE` property to configure AddressBroker to prefer the PO Box, street, or bottommost address line, when returning a match. Basically, whichever preference you select is returned in the `AddressLine` output field, provided the address was matched. If only one of the addresses matched, the match is returned in the `AddressLine` output field, regardless of the value assigned to `ADDRESS_PREFERENCE`. AddressBroker's `ADDRESS_PREFERENCE` property has no effect when an entry has only one address (street *or* PO Box, but not both).

When both addresses match, the fields in which the "non-preferred" address information is returned depend on the value of `ADDRESS_PREFERENCE` and the value assigned to `INPUT_MODE` (see "Address line input modes" on page 447).

Address preference with two-line input mode

Consider this example address:

25 Main Suite 200(`AddressLine`)

Box 100(`AddressLine2`)

New York, NY, 10044(`LastLine`)

With `ADDRESS_PREFERENCE` set to prefer a street address, the output field values are:

25 Main St Suite 200(`AddressLine`)

PO Box 100(`AddressLine2`)

New York, NY, 10044-0052(`LastLine`)

Both address lines are valid addresses. Both `AddressLine` and `AddressLine2` have been standardized. The `Lastline` field has also been standardized. If you had set `ADDRESS_PREFERENCE` to prefer a PO Box, the output field values would be:

PO Box 100(`AddressLine`)

25 Main Suite 200(`AddressLine2`)

New York, NY, 10008(`LastLine`)

Now consider the next example address:

000 Main Suite 200(`AddressLine`)

Box 100(`AddressLine2`)

New York, NY(`LastLine`)

The information in `AddressLine` is not a valid address, and does not return a match. The output field values for this example are:

PO Box 100	(`AddressLine`)
000 Main Suite 200	(`AddressLine2`)
New York, NY, 10008	(`LastLine`)

When only one address matches, it is returned in the `AddressLine` output field, regardless of the `ADDRESS_PREFERENCE` setting. AddressBroker always attempts to return a matched address rather than no match. In this example, the `AddressLine` and `LastLine` output fields have been standardized. The unmatched address information is returned in the `AddressLine2` output field. It is not standardized.

The first three rows in the following table show what values are returned when two valid addresses are submitted in an address record for each of the preference modes. The table below also explains which output fields hold the data and whether or not the data has been processed.

The second three rows in the table below show what values are returned when two addresses are submitted, but only one is matched. Two-line input mode:

ADDRESS PREFERENCE =	input field = AddressLine	output field = AddressLine	Std.[1]	input field = AddressLine2	output field = AddressLine2	Std.[1]
AB_ADDRESS_STREET	25 Main	25 Main St	Y	BOX 100	PO BOX 100	Y
AB_ADDRESS_POBOX	25 Main	PO BOX 100	Y	BOX 100	25 Main St	Y
AB_ADDRESS_BOTTOM[2]	25 Main	25 Main St	Y	BOX 100	PO BOX 100	Y
AB_ADDRESS_STREET	000 Main	PO BOX 100	Y	BOX 100	000 Main	N
AB_ADDRESS_POBOX	25 Main	25 Main St	Y	blank —or— BOX 00	blank —or— BOX 00	N
AB_ADDRESS_BOTTOM[2]	000 Main	PO BOX 100	Y	BOX 100	000 Main	N

1. Output field contains a standardized address (when possible).
2. In Multiline matching, setting ADDRESS_PREFERENCE to AB_ADDRESS_BOTTOM causes the address information in the highest `Line1` input field to be returned in the `AddressLine` output field.

Address preference with two-line parsed last line input mode

`ADDRESS_PREFERENCE` with `INPUT_MODE` set to **AB_INPUT_PARSED_LASTLINE** is the same as two-line input described in the preceding section.

Address preference with parsed input mode

When AddressBroker's `INPUT_MODE` property is set to **AB_INPUT_PARSED**, only one address line can be entered at a time. AddressBroker's `ADDRESS_PREFERENCE` property has no effect.

Address preference with multiline input mode

Multiline input mode requires you to use the `Line1...6` input fields. When more than one input field contains a valid address, the preferred, standardized, match is returned in the `AddressLine` output field. City, state, and ZIP Code information is returned (standardized) in the `LastLine` output field. Any information that is not part of a matched address is returned in the `Line1...6` output fields.

Consider this example address:

25 Main Suite 200(`Line1`)

Box 100(`Line2`)

New York(`Line3`)

NY(`Line4`)

With `ADDRESS_PREFERENCE` set to prefer a street address, the output fields hold:

25 Main St Suite 200(`AddressLine`)

New York, NY, 10044-0052(`LastLine`)

null(`Line1`)

Box 100(`Line2`)

null(`Line3`)

null(`Line4`)

In the example above, both addresses are valid, but only the `AddressLine` and `LastLine` output fields are standardized. If you had set `ADDRESS_PREFERENCE` to prefer a PO Box, the output would look like this:

PO Box 100(`AddressLine`)

New York, NY, 10008(`LastLine`)

25 Main Suite 200(`Line1`)

null(`Line2`)

null(`Line3`)

null(`Line4`)

In the next example, only the PO Box is a valid address:

000 Main Suite 200(`Line1`)

Box 100(`Line2`)

New York(`Line3`)

NY(`Line4`)

When only one address matches, it is returned in the AddressLine output field, regardless of the `ADDRESS_PREFERENCE` setting. AddressBroker always attempts to return a matched address rather than no match. The output field values for this example are:

PO Box 100(`AddressLine`)

New York, NY, 10008(`LastLine`)

000 Main Suite 200(`Line1`)

null(`Line2`)

null(`Line3`)

null(`Line4`)

The first three rows in the table below show what values are returned when two valid addresses are submitted in an address record for each of the preference modes. This table also explains which output fields hold the data and whether or not the data has been processed.

The second three rows below show what values are returned when two addresses are submitted, but only one is matched. Multiline input mode:

ADDRESS PREFERENCE =	**input field = Line1**	**output field =**	**Std**[1]	**input field = Line2**	**output field =**	**Std.**[1]
`AB_ADDRESS_STREET`	25 Main	AddressLine = 25 Main St Line1 = blank	Y	Box 100	Line2 = Box 100	N
`AB_ADDRESS_POBOX`	25 Main	Line1 = 25 Main	N	Box 100	AddressLine = PO Box 100 Line2 = blank	Y
`AB_ADDRESS_BOTTOM`[2]	25 Main	Line1 = 25 Main	N	Box 100	AddressLine = PO Box 100 Line2 = blank	Y
`AB_ADDRESS_STREET`	000 Main	Line1 = 000 Main (no match)	N	Box 100	AddressLine = PO Box 100 Line2 = blank	Y

ADDRESS PREFERENCE =	input field = Line1	output field =	Std[1]	input field = Line2	output field =	Std.[1]
AB_ADDRESS_POBOX	25 Main	AddressLine = 25 Main St Line1 = blank	Y	blank —or— Box 00	Line2 = blank —or— Box 00 (no match)	N
AB_ADDRESS_BOTTOM[2]	25 Main	AddressLine = 25 Main St Line1 = blank	Y	blank —or— Box 00	Line2 = blank —or— Box 00 (no match)	N

1. Output field contains a standardized address (when possible).
2. In Multiline matching, setting ADDRESS_PREFERENCE to AB_ADDRESS_BOTTOM causes the address information in the highest Line1 input field to be returned in the AddressLine output field.

USPS enhanced line-of-travel (eLOT) codes

eLOT codes are alphanumeric codes based on the ZIP + 4 and Carrier Route of an address. eLOT codes signify the approximate order in which the postal carrier delivers mail. These codes are generally used to qualify for greater bulk mail discounts. To retrieve eLOT codes, you must input standardizable addresses; ZIP and ZIP + 4 Codes alone are not sufficient.

eLOT code information is stored in Us.gsl, shipped on the *Data Products Suite, Disc B*. Include the path to this file in AddressBroker's GEOSTAN_PATHS property to access this feature.

Your license file controls access to the eLOT codes file. To upgrade your license to include eLOT codes, contact Precisely.

B – Early Warning System Data

Early Warning System (EWS) data is a free data file the USPS provides to prevent matching errors due to the age of the address data in the Use.gsd and Usw.gsd files.

You can use the Use.gsd and Usw.gsd files that Precisely provides on the *Centrus® Data Products Suite* DVDs or Internet download for 135 days. However, during that time, the USPS may add new addresses to the Address Management System (AMS), from which the USPS address data is extracted. Precisely then adds this new data to the Use.gsd and Usw.gsd file. However, any new addresses activated after the creation of the Use.gsd and Usw.gsd files are not accessible by your Centrus products until you receive and install new data. Therefore, new addresses may be matched to a record in the current Use.gsd and Usw.gsd files that may not be the best match when the updated USPS address information is used, or the address may not match a record at all.

The USPS creates the EWS data set by examining their address database for new records that could match incorrectly to an existing record, and for addresses not present in the most recent USPS data product. The USPS creates a new EWS data set for download each week. See the *Release Notes for the Centrus Data Products* for information about downloading and installing the EWS data set.

By downloading the new EWS data on a regular basis, you can ensure more accurate address matching and standardization. If the EWS data file is present, you receive match code E028 if an input address matches to a record in the EWS data and no match is made.

C – USPS Link products

In this appendix

This appendix provides information on Delivery Point Validation (DPV), Locatable Address Conversion System process (LACS^{Link}), Suite^{Link} , and Residential Delivery Indicator (RDI) available with this Precisely product, and includes the following topics:

Note: GeoStan requires the DPV and LACS^{Link} options in CASS mode to receive ZIP + 4 and ZIP + 4 related output (DPBC, USPS record type, etc.,). GeoStan also requires the DPV Suite^{Link}, and LACS^{Link} options to produce a CASS form PS 3553.

DPV overview

Delivery Point Validation (DPV™) is a United States Postal Service (USPS®) technology that validates the accuracy of address information down to the physical delivery point. DPV is only available through a CASS-certified vendor, such as Precisely.

Previous address-matching software could only validate that an address fell within the low-to-high address range for the named street. By incorporating the DPV technology, you can resolve multiple matches and determine if the actual address exists. Using DPV reduces undeliverable-as-addressed (UAA) mail that results from inaccurate addresses, reducing postage costs and other business costs associated with inaccurate address information.

DPV also provides unique address attributes to help produce more targeted mailing lists. For example, DPV can indicate if a location is vacant and can identify commercial mail receiving agencies (CMRAs) and private mail boxes.

Although DPV can validate the accuracy of an existing address, you cannot use DPV to create address lists. DPV is a secure dataset of USPS addresses. For example, you can validate that 123 Elm Street Apartment 6 exists, but you cannot ask who lives in Apartment 6 or if there is an Apartment 7 at the same street address.

With DPV, your application automatically processes *every* ZIP+4 coded record against the DPV files. Using DPV may increase your ZIP+4 match rate, but may also increase processing time. Therefore, you may not wish to use DPV if you are not CASS certifying.

LACS^{Link} overview

The Locatable Address Conversion System (LACS) converts rural addresses to city-style addressees. LACS^{Link} is a USPS technology that provides mailers with an automated process to correct address lists for areas that have undergone LACS processing. Address list conversions occur when the LACS process modifies, changes, or replaces an address. This usually occurs due to one of the following: the conversion of rural routes and box numbers to city-style addresses, the renaming or renumbering of existing city-style addresses to avoid duplication, or the establishment of new delivery addresses.

LACS^{Link} is a secure dataset of USPS addresses. Although LACS^{Link} can validate the accuracy of an existing address, you cannot use LACS^{Link} to create address lists.

Note: LACS^{Link} is not run in multiple match searches.

False positive addresses overview

False positive addresses, also known as seed records, are addressees the USPS monitors to ensure users are not attempting to create a mailing list from the DPV or LACS^{Link} data.

Note: Per the USPS regulations, Precisely must contact the USPS with the name and address of the organization for every false positive address encountered. If multiple incidents of artificial address detection occurs, the USPS may ask Precisely to suspend a customer's DPV or LACS^{Link} processing capability.

If you encounter a false positive, you will receive a message. Processing continues to the end of your job, but further DPV or LACS^{Link} processing is disabled. DPV or LACS^{Link} processing is not available for subsequent jobs until you have reported the false-positive address encounter to Precisely and have received a new security key.

A message similar to the following appears when you encounter a false positive address:

DPV	DPV processing was terminated due to the detection of what is determined to be an artificially created address. No address beyond this point has been DPV validated. In Accordance with the License Agreement between USPS and Precisely, DPV shall be used to validate legitimately obtained addresses only, and shall not be used for the purpose of artificially creating address lists. The written Agreement between Precisely and the Precisely customer shall also include the same restriction against using DPV to artificially create address lists. Continuing use of DPV requires compliance with all terms of the License Agreement. If you believe this address was identified in error, please contact Precisely.
LACS/Link	LACS/Link processing was terminated due to the LACS/Link DEVELOPER LICENSEE PERFORMANCE REQUIREMENTS detection of what is determined to be an artificially created address. No address beyond this point has been LACS/Link processed. In accordance with the License Agreement between USPS and Precisely, LACS/Link shall be used to convert legitimately obtained addresses only, and shall not be used for the purpose of artificially creating address lists. The written Agreement between Precisely and the Precisely customer shall also include this same restriction against using LACS/Link to artificially create address lists. Continuing use of LACS/Link requires compliance with all terms of the License Agreement. If you believe this address was identified in error, please contact Precisely.

When implementing DPV and LACS^{Link} you need to create a false positive file that contains the Header Record and Detail Record information. You must provide this file to obtain a new security file from Precisely Technical Support.

For information purposes, the following tables contain the layout of the header and detail records of the false positive files for DPV and LACS^{Link}. The header record contains the mailer information from the Mailer Parameter Record and statistics gathered by the application.

Position	Length	Description	Format
1-40	40	Company name	alphanumeric
41-98	58	Address line	alphanumeric
99-126	28	City name	alphanumeric
127-128	2	State abbreviation	alphabetical
129-137	9	9-digit ZIP code	numeric
138-146	9	Total records DPV/LACS^{Link} processed	numeric
147-155	9	Total records DPV/LACS^{Link} matched	numeric
156-164	9	% match rate to DPV	numeric
165-173	9	% match rate to ZIP+4	numeric
174-178	5	Number of ZIP codes on file	numeric
179-180	2	Number of false-positives	numeric

Note: Positions 156-180 in the previous table do not exist in the LACS^{Link} false-positive header record.

The detail record contains false positive record information.

Position	Length	Description	Format
1-2	2	Street pre-directional	alphanumeric
3-30	28	Street name	alphanumeric
31-34	4	Street suffix abbreviation	alphanumeric
35-36	2	Street post-directional	alphanumeric
37-46	10	Address primary number	alphanumeric
47-50	4	Address secondary abbreviation	alphanumeric
51-58	8	Address secondary number	numeric
59-63	5	Matched ZIP code	numeric

Position	Length	Description	Format
64-67	4	Matched ZIP+4	numeric
68-180	113	Filler	

Data expiration

The USPS has determined that the ZIP+4 Directory data, DPV data, and LACS^{Link} data expire in 105 days for CASS processing. The date is measured from the release of the Postal database, which is the 15th of the month indicated on the Precisely data CD. For example, the June data release is good for 105 days from the 15th of June. However, in non-CASS processing modes, ZIP+4 data expires in 135 days.

RDI overview

The Residential Delivery Indicator (RDI™) is a United States Postal Service (USPS®) data product that identifies whether a delivery type is classified as residential or business. If you are shipping to residences, you may lower costs by shipping with the Postal Service™ and avoid residential delivery surcharges typically charged by other shipping companies.

Note: To use RDI, DPV must also be initialized.

Implementing LACS^Link and DPV

LACS^Link and DPV processing utilizes additional data. Precisely provides this data on separate CDs from the traditional GeoStan data, and is dependent on your contract with Precisely. For more information on installing DPV and LACS^Link data, see the *DPV and LACS/Link Release Notes.*

Note: DPV and LACS^Link are optional when processing records in CASS mode. However, you must use DPV and LACS^Link data for CASS certification.

When you implement DPV and LACS^Link you must first initialize GeoStan. After you have initialized GeoStan, you can initialize DPV and LACS^Link.

If you initialize DPV, GeoStan automatically uses DPV to resolve match candidatees. DPV will not delivery point validate unless you specifically request DPV output. If you do not specifically request DPV output DPV will never hit a false-positive address.

If you initialize LACS^Link, GeoStan automatically uses LACS^Link to convert addresses according to the guidelines created by the USPS.

False positive report example code

The following code is an example of how to implement DPV and LACS^Link false positive reporting using the C language:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define GEOSTAN_PROPERTIES
#include "geostan.h"
int
main(int argc, pstr * argv)
{

   GsId gs;
   GsFunStat retcode;
   char buffer[GS_MAX_STR_LEN];
   char buffer2[GS_MAX_STR_LEN];
   FILE * DPVfalsPosOut;   /* DPV false positive file pointer */
   FILE * LACSfalsPosOut;  /* LACSLink false positive file pointer */
   GsFalsePosHeaderData FPheaderData; /* DPV/LACSLink false positive
                                         header record */
   GsFalsePosDetailData FPdetailData; /* DPV/LACSLink false positive
                                         detail record */
   char * mailerName = "PRECISELY";
   char * mailerAddress = "4750 WALNUT ST STE 200";
   char * mailerCity = "BOULDER";
   char * mailerState = "CO";
   char * mailerZip = "803012532";

    PropList initProps;
```

```
PropList statusProps;
PropList findProps;
qbool bVal;

 GsPropListCreate( &initProps, GS_INIT_PROP_LIST_TYPE );

 // Replace paths and keys with your installation
GsPropSetStr( &initProps, GS_INIT_DPV_SECURITYKEY,
   "1237-5678-9abc-def0" );
GsPropSetStr( &initProps, GS_INIT_DPV_DIRECTORY,
   "C:\\Program Files\\Centrus\\Datasets\\Current" );
GsPropSetLong( &initProps, GS_INIT_DPV_DATA_ACCESS,
   DPV_DATA_FULL_FILEIO );
GsPropSetBool( &initProps, GS_INIT_DPV, TRUE );
GsPropSetStr( &initProps, GS_INIT_LACSLINK_SECURITY_KEY,
   "1237-5678-9abc-def0" );
GsPropSetStr( &initProps, GS_INIT_LACSLINK_DIRECTORY,
   "C:\\Program Files\\Centrus\\Datasets\\Current" );
GsPropSetBool( &initProps, GS_INIT_LACSLINK, TRUE );
GsPropSetStr( &initProps, GS_INIT_SUITELINK_DIRECTORY,
   "C:\\Program Files\\Centrus\\Datasets\\Current" );
GsPropSetBool( &initProps, GS_INIT_SUITELINK, TRUE );
GsPropSetLong( &initProps, GS_INIT_GSVERSION,
   GS_GEOSTAN_VERSION );
GsPropSetBool( &initProps, GS_INIT_OPTIONS_ADDR_CODE, TRUE );
GsPropSetBool( &initProps, GS_INIT_OPTIONS_Z9_CODE, TRUE );
GsPropSetStr( &initProps, GS_INIT_DATAPATH,
   "C:\\Program Files\\Centrus\\Datasets\\Current" );
GsPropSetStr( &initProps, GS_INIT_Z4FILE,
   "C:\\Program Files\\Centrus\\Datasets\\Current\\us.z9" );
GsPropSetLong( &initProps, GS_INIT_PASSWORD, 12345678 );
GsPropSetLong( &initProps, GS_INIT_CACHESIZE, 2 );
GsPropSetStr( &initProps, GS_INIT_LICFILENAME,
   "C:\\Program Files\\Centrus\\Geolib\\Geostan.lic" );

GsPropListWrite( &initProps, "stdout", NULL, 0 );
GsPropListCreate( &statusProps, GS_STATUS_PROP_LIST_TYPE );

 /* initialize GeoStan */
gs = GsInitWithProps( &initProps, &statusProps );
GsPropListWrite( &statusProps, "stdout", 0, 0 );
while ( GsErrorHas(gs) )
{

   GsErrorGetEx (gs, buffer, buffer2);
   printf ("%s\n%s\n", buffer, buffer2);
}

if ( gs == 0 )
{

   printf( "GeoStan failed to initialize.\n" );
   exit(1);
}

 // Verify DPV loaded correctly
retcode = GsPropGetBool( &statusProps,
   GS_STATUS_DPV_FILE_SECURITY, &bVal );
if ( GS_SUCCESS == retcode )
{
```

```
   if ( !bVal )
   {
      printf( "DPV security key failed to verify.\n" );
   }
   else if ( GS_SUCCESS == GsPropGetBool(&statusProps,
      GS_STATUS_DPV_FILE_ALL, &bVal) )
   {
      if ( !bVal )
         printf( "DPV data failed to initialize.\n" );
   }

}


retcode = GsPropGetBool( &statusProps,
   GS_STATUS_LACSLINK_FILE_SECUR, &bVal );
if ( GS_SUCCESS == retcode )
{
   if ( !bVal )
   {
      printf( "LACSLink security key failed to verify.\n" );
   }
   else if ( GS_SUCCESS == GsPropGetBool(&statusProps,
      GS_STATUS_LACSLINK_FILE_ALL, &bVal) )
   {
      if ( !bVal )
         printf( "LACSLink data failed to initialize.\n" );
   }

}


 retcode = GsPropGetBool( &statusProps,
   GS_STATUS_SUITELINK_FILE_ALL, &bVal );
if ( GS_SUCCESS == retcode )
{
   if ( !bVal )

      printf( "SuiteLink data failed to initialize.\n" );
}


GsPropListCreate( &findProps, GS_FIND_PROP_LIST_TYPE );
GsPropSetBool( &findProps, GS_FIND_ADDRCODE, TRUE );
GsPropSetBool( &findProps, GS_FIND_Z9_CODE, TRUE );
GsPropSetBool( &findProps, GS_FIND_FINANCE_SEARCH, TRUE );
GsPropSetLong( &findProps, GS_FIND_MATCH_MODE, GS_MODE_CASS );

GsClear( gs );
GsDataSet( gs, GS_FIRM_NAME, "DIXIES DAISYS FLOWER SERVICE");
GsDataSet( gs, GS_ADDRLINE, "74203 PERRANIA");
GsDataSet( gs, GS_LASTLINE, "GRANT CO 80448");
retcode = GsFindWithProps( gs, &findProps );

GsPropListWrite( &findProps, "stdout", 0, 0 );
while ( GsErrorHas(gs) )
{

   GsErrorGetEx (gs, buffer, buffer2);
```

```
   printf ("%s\n%s\n", buffer, buffer2);
}

 GsDataGet(gs, GS_OUTPUT, GS_DPV_FALSE_POS, buffer,
   sizeof(buffer));
if ( *buffer == 'Y' )
{

   /*
      A DPV false positive occurred.
      Write a false positive report
   */
   DPVfalsPosOut = fopen("DPVfalsePos.rpt", "w");
   if ( DPVfalsPosOut )
   {

      /* Get the false positive header data */
      memset(&FPheaderData, 0, sizeof(FPheaderData));
      strcpy(FPheaderData.MailersCompanyName, mailerName);
      strcpy(FPheaderData.MailersAddressLine, mailerAddress);
      strcpy(FPheaderData.MailersCityName, mailerCity);
      strcpy(FPheaderData.MailersStateName, mailerState);
      strcpy(FPheaderData.Mailers9DigitZip, mailerZip);
      GsDpvGetFalsePosHeaderStats(gs, &FPheaderData,
         sizeof(FPheaderData));
      /* Format the false positive header data */
      retcode = GsFormatDpvFalsePosHeader( gs,&FPheaderData,
         sizeof(FPheaderData), buffer, sizeof(buffer) );
      /* Write the header to the false positive file */
      if (retcode == GS_SUCCESS)
      {

         fprintf( DPVfalsPosOut,"%s\n", buffer );
      }
      else
      {

         GsErrorGetEx(gs, buffer, buffer2);
         printf( "Error calling GsFormatDpvFalsePosHeader:"
            "\n%s\n%s\n",                buffer, buffer2 );
      }

      /* Get the false positive detail data */
      retcode = GsDpvGetFalsePosDetail(gs, &FPdetailData,
         sizeof(FPdetailData));
      if (retcode != GS_SUCCESS)
      {

         GsErrorGetEx(gs, buffer, buffer2);
         printf( "Error calling GsDpvGetFalsePosDetail:"
            "\n%s\n%s\n",
            buffer, buffer2);
      }

      /* Format the false positive detail data */
      retcode = GsFormatDpvFalsePosDetail(gs, &FPdetailData,
         sizeof(FPdetailData), buffer, sizeof(buffer));
      /* Write the detail to the false positive file */
      if (retcode == GS_SUCCESS)
      {
```

```
            fprintf( DPVfalsPosOut,"%s\n", buffer );
         }
         else
         {
            GsErrorGetEx(gs, buffer, buffer2);
            printf( "Error calling GsFormatDpvFalsePosDetail:"
               "\n%s\n%s\n",
               buffer, buffer2);
         }

         fclose(DPVfalsPosOut);
      }
      else
      {

         printf("Failed to open DPV false positive file "
            "(errno =%d).\n", errno);
      }

   }


   GsClear( gs );
   GsDataSet( gs, GS_ADDRLINE, "RR 1 BOX 6300");
   GsDataSet( gs, GS_LASTLINE, "MOUNT SIDNEY VA 24467");
   retcode = GsFindWithProps( gs, &findProps );

   GsDataGet(gs, GS_OUTPUT, GS_LACSLINK_IND, buffer,
      sizeof(buffer));
   if (*buffer == 'F')
   {

      /*
         A LACSLink false positive occurred.
         Write a false positive report */
      LACSfalsPosOut = fopen("LACSfalsePos.rpt", "w");
      if ( LACSfalsPosOut )
      {

         /* Get the false positive header data */
         memset(&FPheaderData, 0, sizeof(FPheaderData));
         strcpy(FPheaderData.MailersCompanyName, mailerName);
         strcpy(FPheaderData.MailersAddressLine, mailerAddress);
         strcpy(FPheaderData.MailersCityName, mailerCity);
         strcpy(FPheaderData.MailersStateName, mailerState);
         strcpy(FPheaderData.Mailers9DigitZip, mailerZip);
         GsLACSGetFalsePosHeaderStats(gs, &FPheaderData,
            sizeof(FPheaderData));
         /* Format the false positive header data */
         retcode = GsFormatLACSFalsePosHeader(gs, &FPheaderData,
                                 sizeof(FPheaderData), buffer,
                                 sizeof(buffer));
         /* Write the header to the false positive file */
         if (retcode == GS_SUCCESS)
         {

            fprintf( LACSfalsPosOut,"%s\n", buffer );
         }
         else
```

```
        {
           GsErrorGetEx(gs, buffer, buffer2);
           printf("Error calling GsFormatLACSFalsePosHeader:"
              "\n%s\n%s\n", buffer,
              buffer2);
        }

        /* Get the false positive detail data */
        retcode = GsLACSGetFalsePosDetail(gs, &FPdetailData,
           sizeof(FPdetailData));
        if (retcode != GS_SUCCESS)
        {
           GsErrorGetEx(gs, buffer, buffer2);
           printf("Error calling GsLACSGetFalsePosDetail:"
              "\n%s\n%s\n", buffer,
              buffer2);
        }

        /* Format the false positive detail data */
        retcode = GsFormatLACSFalsePosDetail(gs, &FPdetailData,
           sizeof(FPdetailData), buffer, sizeof(buffer));
        /* Write the detail to the false positive file */
        if (retcode == GS_SUCCESS)
        {
           fprintf( LACSfalsPosOut,"%s\n", buffer );
        }
        else
        {
           GsErrorGetEx(gs, buffer, buffer2);
           printf("Error calling GsFormatDpvFalsePosDetail:"
              "\n%s\n%s\n", buffer,
              buffer2);
        }

        fclose(LACSfalsPosOut);
     }
     else
     {
        printf("Failed to open LACSLink false positive file"
           " (errno =%d).\n", errno);
     }

  }


  GsTerm( gs );
  GsPropListDestroy( &findProps );
  GsPropListDestroy( &initProps );
  GsPropListDestroy( &statusProps );

  return 0;
}
```

Reporting a false positive address

You report false positive address matches and obtain a new security key in the same manner for both DPV and LACS^Link. You must provide the false positive report file to Precisely to obtain a replacement security key.

To report a false positive address match and obtain a new security key:

1. Go to Precisely Support at https://support.precisely.com/.
2. Log into the site, by entering your user ID and password.

   Note: If you do not know your user ID and password, select the *Need Your User ID or Password* link. Enter your email address as instructed. Preciselywill email the user ID and password if your email address exactly matches the email in the Precisely customer database.
3. Click *My Products* from the column on the left of the Precisely Support site. A screen appears with a listing of all of your Precisely software products.
4. Click on the appropriate product name. A screen appears with the platforms available for the product.
5. Select *View Details* from the right-most column. A screen appears with detailed information for the platform.
6. Select *Download DPV* or *Download LACS^Link* in the Database section. A window appears asking you for specific information.
7. Enter your old license key and attach your false-positive file by clicking the Browse button.
8. When prompted, save the file that contains the new security key to your machine.

   You can now use the new security key located in the file you downloaded from the Precisely Support site when prompted by your application.

If you need assistance, open a Support case at https://support.precisely.com/casemanagement. Have your false-positive file ready to provide to the Precisely representative.

Understanding Suite^Link for secondary numbers

The purpose of Suite$^{Link™}$ is to improve business addressing by adding known secondary (suite) numbers to allow delivery sequencing where it would otherwise not be possible. Suite^{Link} uses the input business name, street number location, and 9 digit ZIP+4 to return a unit descriptor (i.e. "STE") and unit number for that business.

As an example, when entering the following address with Suite^{Link} enabled in CASS mode:

UT Animal Research

910 Madison Ave

Memphis TN 38103

GeoStan returns the following:

UT Animal Research

910 Madison Ave STE 823

Memphis TN 38103

Or

UT Animal Research

910 Madison Ave #823

Memphis TN 38103

If you have licensed the Suite^{Link} processing option, you must install the Suite^{Link} data and set the Suite^{Link} initialization properties for GeoStan to process your address through Suite^{Link}. For more information on Suite^{Link} enums and functions, see the following sections:

- “Enums for storing and retrieving data” on page 96 for C and page 279 for COBOL.
- GsFindWithProps properties

Suite^{Link} is required for CASS certification.

D – User-defined Data Files

In this appendix

This chapter discusses using auxiliary files and User Dictionaries.

User Dictionary

This section includes information on creating User Dictionaries, source data requirements and required fields, and other information specific to working with User Dictionaries.

Note: User Dictionaries are not for use with CASS geocoding or reverse geocoding.

Understanding User Dictionary capabilities and requirements

The capabilities of User Dictionaries and the basic requirements for creating them are as follows.

- All fields supported by normal street geocoding can be included in User Dictionaries.
- Landmarks and place names are supported in User Dictionaries. Postal or geographic centroid geocoding are not supported in User Dictionaries.
- User Dictionaries support address browsing using partial street names or landmarks and place names.
- GSDs are necessary to create the User Dictionary. This is because the GSDs have some internal structure that must be available when creating a User Dictionary.

The results from a User Dictionary are similar to that from the GSD. For address matches where the first letter of the match code would be 'S', a User Dictionary match has the letter 'J'. The value of the RecordType is 'U'. Also, the enum `DataType` returns a new value for the User Dictionary record matches.

For example: SE9 is a match code for a match that comes from a GSD, while JE9 is for a match that comes from a User Dictionary. See GeoStan location codes for a complete description of match codes.

Source data requirements

The source data for User Dictionaries includes street data but can also include place names and intersections.

To create a User Dictionary, your source data must conform to the following requirements:

- Source records must include required fields, and these fields are mapped during the User Dictionary creation process. If a value of a required field is empty for a particular record, then that record will not be imported into the User Dictionary. Required fields may vary for different countries. The MapInfo table must contain specific fields, which GeoStan then uses to convert the table into the dictionary format. These input fields are described in Required Input Fields on page.
- Source records must be in a MapInfo table (TAB file). The TAB file requirements vary for different countries.

- Segments must have two or more defined endpoints to be loaded into a User Dictionary. Segments without endpoints are ignored.
- Segments that make up intersections must have one or more end points in the intersection for GeoStan to recognize it as an intersection. Source records can be either point objects or segments.
- Each row in the table is equivalent to a street segment.

Required input fields

You must specify the field names in the MapInfo table (TAB file) in order for the table to be translated into a User Dictionary. Certain fields are required and must be present in the MapInfo table. Other fields are optional, but are strongly recommended because there may be negative consequences if they are omitted. This is described in Optional (Recommended) Input Fields on page Optional (recommended) input fields. If any of the required fields are missing, a missing field error code is returned.

The following table describes the required input fields.

Required fields	Description	Maximum field length
Left start address	Start of address range on left side of street.	10
Right start address	Start of address range on right side of street.	10
Left end address	End of address range on left side of street.	10
Right end address	End of address range on right side of street.	10
Street name	Name of street.	30
State abbreviation	Two-character state abbreviation.	2
Left ZIP Code	ZIP Code for left side of street.	5
Right ZIP Code	ZIP Code for the right side of the street.	5

Optional (recommended) input fields

The Left and Right Odd/Even Indicator fields are used to specify whether the sides of the street segment contain odd or even address ranges. Although these indicators are not required for creating a User Dictionary, it is important to use the Odd/Even Indicators when your data contains odd/even address numbers.

When the Odd/Even Indicator is specified, but is inconsistent with address numbers, the indicator is set to Both.

When the Odd/Even Indicator is not specified and both Start Address and End Address have values, the indicator is set to Both, unless the start and end address numbers are the same number. In that case, the indicator is set to Odd if the address numbers are odd, and set to Even if the address numbers are even.

When the Odd/Even Indicator is not specified and both Start Address and End Address have values, the indicator is set to Both (odd and even).

Note: If your table contains Odd/Even indicator information, we strongly recommend that you use the Odd/Even indicator fields. These fields ensure that your geocoded addresses are located on the correct side of the street. Omitting the fields when your data contains Odd/Even information may produce incorrect results.

The following table describes the optional input fields.

Optional fields	Description	Maximum field length
Left Odd/Even indicator*	Left side of the street contains only odd or even address ranges (O=odd, E=even, B=both)	1
Right Odd/Even indicator*	Right side of the street contains only odd or even address ranges (O=odd, E=even, B=both)	1
City*	City name	28
Left ZIP + 4 Code	4-digit ZIP + 4 add-on for left side of street.	4
Right ZIP + 4 Code	4-digit ZIP + 4 add-on for right side of street.	4
Left Census Block	Census Block ID for left side of street	15
Right Census Block	Census Block ID for right side of street	15
Place Name	Place name	40

* These fields are highly recommended.

User Dictionary file names and formats

GeoStan has some requirements for User Dictionary files that you must be aware of before you create a User Dictionary:

- Each User Dictionary has a base name of eight characters or fewer.
- Each User Dictionary resides in its own directory.
- The maximum length of a path to a User Dictionary is 1024 characters.
- The ZIP Code range in the MapInfo table for a User Dictionary is unlimited.

Because each User Dictionary resides in its own directory, User Dictionaries may share the same name. However, it is generally good practice to use a unique name for each User Dictionary.

Some of the output files are tied to the base name. The other output files have constant names. For example, the output files for a dictionary called ud1 are the following:

```
postinfo.jdr
postinfo.jdx
lastline.jdr
post2sac.mmj
geo2sac.mmj
sac2fn_ud.mmj
ud1.jdr
ud1.jdx
ud1.bdx
```

If your data includes place names, the dictionary contains the following files:

```
ud1.pdx
ud1.pbx
```

The dictionary also contains these log files:

```
ud1.log
ud1.err
```

Additional User Dictionary considerations

See the following topics for more information when working with User Dictionaries.

Data Access License

You must still have a valid access license to the data contained in the GSD when you are geocoding against your User Dictionary. For example, if you create a dictionary of New York streets and addresses, you must purchase the New York or entire U.S. GSD.

Use without GSD data files

To utilize a User Dictionary without the use of GSDs, the files listed below are required:

- ctyst.dir - The USPS City State table.
- parse.dir - The GeoStan dictionary

To perform postal centroid geocoding, in addition to a GSD or a User Dictionary and the files listed above, the following files are necessary:

- us.z9 - Postal centroid information.
- cbsac.dir - Required only if county names or CBSA/CSA data are needed.

CASS standards

You cannot geocode to CASS standards using a User Dictionary. This also means that the ParcelPrecision Dictionary cannot be used during CASS geocoding.

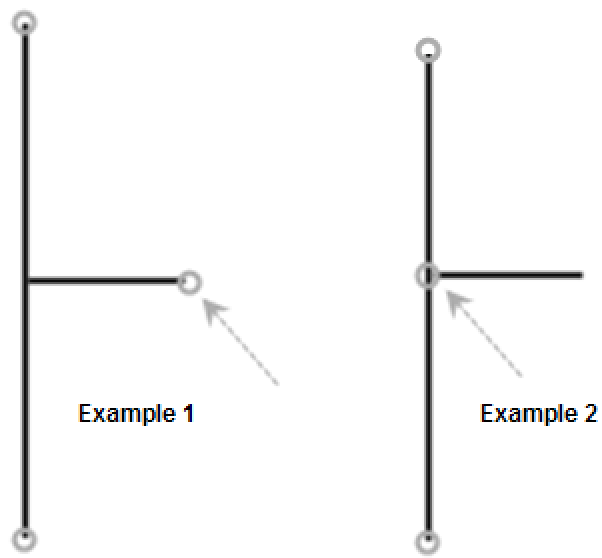
Address range order

GeoStan determines the order of the address range based on a comparison of the start and end addresses. The comparison produces the following results:

- If the end is greater than the start, the range is ascending.
- If the start is greater than the end, the range is descending.
- If the start is equal to the end, the range is ascending.

Street intersections and User Dictionaries

When geocoding to street intersections with a User Dictionary, GeoStan cannot recognize the intersections if one or more of the segments that make up the intersection does not have an end point at the intersection. This can happen when you create the User Dictionary from a customized street table in which some segments that terminate at intersections do not have end points (Example 1).



Example 1: Intersection in User Dictionary does not have end points for all segments. GeoStan does not recognize this as an intersection.

Example 2: Intersection in TIGER-based GSD includes end points for all segments. GeoStan geocodes to this intersection.

City lookup

GeoStan relies on USPS data to determine addresses. If a new address was input, it might not have been recognized despite the address being valid if it was not yet valid according to the USPS. An example of an input address that would not match against a UD:

> 1 Second Street
> Stickville, NY 11111

In this example, the city is fictitious and the zip is made up. This would fail to match even with a UD record having that city and that zip, because they are not found in the USPS data. But a user may possess a UD with such a city and zip.

When matching to a UD record, GeoStan, if necessary, corrects the city name and/or zip code to the data that is in the UD record. GeoStan is now able to obtain matches for non-USPS cities and zips that were prevented from succeeding or which required temporary workarounds.

Using User Dictionaries with address point interpolation

An important part of the process of creating a User Dictionary is to specify a mapping of fields from your source data. See the *MapInfo User Dictionary Utility Product Guide*, for a complete discussion. There are two main categories of data fields: required and optional.

Of the optional fields, there are two that have an impact on the address point interpolation feature. These are the "Left Odd/Even" and "Right Odd/Even" fields. If these are not populated, the results from address point interpolation is less accurate.

Please be aware that aforementioned fields are not populated by source data obtained via MapInfo StreetPro. You must modify the source TAB file by adding the "Left Odd/Even" and "Right Odd/Even" indicator fields, and create queries to populate them. Source data obtained from other products, or your own data, may have similar issues.

To add the "Left Odd/Even" and "Right Odd/Even" indicator fields to a source TAB file, you must add them and then run a series of SQL update queries to populate them. The fields should be filled in with “O” (odd), “E” (even), or “B” (both). Below are the steps for adding these fields:

1. Add two 1-char columns to your TAB file.

   Naming each column, for example, Ind_Right and Ind_Left.

2. Perform the following updates to populate these fields:

- Update <tablename>

  Set Ind_Left="E", Ind_Right="O"

  Where From_Left mod 2=0 AND To_Left mod 2=0

- Update <tablename>

  Set Ind_Left="O", Ind_Right="E"

  Where From_Left mod 2=1 AND To_Left mod 2=1

- Update <tablename>

  Set Ind_Left="B", Ind_Right="B"

  Where From_Left="" AND To_Left=""

Note: These example queries are simplified for illustrative purposes. Your actual queries may need to be more complex.

Auxiliary files

This chapter contains information about the GeoStan Auxiliary files feature, and includes the following topics:

- Auxiliary file matching overview
- Auxiliary file requirements
- Auxiliary file layout

Auxiliary file matching overview

Although Precisely provides robust data for you to match your input address lists against, in some cases you may want to match your address lists against speciality data. GeoStan provides you with the ability to create auxiliary files for these instances.

Creating your auxiliary files

You can create customized auxiliary files that contain records that meet your particular needs to use in GeoStan when matching address lists. This section contains information on creating auxiliary files, and contains the following topics:

- Auxiliary file requirements
- Record types
- Auxiliary file organization
- Default values

Auxiliary file requirements

GeoStan requires that the auxiliary file comply with the following:

- File must be a comma delimited, fixed width text file
  - On Windows and Unix, each record must be ASCII
  - On MVS, each record must be in EBCDIC
  - On VMS, the file name must be specified with the DDNAME and end in a number
- File must be less than 2 GB
- File must have a .gax extension
- File must have less than 500,000 records
- File must follow the column field order and lengths specified in “Auxiliary file layout” on page 484

Record types

There are two types of auxiliary file records:

- Street Records
  A street record contains a range of one or more addresses on a street. To be a valid street record the record must have the following fields:
  - ZIP Code
  - Street name
  - Street type abbreviation, if part of the address
  - Pre-directional abbreviation, if part of the address
  - Post-directional abbreviation, if part of the address
  - Low house number within the street segment
  - High house number within the street segment
  - Beginning longitude of the street segment
  - Beginning latitude of the street segment

  In addition, a street record must NOT have:
  - Unit numbers
  - Mailstops
  - Private mail boxes (PMBs)
- Landmark Records
  A landmark record represents a single site. To be a valid landmark record the record must have the following fields:
  - ZIP Code
  - Street name containing the name of the landmark
  - Beginning latitude of the street segment
  - Beginning longitude of the street segment

  In addition, a landmark record must NOT have the following fields:
  - Street type abbreviation
  - Pre-directional abbreviation
  - Post-directional abbreviation
  - Low house number
  - High house number

GeoStan ignores any record that does not comply with the preceding requirements.

Auxiliary file organization

You must comply with the following organizational rules when creating your auxiliary file.

- The first row of the auxiliary file must be the column field names.

- Use semicolons in the first column to indicate a row is a comment, not a record. GeoStan ignores rows that begin with a semicolon.
- Order the records within the file by descending ZIP Code then descending street name for optimal performance.
- All records must represent only one side of a street. To represent both sides of a street, you must create a record for each side of the street.
- All records must represent segments that are straight lines. Records cannot represent a non-straight segment.
- If house numbers are present in the record, the house number range must be valid according to USPS rules documented in Publication 28, Appendix E.
- The numeric fields, such as ZIP Codes, must contain all numbers.

Default values

GeoStan uses the following defaults if you do not include the values in the auxiliary file:

- House number parity = B (both left and right)
- Segment direction = A (ascending)
- Side of street = U (unknown)

Matching to auxiliary files

This section provides information on the matching performed by GeoStan to auxiliary files, and contains the following topics:

- Matching overview
- Record type matching rules
- Unavailable GeoStan features and functions
- Auxiliary match output

Matching overview

GeoStan performs the following steps when matching an input address to an auxiliary file:

1. GeoStan determines if there is an auxiliary file present.

   GeoStan only accepts one auxiliary file. If more than one auxiliary files is present, GeoStan attempts to match against the first file. GeoStan ignores any additional auxiliary files for matching, regardless if Geostan found a match to the first auxiliary file.

   If a record within the auxiliary files is invalid, GeoStan returns a message indicating the auxiliary file has an invalid record. GeoStan continues to process input addresses against the auxiliary file, but will not match to the invalid auxiliary file record.

2. If an auxiliary file is present, GeoStan first attempts to match to the auxiliary file.

   GeoStan assumes that the auxiliary file is the most accurate data set and first attempts to find a match to the input address in the auxiliary file. If GeoStan cannot find a match in the auxiliary file, it continues to process as normal against the traditional GeoStan datasets.

   Note: GeoStan only matches your input address lists to your auxiliary file if there is an exact match. Therefore, your input address list should be as clean as possible; free of misspellings and incomplete addresses.

3. If GeoStan finds an exact record match to the auxiliary file, it standardizes the match to USPS regulations and returns the output of the auxiliary file match.

   Note: You cannot update the auxiliary file while GeoStan is running. If you want to update the auxiliary file, you need to terminate GeoStan before attempting to replace the file.

Record type matching rules

When attempting a match against an auxiliary file, GeoStan abides by the following rules:

- Street record match
  - The house number must match the auxiliary record.
  - The house number must fall between the low and high house number values of the auxiliary record.
  - The house number must agree with the parity of the auxiliary record.
  - The ZIP Code must exactly match the ZIP Code of the auxiliary record.
- Landmark record match
  - The ZIP Code and input address line must be present and exactly match the auxiliary record.
  - The input address cannot have any other data, such as a house number, unit number, or Private Mail Box (PMB).

NOTE: GeoStan only matches the ZIP Code against the auxiliary file. GeoStan does not verify that the ZIP Code of the input address record is correct for the city and state. You should validate this information in your input lists before processing against the auxiliary file.

Unavailable GeoStan features and functions

The following contains the features and functions that do not apply when GeoStan makes an auxiliary file match.

- GeoStan does not match to

  - two-line addresses
  - multi-line addresses
  - intersection addresses
  - dual addresses
- GeoStan does not match when processing in CASS mode
- GeoStan does not perform EWS, ZIPMove, LACS^{Link}, or DPV processing on auxiliary matches
- You cannot create an auxiliary file for the reverse geocoding option

Auxiliary match output

Several standard GeoStan outputs do not apply to an auxiliary match since GeoStan matches to an exact auxiliary match and does not perform any additional validation for the match. For example, GeoStan does not return the block suffix, the check digit, or any DPV enum.

GeoStan provides special data type, match codes, and location code values for auxiliary matches. See the enum chapter for the GeoStan API you are using for more information.

When GeoStan finds a match to an auxiliary file, the default output is follows the following conventions:

- GeoStan formats the auxiliary file match as a street-style address for output. This excludes PO Boxes, Rural Routes, General Delivery, etc.
- GeoStan follows the casing setting you indicate (by default, upper case) by the casing function. GeoStan does not maintain the casing in the auxiliary file for mixed casing values. For example, GeoStan returns O’Donnell as ODONNELL or Odonnell depending on the setting of the casing function.

  Note: GeoStan does not change the casing for the User Data field.
- GeoStan removes spaces at the beginning and ending of fields in the auxiliary file.

  Note: GeoStan does not remove spaces for the User Data field.

Auxiliary file layout

The first row of the auxiliary file must be the field names. The field names must maintain the order as presented in the following table.

Field	Description	Required	Required for Street Segment Match	Exact match required if Present	Length	Position
ZIP Code	5-digit ZIP Code.	X	X	X	5	1-5
Street name	Name of the street or landmark.	X		X	30	6-35
Street type abbreviation	Street type. Also called street suffix. See the USPS Publication 28, Appendix C for a complete list of supported street types.		X	X	4	36-39
Pre-directional	USPS street name pre-directional abbreviation. Supported values are N, E, S, W, NE, NW, SE, and SW.			X	2	40-41
Post-directional	USPS street name post-directional abbreviations. Supported values are N, E, S, W, NE, NW, SE, and SW.			X	2	42-43
RESERVED	RESERVED				4	44-47
Low house number	Low house number of the address range.	X	X		11	48-58
High house number	High house number of the address range.	X	X		11	59-69
House number parity	Side of the street of the house number: *L* – Left side of the street *R* – Right side of the street *B* – Both sides of the street (*default*) *U* – Unknown side of the street	X	X		1	70
Segment direction	Direction the house numbers progress along the segment: *F* – Forward (*default)* *R* – Reverse	X	X		1	71
RESERVED	RESERVED				1	72
FIPS state	US government FIPS state code.				2	73-74
FIPS county	US government FIPS county code.				3	75-77
Census tract	US Census tract number.				6	78-83
Census block group	US Census block group number.				1	84
Census block ID	US Census block ID number.				3	85-87
RESERVED	RESERVED				5	88-92
State abbreviation	USPS state abbreviation.				2	93-94

Field	Description	Required	Required for Street Segment Match	Exact match required if Present	Length	Position
County name	Name of the county.				25	95-119
MCD code	Minor Civil Division code.				5	120-124
MCD name	Minor Civil Division name.				40	125-164
CBSA code	Core Based Statistical Area code.				5	165-169
CBSA name	Core Based Statistical Area name.				49	170-218
RESERVED	RESERVED				5	219-223
City Name	City name. Overrides the city/state preferred city name upon a return.				40	224-263
RESERVED	RESERVED				237	264-500
User-defined data	User-defined data.				300	501-800
Record ID Number	User-defined unique record identifier.				10	801-810
Side of street	Side of the street for the address: *L* – Left side *R* – Right side *B* – Both sides *U* – Unknown side (*default*) This is relative to the segment endpoints and the segment direction.				1	811
Beginning longitude	Beginning longitude of the street segment in millionths of degrees.				11	812-822
Beginning latitude	Beginning latitude of the street segment in millionths of degrees.				10	823-832
Ending longitude	Ending longitude of the street segment in millionths of degrees.				11	833-843
Ending latitude	Ending latitude of the street segment in millionths of degrees.				10	844-853

Glossary

A

address elements
The components of a street address, including house number, prefix direction, street name, street type, and postfix direction. These elements are parsed by GeoStan and should not be entered separately.

address geocoding
See geocode, geocoding.

address standardization
Address standardization is the process of taking an address and verifying that each component meets U.S. Postal Service guidelines for addresses. For example, when properly abbreviated, “123 Main Avenue” appears as “123 Main Ave.” During standardization, minor misspellings, dropped address elements, and abbreviations are corrected and the correct city, state, and ZIP Code are provided.

alias
A recognized alternate for a street name maintained by association in the database.

alias information
Data returned with certain enums when it exists. Not returned by all enums even if specifically requested.

alternate record
Additional or differing information that may be available about a specific address but that differs from the base record. See the enums table for necessary flag settings.

B

base record
The principle, rather than an alternate, record within the database.

block assignments (or blockface)
For the assignment of ZIP + 4 codes, one side of a street, from one intersection to the next.

C

carrier route
The addresses to which a carrier delivers mail. In common usage, a carrier route includes city routes, rural routes, highway contract routes, post office box sections, and general delivery units.

CASS
Coding Accuracy Support System. A service offered to mailers, service bureaus, and software vendors that improves the accuracy of delivery point codes, ZIP + 4 codes, 5-digit ZIP Codes, and carrier route information on mail. CASS provides a common platform to measure the quality of address matching software and useful diagnostics to correct software problems.

CBSA
A statistical geographic entity consisting of the county or counties associated with at least one core (urbanized area or urban cluster) of at least 10,000 population, plus adjacent counties having a high degree of social and economic integration with the core as measured through commuting ties with the counties containing the core. Metropolitan and Micropolitan Statistical Areas are the two categories of Core Based Statistical Areas.

CBSA Division
A subdivision of CBSA.

Census block ID
The 15-digit identification number used to specify a particular aggregate or block of addresses associated through census processes.

Census FIPS Code/Census ID
See FIPS code.

centroid
The calculated center of an area. The coordinates that define a centroid are the average of the sets of coordinates that describe the area.

centroid match
An address that has, through geocoding, been found to match a defined geocentroid.

city state key
A six-character USPS key that uniquely identifies a city name in the city/state file. Each city has a unique city state key.

CMSA name, CMSA number
Consolidated Metropolitan Statistical Area. The name represents the largest city in a statistical area. The number represents a 4-digit FIPS code.

County
The primary legal division of every state except Alaska and Louisiana. A number of geographic entities are not legally designated as a county, but are recognized by the U.S. Census Bureau as equivalent to a county for data presentation purposes. These include the boroughs, city and boroughs, municipality, and census areas in Alaska; parishes in Louisiana; and cities that are independent of any county in Maryland, Missouri, Nevada, and Virginia. They also include the municipios in Puerto Rico, districts and islands in American Samoa, municipalities in the Northern Mariana Islands, and islands in the Virgin Islands of the United States. Because they contain no primary legal divisions, the Census Bureau treats the District of Columbia and Guam each as equivalent to a county (as well as equivalent to a state) for data presentation purposes. In American Samoa, a county is a minor civil division.

coordinates
See latitude/longitude coordinates.

CPO
Community Post Office. A contract postal unit that provides service in small communities where independent post offices have been discontinued. A CPO bears its community's name and ZIP Code as part of a recognized address.

CSA
A geographic entity consisting of two or more adjacent Core Based Statistical Areas (CBSAs) with employment interchange measures of at least 15. Pairs of CBSAs with employment interchange measures of at least 25 combine automatically. Pairs of CBSAs with employment interchange measures of at least 15, but less than 25, may combine if local opinion in both areas favors combination.

D

datum
A mathematical model of the Earth used to calculate the coordinates on any map, chart, or survey system. Surveyors take an ellipsoid model of the Earth and fix it to a base point. The North American Datum (NAD) is the official reference ellipsoid used for the primary geodetic network in North America.

directionals

A geographic address line component that precedes (predirectional) or follows (postdirectional) the street name.

DPBC

The Delivery Point Bar Code is a POSTNET barcode that consists of 62 bars with beginning and ending frame bars and 5 bars each for the 9 digits of the ZIP + 4 code, the last 2 digits of the primary street address number (or post office box, and so on), and a correction digit. The DPBC allows automated sorting of mail to the carrier level in walk sequence.

DPC certified

Delivery point code certified. A software or hardware device that meets U.S.P.S. standards for evaluating a properly standardized ZIP + 4 code address and determines the correct 2-digit DPC and checkdigit.

E

eLOT

The Enhanced Line of Travel (eLOT) Product was developed to provide mailers the ability to sort their mailings in approximate carrier-casing sequence. To aid in mail sorting, eLOT contains an eLOT sequence number field and an ascending/descending code. The eLOT sequence number indicates the first occurrence of delivery made to the add-on range within the carrier route, and the ascending/descending code indicates the approximate delivery order within the sequence number. Mailers can use eLOT processing to qualify for enhanced carrier route presort discounts.

F

Finance Area

A Finance Area is an area defined by the U.S. Postal Service from which it collects cost and statistical data. A Finance Area is frequently used for area searches, since it covers some or all of the ZIP Code areas in a town or city.

finance number

An assigned six-digit number that identifies and installation for processing it’s financial data. The first two digits are the state code and the next four are uniquely assigned from 0001 through 9999 to each installation in alphabetical order.

FIPS code

Federal Information Processing Standards code. A FIPS Code, also called a Census ID, uniquely identifies each piece of Census geography. The syntax of the FIPS code is as follows:

`ssccctttt.ttgbbb` where:
`ss` = the two-digit State Census FIPS Code
`ccc` = the three-digit County Census FIPS Code
`tttt.tt` = the 6-digit Census Tract Census FIPS Code
`g` = the single-digit Block Group Census FIPS Code
`bbb` = the Block Census FIPS Code

G

GDT

Geographic Data Technology data. Produced by TomTom, a premium vendor of street segment files.

geocode, geocoding

A geocode is the geographic information associated with a unique address or centroid, such as longitude and latitude. Geocoding is the process of assigning data based upon location information. GeoStan uses an address or ZIP Code to assign latitude, longitude, and Census FIPS information.

GIS

Geographic Information System. A computer-based tool for enhancing geographic data by analyzing both the physical location in space and the set of characteristics associated with a location.

GSD files

GeoStan directory files.

GsEnums

Enumerated types in the GeoStan API. These enums are prefixed with “GS_” and are defined in the geostan.h file.

GSL file

USPS eLOT and Z4Change data. This files is used to assign line of travel (LOT) codes to addresses.

GSU files

GSU files contain information to match addresses based on unique ZIP Code and additional highrise unit information.

GSX files

Geographic spatial index. These files are used by spatial functions and reverse geocoding in GeoStan.

GSZ file

GeoStan ZIPMove file contains USPS ZIPMove data.

H

handle

A reference to an object that is required by the Library and is not to be manipulated directly by the developer. The handle is generated when the library is initialized and is required for many library functions.

HERE

A premium vendor of street segment and point-level data, formerly known as "NAVTEQ".

I

intersection matches

Intersections matches are indicated by an x___ match code. For example, 28th Street and Valmont intersections may be standardized and geocoded and return demographic information. Intersections do not represent a valid address for mailings.

L

LACS

Locatable Address Conversion System. This system corrects addresses electronically for areas that have undergone permanent address conversions. The address conversion occurred as a result of the 911 system implementation and involves renumbering and renaming rural route and highway contract route information as city-style addresses with street number and name.

lat/lon; latitude/longitude coordinates

Longitude and latitude coordinates are always in degrees, and are always represented as 64-bit doubles. Positive numbers represent the Eastern and Northern hemispheres, respectively, and negative numbers represent the Western and Southern hemispheres. For example, the point 140W by 30N would be represented as –140.0,30.0. The library always assumes that the longitude coordinate is the horizontal direction and the latitude coordinate is the vertical direction. Support is not provided for user coordinates.

location code

Location codes indicate the accuracy of the assigned geocode.

M

mail stop designator

This designator indicates a routing code used by a company for internal mail delivery.

MASS

Multiline (OCR) Accuracy Support System. A tool similar to Coding Accuracy Support System (CASS) that accesses and checks the address matching software used by customers' multiline optical character readers (OCRs).

match code

Indicates the portions of the address that matched or did not match with the address information in the GeoStan data files.

match mode

The algorithm used by GeoStan to match an input address to an address in the data files.

match rates

The number of input addresses that correspond (can be matched) to address information in data files.

MBR

Minimum bounding rectangle. A geographic region defined by and minimum and maximum latitude and longitude.

Metropolitan Statistical Area

A Core Based Statistical Area associated with at least one urbanized area that has a population of at least 50,000. The Metropolitan Statistical Area comprises the central county or counties containing the core, plus adjacent outlying counties having a high degree of social and economic integration with the central county as measured through commuting.

Micropolitan Statistical Area

A Core Based Statistical Area associated with at least one urban cluster that has a population of at least 10,000, but less than 50,000. The Micropolitan Statistical Area comprises the central county or counties containing the core, plus adjacent outlying counties having a high degree of social and economic integration with the central county as measured through commuting.

MSA name/number

Metropolitan Statistical Area. The name represents the name of the largest central city and the number is the 4-digit FIPS code.

match candidate resolution

The process of resolving an address match when more than one street segment has been identified as corresponding to the input address.

N

NAD

The North American Datum (NAD) is the official reference ellipsoid used for the primary geodetic network in North America.

NAD27

NAD27 has its origin at Meades Ranch, Kansas. NAD27 does not include the Alaskan islands and Hawaii. Latitudes and longitudes that are surveyed in the NAD27 system are valid only in reference to NAD27 and do not tie to any maps outside the U.S.

NAD83

NAD83 is earth-centered and defined with satellite and terrestrial data. NAD83 is compatible with the World Geodetic System 1984 (WGS84), the terrestrial reference frame associated with the NAVSTAR Global Positioning System (GPS) now used extensively for navigation and surveying. Note that TomTom uses WGS84 instead of NAD83. These two coordinate systems are compatible.

NCSC

National Customer Support Center. The U.S.P.S. CASS support center can be reached at www.usps.gov/ncsc.

O

object

A basic functional unit of a library. A library contains functions that allow the user to create, manipulate, and destroy objects. C programmers access objects through handles that are provided through object creation functions.

P

postdirectional (postdir)
See directionals.

predirectional (predir)
See directionals.

R

record matching algorithm
Programmed logic that allows evaluation of the results of all field matching algorithms to determine whether two records match (i.e., are duplicates).

road class code
A key in the street segment file that identifies a road as major or minor according to the Census Feature Classification Code.

RR
Rural Route. A delivery route served by a rural carrier.

S

soundex algorithm
A type of field matching algorithm that compares two fields based on their pronunciation.

soundex key
Generated by the GsSoundex function. Used to search the database by employing a soundex algorithm.

spatial query functions
Used to extract data from the GSD files. These functions specify the area to be searched through a minimum bounding rectangle rather than through city/state/ZIP or finance area.

stage 1 file
A sample address file provided by the U.S.P.S to determine if software/hardware meets postal requirements for CASS.

stage 2 file
An address file provided by the U.S.P.S. that is used to grade software/hardware to determine if it meets postal requirements for CASS.

street network files
Files provided by vendors (other than U.S.P.S.) the contain address and geocode information.

T

TomTom
A premium data vendor of street segment files (previously known as TeleAtlas).

TIGER files
Topographically Integrated Geographic Encoding and Referencing. A digital database of geographic features created by the US Geological Survey (USGS), covering the entire United States.

TLID
TIGER/Line® Identification Number.
The TIGER/Line® files use a permanent 10-digit TLID to uniquely identify a complete chain for the Nation. The 10-digit TLID will not exceed the value 231-1 (2,147,483,647) and represents the same complete chain in all versions of this file, beginning with the TIGER/Line® Precensus Files, 1990. The minimum value is 100,001. Topological changes to the complete chain causes the TLIDs to change. For instance, when updates split an existing complete chain, each of the new parts receives a new TLID; the old TLID is not reused.

As distributed, TIGER/Line® files are grouped by county (or statistically equivalent entity). A complete chain representing a segment of the boundary between two neighboring counties may have the same TLID code in both counties or it may have different TLID codes even though the complete chain represents the exact same feature on the ground.

U

unit designator
Indicates the type of unit (e.g., apartment, unit).

USPS data files
Files provided by the post office containing address and ZIP Code information.

Z

ZIP + 4 directory file
Address records that contain the ZIP + 4 codes for all delivery points, in an electronic form.

ZIP + 4 centroid geocoding
See geocoding.

ZIP Code
Zone Improvement Plan Code. Established in 1963 the five-digit numeric code of which the first three digits identify the delivery area of a sectional center facility or a major-city post office serving the delivery address area. The next two (the fourth and fifth) digits identify the delivery area of an associate post office, post office branch, or post office station. All post offices are assigned at least one unique 5-digit code. ZIP Code is a USPS trademark.

ZIP + 4 is an enhanced code consisting of the 5-digit ZIP Code and four additional digits that identify a specific range of delivery addresses. The nine-digit numeric code, established in 1981, composed of two parts: (a) The initial code: the first five digits that identify the sectional center facility and delivery area associated with the address, followed by a hyphen; and (b) the four-digit expanded code: the first two additional digits designate the sector and the last two digits designate the segment. ZIP + 4 is also a USPS trademark.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

O

P

R

S

T

U

W

precisely

1700 District Ave Ste 300
Burlington MA 01803-5231

precisely.com

© 1994, 2021 Precisely. All rights reserved.